

LEHRBUCH

Joachim L. Zuckarelli

# Programmieren lernen mit Python und JavaScript

Eine Praxisorientierte  
Einführung für Einsteiger

Inklusive  
SN Flashcards  
Lern-App

MOREMEDIA



Springer Vieweg

# Programmieren lernen mit Python und JavaScript

Joachim L. Zuckarelli

# Programmieren lernen mit Python und JavaScript

Eine praxisorientierte Einführung für Einsteiger



Springer Vieweg

Joachim L. Zuckarelli  
München, Deutschland

ISBN 978-3-658-29849-4

<https://doi.org/10.1007/978-3-658-29850-0>

ISBN 978-3-658-29850-0 (eBook)

Die Deutsche Nationalbibliothek verzeichnetet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.d-nb.de> abrufbar.

© Springer Fachmedien Wiesbaden GmbH, ein Teil von Springer Nature 2021  
Das Werk einschließlich aller seiner Teile ist urheberrechtlich geschützt. Jede Verwertung, die nicht ausdrücklich vom Urheberrechtsgesetz zugelassen ist, bedarf der vorherigen Zustimmung des Verlags.  
Das gilt insbesondere für Vervielfältigungen, Bearbeitungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Verarbeitung in elektronischen Systemen.

Die Wiedergabe von allgemein beschreibenden Bezeichnungen, Marken, Unternehmensnamen etc. in diesem Werk bedeutet nicht, dass diese frei durch jedermann benutzt werden dürfen. Die Berechtigung zur Benutzung unterliegt, auch ohne gesonderten Hinweis hierzu, den Regeln des Markenrechts. Die Rechte des jeweiligen Zeicheninhabers sind zu beachten.

Der Verlag, die Autoren und die Herausgeber gehen davon aus, dass die Angaben und Informationen in diesem Werk zum Zeitpunkt der Veröffentlichung vollständig und korrekt sind. Weder der Verlag, noch die Autoren oder die Herausgeber übernehmen, ausdrücklich oder implizit, Gewähr für den Inhalt des Werkes, etwaige Fehler oder Äußerungen. Der Verlag bleibt im Hinblick auf geografische Zuordnungen und Gebietsbezeichnungen in veröffentlichten Karten und Institutionsadressen neutral.

Planer: Sybille Thelen

Springer Vieweg ist ein Imprint der eingetragenen Gesellschaft Springer Fachmedien Wiesbaden GmbH und ist ein Teil von Springer Nature.  
Die Anschrift der Gesellschaft ist: Abraham-Lincoln-Str. 46, 65189 Wiesbaden, Germany

Meinen Eltern

# Einführung

---

## Die neue Welt

---

Unsere Welt war immer eine Welt von physischen Dingen – von Autos, Häusern, Möbeln und Toastern. Ihre Produktion bildete das Fundament des Wohlstands unserer Industriegesellschaften.

In den vergangenen zwei bis drei Jahrzehnten ist zu dieser physischen Welt immer stärker und mit zunehmender Geschwindigkeit und Vehemenz eine weitere, immaterielle Welt hinzugereten, die die Welt der physischen Dinge keineswegs bedeutungslos macht, aber die Funktion und das Zusammenwirken der physischen Dinge in weiten Teilen neu definiert. Darüber hinaus bringt diese immaterielle Welt ganz neue Phänomene hervor, die weitestgehend losgelöst von jeglicher physischen Grundlage existieren und dennoch unseren Wohlstand und Lebensstandard ganz entscheidend mitbestimmen.

Diese immaterielle Welt ist die Welt der *Software*.

Der Gebrauch von Software prägt in weiten Teilen längst unser Leben, auch, wenn es uns nicht ständig und vollends bewusst ist. Keine Straßenbahn fährt, kein Kühlschrank kühlt das Bier, kein Auto rollt auch nur einen einzigen Meter weit, ohne dass *Software* im Verborgenen arbeitet.

Die Welt wird digitaler und vernetzter – solche Aussagen aus dem Munde von Politikern, Verbandsvertretern oder Wirtschaftskapitänen kommen uns heute beinahe schon wie Plättitüden vor. Und doch entsprechen sie der Realität.

Was die Welt digitaler und vernetzter macht, ist *Software*. Deren Bedeutung nimmt zu und mit ihr die Bedeutung jener, die Software verstehen und entwickeln können. Software ist heute das, was die Welt im Innersten zusammenhält.

Verglichen mit der immensen Bedeutung, die Computerprogramme für unser tägliches Leben haben, wissen viele Menschen – einschließlich Entscheidungsträgern in Wirtschaft und Politik – eher wenig darüber, was Software ist und wie sie funktioniert. Jedes Kind lernt in der Schule in den Grundzügen, wie Automotoren funktionieren und warum Flugzeuge fliegen; aber wenn die Schule Grundkenntnisse des Programmierens vermitteln soll, flammt eine Diskussion darüber auf, ob das überhaupt sinnvoll und angesichts des vollgepackten Stundenplans organisatorisch realisierbar ist.

Dabei gibt es viele gute Gründe, sich näher mit Software und ihrer Entwicklung zu beschäftigen.

*Wirtschaftliche* Gründe gehören sicherlich dazu, selbst wenn man die Helden-Geschichten erfolgreicher Start-up-Unternehmer, die uns die Medien praktisch täglich präsentieren, außen vorlässt. Sowohl die Entwicklung von Software als auch die Zusammenarbeit mit Menschen, die das tun, wird in der Arbeitswelt auch so immer wichtiger, letztere sogar für Arbeitnehmer, die selbst keine Technik-Gurus sind und nie selbst in der Softwareentwicklung arbeiten werden.

*Gesellschaftliche* Gründe gehören dazu, denn die mit der steigenden Bedeutung von Software einhergehende Zunahme der Automatisierung hat Auswirkungen auf

die Arbeitswelt und unsere Rolle darin – viel größere und fundamentalere sogar, als vielen Politikern bewusst ist. Auch die gesellschaftliche Debatte um Nutzen und mögliche Risiken der Anwendung künstlicher Intelligenz ist leichter zu führen für den, der mit Grundkenntnissen der Funktionsweise von Algorithmen und Software aufwarten kann. Wer informiert politische Entscheidungen treffen will, muss verstehen, wie die Welt von Software geprägt und verändert wird.

Nicht zuletzt sprechen ganz *praktische* Gründe dafür, sich mit Software und der Entwicklung von Software zu befassen: Wer programmieren kann, macht sich das Leben an vielen Stellen deutlicher einfacher; er kann Dinge schneller, fehlerfreier und mit weniger Aufwand erledigen als andere und sogar manche Dinge tun, die sonst niemand kann. Und – auch das soll hier nicht verschwiegen werden – eine Menge Spaß ist auch dabei!

Viele von uns sind gute *Konsumenten* von Software. Dieses Buch tritt an, Ihnen die andere Seite näher zu bringen, die Seite derjenigen, die entwickeln, was wir täglich konsumieren. Sie sollen am Ende kein Profi-Programmierer sein, aber doch die Grundlagen der Entwicklung von Software verstehen und in der Lage sein, selbst Programme zu schreiben und Ihr Wissen weiter auszubauen, wenn Sie das möchten.

## Der Ansatz dieses Buches

---

Wenn Sie mit professionellen Softwareentwicklern sprechen, werden Sie häufig hören, dass diese eine ganze Reihe von Programmiersprachen – also Sprachen, in denen Computerprogramme verfasst werden – beherrschen. Einige Programmierer zählen dabei eine beeindruckend lange Liste von Sprachen mit teilweise sehr seltsam anmutenden Namen auf. Wie ist das möglich? Wie können die alle vier, fünf, sechs dieser Programmiersprachen sprechen? Sind das alles Genies?

Mitnichten. Allerdings machen sich die Programmierer einen einfachen Umstand zunutze, nämlich, dass sich Programmiersprachen in vielen Aspekten sehr ähnlich sind, viel ähnlicher als es natürliche Sprachen wie Englisch oder Spanisch untereinander sind. Viele *Grundkonzepte* sind in praktisch allen Programmiersprachen auf die eine oder andere Weise zu finden. Sie mögen in jeder Sprache anders heißen, sind aber letztlich stets nur unterschiedlich umgesetzte Varianten derselben Idee. Wer diese Grundkonzepte verstanden hat, kann sich neue Programmiersprachen schnell aneignen, denn er weiß genau, worauf er schauen sollte und muss lediglich verstehen, wie die neu zu lernende Programmiersprache das jeweilige Grundkonzept umsetzt. Das macht Sprachenlernen erheblich einfacher.

Die meisten Bücher, die eine Einführung in das Programmieren versprechen, behandeln eine spezielle Sprache – und *nur* diese eine Sprache. Sie beginnen gleich mit dieser Sprache und vermitteln alle Grundkonzepte am Beispiel dieser einen Sprache. Das Ihnen nun vorliegende Buch verfolgt einen anderen Ansatz. Nachdem wir uns in Teil 1 mit der Frage beschäftigt haben, was Programmieren eigentlich ist, warum man es lernen sollte und was es mit diesen ganzen Programmiersprachen eigentlich auf sich hat, werden wir uns erst mal mit den Grundkonzepten befassen.

Teil 2 ist ausschließlich ihnen gewidmet. Er erläutert die Grundkonzepte anhand von 9 Fragen, die Sie sich jedes Mal, wenn Sie mit einer neuen Programmiersprache beschäftigen, stellen können. Die Grundkonzepte, die im Rahmen dieser 9 Fragen behandelt werden, bilden den Dreh- und Angelpunkt des Programmieren-Lernens. Haben Sie diese Grundkonzepte einmal verstanden, können sie *jede*, wirklich *jede* Programmiersprache lernen. Auch wenn sich die Beschäftigung mit den Grundkonzepten, die in allen Programmiersprachen ähnlich sind, zunächst vielleicht eher theoretisch und trocken anhören mag, sie ist es nicht: Sie werden unzählige Beispiele aus einer ganzen Palette von Programmiersprachen sehen, damit Sie sich selbst davon überzeugen können, wie ähnlich sich die Sprachen doch letztlich sind.

Ausgestattet mit einem guten Verständnis der Grundkonzepte des Programmierens werden Sie dann im zweiten und dritten Teil des Buches zwei der populärsten Programmiersprachen überhaupt kennenlernen – *Python* und *JavaScript*. Beide Teile orientieren sich im Aufbau an unseren 9 Fragen aus dem Grundkonzepteteil. Und die Grundkonzepte sind es auch, mit deren Umsetzung in Python und JavaScript wir uns in diesen Teilen des Buches befassen. Dabei gehen wir sehr pragmatisch vor. Wir werden nicht jede Feinheit und Spitzfindigkeit der Programmiersprachen herausarbeiten. Sie sollen mit diesem Buch nicht zu einem Kenner der theoretischen Sprachdefinition werden. Sie sollen lernen, mit Python und JavaScript praktisch zu programmieren.

Wenn Sie bereits Programmiererfahrung mitbringen, könnten Sie auch direkt in den Python- oder den JavaScript-Teil einsteigen. Beide sind so aufgebaut, dass sie beinahe wie eigene Bücher gelesen werden können, selbst dann, wenn Sie den Grundkonzepteteil zuvor übersprungen haben.

Natürlich müssen Sie nicht unbedingt beide Programmiersprachen hintereinanderweg lernen. Schauen Sie sich ruhig erst einmal eine an, arbeiten Sie selbst etwas mit ihr und beginnen Sie dann mit der zweiten. Oder lernen Sie eine ganz andere Sprache! Nach dem Grundkonzepteteil werden Sie darauf sehr gut vorbereitet sein. Es wird Ihnen leichter fallen, als wenn Sie alle Grundkonzepte allein am Beispiel einer einzigen Sprache erlernt hätten.

Mit zahlreichen Übungen bieten die beiden Teile zu Python und JavaScript viele Möglichkeiten, Ihr Wissen praktisch anzuwenden und dadurch zu vertiefen. Jede Aufgabe zeigt Ihnen die ungefähre Bearbeitungszeit, mit der Sie rechnen müssen. Einige Aufgaben sind dabei einfacher, andere (die mit dem vorangestellten Ausrufezeichen) erfordern das Entwickeln größerer, zusammenhängender Programme – eine Musterlösung finden Sie aber zu allen Aufgaben gleichermaßen.

Viel Spaß bei Ihren ersten Schritten in die Welt des Programmierens!

# Inhaltsverzeichnis

---

## I Über das Programmieren

1	<b>Was ist Programmieren?</b> .....	3
1.1	Die geheimnisvolle Macht – oder: Das Bewusstsein bestimmt das Sein .....	4
1.2	Algorithmen.....	6
1.3	Grenzen von klassischen Algorithmen: Das Spielfeld der künstlichen Intelligenz..	8
1.3.1	Nur scheinbar intelligent.....	8
1.3.2	Katze oder nicht Katze – das ist hier die Frage .....	11
2	<b>Warum programmieren lernen?</b> .....	15
2.1	Viele gute Gründe.....	16
2.2	Klisches und Vorurteile.....	20
3	<b>Was ist eine Programmiersprache?</b> .....	25
3.1	Sprachen für Menschen, Sprachen für Maschinen.....	26
3.2	Übersetzung und Ausführung von Programmiersprachen .....	28
3.3	Von der Maschinensprache bis zur Hochsprache.....	30
4	<b>Warum gibt es so viele Programmiersprachen?</b> .....	33
5	<b>Welche Programmiersprachen sollte man lernen?</b> .....	39
6	<b>Einige Tipps</b> .....	45

## II Die Grundkonzepte des Programmierens

7	<b>Neun Fragen</b> .....	51
8	<b>Was brauche ich zum Programmieren?</b> .....	55
8.1	Werkzeuge .....	56
8.1.1	Compiler und Interpreter .....	56
8.1.2	Code-Editoren.....	57
8.1.3	Integrierte Entwicklungsumgebungen (IDEs).....	58
8.1.4	Einfache Online-Entwicklungsumgebungen.....	64
8.2	Hilfe und Informationen.....	65
8.3	Ihr Fahrplan zum Erlernen einer neuen Programmiersprache.....	67
9	<b>Was muss ich tun, um ein Programm zum Laufen zu bringen?</b> .....	69
9.1	Aller Anfang ist leicht .....	70
9.2	Hallo, Welt! .....	72
9.3	Ihr Fahrplan zum Erlernen einer neuen Programmiersprache.....	74

10	<b>Wie stelle ich sicher, dass ich (und andere) mein Programm später noch verstehe?</b>	75
10.1	<b>Verständlicher Programm-Code</b>	76
10.2	<b>Gestaltung des Programmcodes und Benamung von Programmelementen</b>	77
10.3	<b>Kommentare</b>	80
10.3.1	Den eigenen Programmcode erläutern	80
10.3.2	Wozu Kommentare sonst noch nützlich sind	82
10.3.3	Dokumentation außerhalb des Programmcodes	83
10.4	<b>Ihr Fahrplan zum Erlernen einer neuen Programmiersprache.</b>	85
11	<b>Wie speichere ich Daten, um mit ihnen zu arbeiten?</b>	87
11.1	<b>Variablen als Platzhalter für Daten</b>	89
11.2	<b>Datentypen von Variablen</b>	90
11.2.1	Unterschiedliche Arten von Daten erfordern unterschiedliche Arten von Variablen ..	90
11.2.2	Wichtige Datentypen	91
11.2.3	Den Datentyp ändern: Konvertierung von Variablen	96
11.3	<b>Variablen anlegen und initialisieren.</b>	98
11.4	<b>Gar nicht so variabel: Konstanten</b>	101
11.5	<b>Geordnete Felder von Variablen/Arrays</b>	101
11.6	<b>Assoziative Felder von Variablen/Hashes</b>	105
11.7	<b>Objekte</b>	106
11.7.1	Eine Welt aus lauter Objekten	107
11.7.2	Klassen	108
11.7.3	Vererbung	110
11.7.4	Methoden	113
11.7.5	Polymorphismus	116
11.7.6	Zugriffsrechte	118
11.8	<b>Ihr Fahrplan zum Erlernen einer neuen Programmiersprache.</b>	120
11.9	<b>Lösungen zu den Aufgaben</b>	120
12	<b>Wie lasse ich Daten ein- und ausgeben?</b>	123
12.1	<b>Formen der Datenein- und -ausgabe</b>	124
12.2	<b>Grafisch oder nicht grafisch – das ist hier die Frage</b>	125
12.2.1	Grafische Benutzeroberflächen	127
12.2.2	Konsolenanwendungen	135
12.3	<b>Arbeiten mit Dateien</b>	139
12.4	<b>Arbeiten mit Datenbanken</b>	145
12.5	<b>Ihr Fahrplan zum Erlernen einer neuen Programmiersprache.</b>	148
12.6	<b>Lösungen zu den Aufgaben</b>	149
13	<b>Wie arbeite ich mit Programmfunktionen, um Daten zu bearbeiten und Aktionen auszulösen?</b>	151
13.1	<b>Funktionen</b>	152
13.2	<b>Bibliotheken</b>	160
13.3	<b>Frameworks</b>	163

13.4	<b>Application Programming Interfaces (APIs) . . . . .</b>	164
13.5	<b>Ihr Fahrplan zum Erlernen einer neuen Programmiersprache. . . . .</b>	166
13.6	<b>Lösungen zu den Aufgaben . . . . .</b>	166
14	<b>Wie steuere ich den Programmablauf und lasse das Programm auf Benutzeraktionen und andere Ereignisse reagieren? . . . . .</b>	169
14.1	<b>Warum eine Ablaufsteuerung notwendig ist. . . . .</b>	171
14.2	<b>Formen der Ablaufsteuerung . . . . .</b>	172
14.3	<b>Wenn-Dann-Sonst-Konstrukte . . . . .</b>	173
14.4	<b>Ein genauerer Blick auf Bedingungen . . . . .</b>	179
14.5	<b>Komplexe Bedingungen mit logischen Operatoren (and, or, not) . . . . .</b>	181
14.6	<b>Gleichartige Bedingungen mit Verzweige-Falls-Konstrukten effizient prüfen (switch/select...case). . . . .</b>	184
14.7	<b>Ereignisse (Events) . . . . .</b>	187
14.8	<b>Ihr Fahrplan zum Erlernen einer neuen Programmiersprache. . . . .</b>	191
14.9	<b>Lösungen zu den Aufgaben . . . . .</b>	192
15	<b>Wie wiederhole ich Programmanweisungen effizient? . . . . .</b>	195
15.1	<b>Schleifen und ihre Erscheinungsformen . . . . .</b>	196
15.2	<b>Abgezählte Schleifen. . . . .</b>	198
15.3	<b>Bedingte Schleifen . . . . .</b>	204
15.4	<b>Ihr Fahrplan zum Erlernen einer neuen Programmiersprache. . . . .</b>	208
15.5	<b>Lösungen zu den Aufgaben . . . . .</b>	208
16	<b>Wie suche und behebe ich Fehler auf strukturierte Art und Weise? . . . . .</b>	213
16.1	<b>Fehler zur Entwicklungszeit . . . . .</b>	214
16.2	<b>Fehler zur Laufzeit . . . . .</b>	215
16.3	<b>Testen . . . . .</b>	217
16.4	<b>Debugging-Methoden . . . . .</b>	218
16.5	<b>Ihr Fahrplan zum Erlernen einer neuen Programmiersprache. . . . .</b>	220
<b>III</b>	<b>Python</b>	
17	<b>Einführung . . . . .</b>	223
18	<b>Was brauche ich zum Programmieren? . . . . .</b>	227
18.1	<b>Den Python-Interpreter installieren . . . . .</b>	228
18.2	<b>Die PyCharm-IDE installieren . . . . .</b>	230
18.3	<b>Hilfe zu Python erhalten . . . . .</b>	231
18.4	<b>Zusammenfassung . . . . .</b>	233
19	<b>Wie bringe ich ein Programm zum Laufen? . . . . .</b>	235
19.1	<b>Entwickeln und Ausführen von Programmen in Python . . . . .</b>	236
19.2	<b>Die Python-Konsole: Python im interaktiven Modus . . . . .</b>	241

19.3	<b>PyCharm kennenlernen</b> .....	242
19.4	<b>Zusammenfassung</b> .....	242
20	<b>Wie stelle ich sicher, dass ich (und andere) mein Programm später noch verstehe?</b> .....	245
20.1	<b>Gestaltung des Programmcodes und Namenskonventionen</b> .....	246
20.1.1	Einrückungen und allgemeine Code-Formatierung .....	246
20.1.2	Anweisungsende ohne Semikolon, Anweisungen über mehrere Zeilen.....	248
20.1.3	Case-sensitivity und Wahl von Bezeichnern.....	250
20.2	<b>Kommentare</b> .....	251
20.3	<b>Dokumentation mit Docstrings</b> .....	253
20.4	<b>Zusammenfassung</b> .....	255
21	<b>Wie speichere ich Daten, um mit ihnen zu arbeiten?</b> .....	257
21.1	<b>Variablen erzeugen und zuweisen</b> .....	259
21.2	<b>Variablen löschen</b> .....	261
21.3	<b>Grundtypen von Variablen</b> .....	261
21.3.1	Zahlen (int, float) .....	262
21.3.2	Zeichenketten (str) .....	262
21.3.3	Wahrheitswerte (bool).....	264
21.3.4	None.....	265
21.3.5	Weitere Datentypen .....	266
21.4	<b>Variablen als Objekte</b> .....	267
21.4.1	Attribute und Methoden von Variablen.....	267
21.4.2	Erzeugen von Variablen mit der Konstruktor-Methode .....	272
21.5	<b>Konvertieren von Variablen</b> .....	273
21.6	<b>Komplexe Datentypen</b> .....	275
21.6.1	Listen .....	275
21.6.2	Tupel.....	282
21.6.3	Dictionaries .....	284
21.6.4	Sets .....	287
21.7	<b>Selbstdefinierte Klassen</b> .....	289
21.7.1	Klassen definieren und verwenden .....	289
21.7.2	Klassen aus anderen Klassen ableiten .....	290
21.7.3	Doppeldeutigkeit vermeiden: Name mangling .....	292
21.8	<b>Zusammenfassung</b> .....	293
21.9	<b>Lösungen zu den Aufgaben</b> .....	296
22	<b>Wie lasse ich Daten ein- und ausgeben?</b> .....	301
22.1	<b>Ein- und Ausgabe in der Konsole</b> .....	302
22.2	<b>Grafische Benutzeroberflächen mit tkinter</b> .....	304
22.2.1	Überblick.....	304
22.2.2	Hallo tkinter!	305
22.2.3	Grafische Bedienelemente (Widgets).....	307
22.2.4	Anordnen der Bedienelemente (Geometry Managers) .....	327

22.2.5	Ereignisse .....	333
22.2.6	Beispiel: Taschenrechner-Applikation.....	337
22.3	Arbeiten mit Dateien .....	343
22.4	Übung: Entwicklung eines einfachen Text-Editors .....	346
22.5	Zusammenfassung.....	347
22.6	Lösungen zu den Aufgaben .....	348
23	<b>Wie arbeite ich mit Programmfunctionen, um Daten zu bearbeiten und Aktionen auszulösen? .....</b>	357
23.1	Arbeiten mit Funktionen .....	358
23.1.1	Definition von Funktionen.....	358
23.1.2	Funktionsargumente .....	359
23.1.3	Rückgabewerte.....	365
23.1.4	Lokale und globale Variablen .....	366
23.2	Funktionen als Klassen-Methoden von Objekten verwenden.....	370
23.3	Arbeiten mit Modulen und Packages.....	374
23.3.1	Programmcode modularisieren .....	374
23.3.2	Elemente aus Modulen importieren.....	375
23.3.3	Die Community nutzen – Der <i>Python Package Index (PyPI)</i> .....	377
23.4	Zusammenfassung.....	380
23.5	Lösungen zu den Aufgaben .....	381
24	<b>Wie steuere ich den Programmablauf und lasse das Programm auf Benutzeraktionen und andere Ereignisse reagieren? .....</b>	385
24.1	if-else-Konstrukte .....	386
24.1.1	Einfache if-else-Konstrukte .....	386
24.1.2	Verschachtelte if-else-Konstrukte .....	389
24.1.3	if-else-Konstrukte mit zusammengesetzten Bedingungen .....	390
24.1.4	if-else-Konstrukte mit alternativen Bedingungen (elif) .....	392
24.2	Ereignisse .....	394
24.3	Zusammenfassung.....	394
24.4	Lösungen zu den Aufgaben .....	395
25	<b>Wie wiederhole ich Programmanweisungen effizient? .....</b>	399
25.1	Abgezählte Schleifen (for) .....	400
25.1.1	Einfache for-Schleifen.....	400
25.1.2	Verschachtelte for-Schleifen .....	405
25.1.3	Listencomprehensionsausdrücke .....	406
25.2	Bedingte Schleifen (while).....	407
25.3	Schleifen vorzeitig verlassen und neustarten.....	410
25.4	Zusammenfassung.....	412
25.5	Lösungen zu den Aufgaben .....	413

<b>26</b>	<b>Wie suche und behebe ich Fehler auf strukturierte Art und Weise .....</b>	417
26.1	<b>Fehlerbehandlung zur Laufzeit .....</b>	418
26.1.1	<b>Fehler durch gezielte Prüfungen abfangen .....</b>	418
26.1.2	<b>Versuche...Fehler-Konstrukte (try...except) .....</b>	420
26.2	<b>Fehlersuche und -beseitigung während der Entwicklung .....</b>	422
26.2.1	<b>Haltepunkte.....</b>	423
26.2.2	<b>Anzeige des Variableninhalts und Verwendung von Watches.....</b>	425
26.2.3	<b>Schrittweise Ausführung.....</b>	426
26.3	<b>Zusammenfassung.....</b>	426
26.4	<b>Lösungen zu den Aufgaben .....</b>	427

## **IV JavaScript**

<b>27</b>	<b>Einführung .....</b>	433
<b>28</b>	<b>Was brauche ich zum Programmieren? .....</b>	437
28.1	<b>Interpreter.....</b>	438
28.2	<b>Code-Editoren und Entwicklungsumgebungen.....</b>	438
28.3	<b>Hilfe und Dokumentation .....</b>	439
28.4	<b>Zusammenfassung.....</b>	440
<b>29</b>	<b>Was muss ich tun, um ein Programm zum Laufen zu bringen?.....</b>	441
29.1	<b>Einbinden von JavaScript-Code in Webseiten.....</b>	442
29.1.1	<b>Das script-Element in HTML .....</b>	442
29.1.2	<b>Sicherheitsaspekte .....</b>	446
29.2	<b>„Hallo Welt“ in JavaScript.....</b>	447
29.2.1	<b>Do-it-yourself: Der (gar nicht so) mühsame Weg .....</b>	447
29.2.2	<b>Mit etwas Nachhilfe: Die Schnelle Umsetzung mit einem Webdienst.....</b>	449
29.3	<b>Zusammenfassung.....</b>	450
<b>30</b>	<b>Wie stelle ich sicher, dass ich (und andere) mein Programm später noch verstehe? .....</b>	451
30.1	<b>Gestaltung des Programm-Codes und Namenskonventionen .....</b>	452
30.2	<b>Kommentare.....</b>	454
30.3	<b>Zusammenfassung.....</b>	455
<b>31</b>	<b>Wie speichere ich Daten, um mit ihnen zu arbeiten?.....</b>	457
31.1	<b>Deklaration von Variablen .....</b>	459
31.2	<b>Elementare Datentypen .....</b>	460
31.2.1	<b>Zahlen (number) .....</b>	460
31.2.2	<b>Zeichenketten (string) .....</b>	462
31.2.3	<b>Wahrheitswerte (boolean) .....</b>	467
31.2.4	<b>Spezielle Typen und Werte (null, undefined, NaN).....</b>	467
31.3	<b>Konvertieren von Variablen.....</b>	469
31.3.1	<b>Implizite Konvertierung.....</b>	469

31.3.2	Explizite Konvertierung .....	470
31.4	Arrays .....	472
31.5	Objekte .....	480
31.5.1	Objektorientierung in JavaScript .....	480
31.5.2	Objekte direkt erzeugen .....	481
31.5.3	Auf Eigenschaften von Objekten zugreifen .....	482
31.5.4	Objekte mit Hilfe des Object-Konstruktors erzeugen .....	483
31.5.5	Objekte mit Hilfe von Konstruktor-Funktionen erzeugen .....	484
31.5.6	JSON .....	485
31.6	Lösungen zu den Aufgaben .....	486
31.7	Zusammenfassung .....	489
32	<b>Wie lasse ich Daten ein- und ausgeben?</b> .....	491
32.1	Überblick über die Ein- und Ausgabe in JavaScript .....	493
32.2	Ausgabe über die Konsole .....	494
32.3	Ein- und Ausgaben über Dialogboxen .....	497
32.4	Ausgabe in ein HTML-Dokument / Webseite .....	498
32.4.1	HTML-Code in die Webseite schreiben .....	498
32.4.2	Das Document Object Model (DOM) .....	500
32.4.3	DOM-Knoten über ihre Eigenschaften selektieren .....	501
32.4.4	DOM-Knoten über die hierarchische Struktur des Dokuments selektieren .....	503
32.4.5	HTML-Elemente ändern .....	506
32.4.6	HTML-Elemente hinzufügen und löschen .....	508
32.5	Eingabe mit Formularen .....	510
32.5.1	Formulare in HTML .....	510
32.5.2	Formulare aus JavaScript heraus ansteuern .....	513
32.6	<b>Beispiel: Einfacher Taschenrechner.</b> .....	517
32.6.1	Die Web-Oberfläche .....	518
32.6.2	Die CSS-Designanweisungen .....	521
32.6.3	Der JavaScript-Code .....	524
32.7	<b>Beispiel: Color Picker</b> .....	525
32.7.1	Die Web-Oberfläche .....	525
32.7.2	Der JavaScript-Code .....	528
32.8	Zusammenfassung .....	530
32.9	Lösungen zu den Aufgaben .....	531
33	<b>Wie arbeite ich mit Programmfunctionen, um Daten zu bearbeiten und Aktionen auszulösen?</b> .....	535
33.1	<b>Arbeiten mit Funktionen</b> .....	536
33.1.1	Definition von Funktionen .....	536
33.1.2	Rückgabewerte .....	541
33.1.3	Argumente und Parameter von Funktionen .....	542
33.1.4	Gültigkeitsbereich von Variablen in Funktionen .....	546
33.2	<b>Arbeiten mit Modulen/Bibliotheken</b> .....	549
33.2.1	Eigene Module entwickeln und verwenden .....	549
33.2.2	Externe Module/Bibliotheken finden und einbinden .....	551

33.3	<b>Frameworks</b> .....	552
33.4	<b>Zusammenfassung</b> .....	553
33.5	<b>Lösungen zu den Aufgaben</b> .....	554
34	<b>Wie steuere ich den Programmablauf und lasse das Programm auf Benutzeraktionen und andere Ereignisse reagieren?</b> .....	557
34.1	<b>if-else-Konstrukte</b> .....	558
34.2	<b>switch-case-Konstrukte</b> .....	561
34.3	<b>Ereignisse</b> .....	562
34.4	<b>Lösungen zu den Aufgaben</b> .....	566
34.5	<b>Zusammenfassung</b> .....	568
35	<b>Wie wiederhole ich Programmanweisungen effizient?</b> .....	571
35.1	<b>Abgezählte Schleifen (for und for..of)</b> .....	572
35.1.1	<b>for-Schleifen mit numerischen Laufvariablen</b> .....	572
35.1.2	<b>for-Schleife mit Objekt-Lauffvariable (for...of)</b> .....	579
35.2	<b>Bedingte Schleifen (while und do...while).</b> .....	580
35.3	<b>Zusammenfassung</b> .....	582
35.4	<b>Lösungen zu den Aufgaben</b> .....	582
36	<b>Wie suche und behebe ich Fehler auf strukturierte Art und Weise?</b> .....	587
36.1	<b>Fehlerbehandlung zur Laufzeit</b> .....	588
36.2	<b>Fehlersuche und -beseitigung während der Entwicklung</b> .....	590
36.3	<b>Zusammenfassung</b> .....	594
	<b>Serviceteil</b>	
	Stichwortverzeichnis .....	599

# Über das Programmieren

## Inhaltsverzeichnis

Kapitel 1	<b>Was ist Programmieren? – 3</b>
Kapitel 2	<b>Warum programmieren lernen? – 15</b>
Kapitel 3	<b>Was ist eine Programmiersprache? – 25</b>
Kapitel 4	<b>Warum gibt es so viele Programmiersprachen? – 33</b>
Kapitel 5	<b>Welche Programmiersprachen sollte man lernen? – 39</b>
Kapitel 6	<b>Einige Tipps – 45</b>



# Was ist Programmieren?

## Inhaltsverzeichnis

- 1.1     **Die geheimnisvolle Macht – oder: Das Bewusstsein bestimmt das Sein – 4**
- 1.2     **Algorithmen – 6**
- 1.3     **Grenzen von klassischen Algorithmen: Das Spielfeld der künstlichen Intelligenz – 8**
  - 1.3.1     Nur scheinbar intelligent – 8
  - 1.3.2     Katze oder nicht Katze – das ist hier die Frage – 11

## Übersicht

In diesem Kapitel beschäftigen wir uns damit, was Programmieren eigentlich ist und wozu wir Programmieren und Programmierer überhaupt benötigen. Wir werden sehen, dass es beim Programmieren um die Entwicklung bzw. Umsetzung von schrittweisen Problemlösungsstrategien, sogenannten Algorithmen, geht. Allerdings sind nicht alle Arten von Problemen, die Computer heutzutage lösen können, der Bearbeitung mit strikten Algorithmen zugänglich; im Bereich der Künstlichen Intelligenz (KI) kommen andere Herangehensweise zum Einsatz, die wir uns ebenfalls in diesem Kapitel anschauen werden.

### 1.1 Die geheimnisvolle Macht – oder: Das Bewusstsein bestimmt das Sein

Was haben Ihr Auto, Ihr Handy und Ihre Waschmaschine gemeinsam? Natürlich: Es ist sind technische Geräte und sicherlich allesamt Wunderwerke der Ingenieurkunst. Und sonst?

Alle laufen mit Software. Erst die Software ist es, die ihnen Leben einhaucht, die es Ihnen erlaubt, die Geräte zu dem Zweck zu verwenden, zu dem sie gebaut worden sind. Ohne Software wären alle diese Geräte einfach nur nutzlose Metall- und Plastikkonstruktionen ohne jegliche Funktion, ohne jeglichen Wert.

Software können wir nicht sehen, aber wir sehen ihre Auswirkungen, vermittelt durch die physischen Geräte, die *Hardware*, auf denen sie läuft. Wenn Sie bei Glätte mit Ihrem Auto ins Rutschen geraten und das Auto abfangen und wieder auf einen stabilen Kurs lenken können, dann spüren Sie die Auswirkungen des Elektronischen Stabilisierungsprogramms (ESP), das fortwährend überprüft, ob sich das Auto auch wirklich in die Richtung bewegt, die Sie mit dem Lenkrad eingeschlagen haben, nötigenfalls nachkorrigiert und Ihnen auf diese Weise hilft, das Fahrzeug unter Kontrolle zu behalten. Ein mächtiger Helfer, den man nicht sieht, der aber da ist, wenn man ihn braucht.

Die Software steuert die physischen Geräte und sagt ihnen, was sie tun sollen. Dieses Prinzip gilt für Ihr Auto, Ihre Waschmaschine und Ihr Handy gleichermaßen. Trotzdem gibt es zwei zentrale Unterschiede in Hinblick auf die Software, die auf diesen Geräten läuft:

*Erstens* nämlich ist die Software, die beispielsweise Ihre Waschmaschine steuert oder die Assistenzfunktionen Ihres Autos bereitstellt, im Funktionsumfang sehr beschränkt, weil sie nur für einen bestimmten Zweck benötigt wird. Sie werden Ihre Geschäftsbriebe oder Ihre Masterarbeit wohl nicht auf Ihrer Waschmaschine schreiben wollen, deshalb gibt es keine Software, die das erlaubt und dem physischen Gerät Waschmaschine fehlen die physischen Ein- und Ausgabesysteme, die dazu notwendig wären. Auf Handys, Tablets, und Computern (Laptops, Desktops, Servern etc.) dagegen läuft eine viel breitere Palette unterschiedlicher Software, denn die Funktion dieser Geräte ist es eben gerade, Programme auszuführen.

Zweitens gab es vom Auto und von der Waschmaschine einst auch Exemplare ohne Software, die rein mechanisch und elektrisch funktionierte. Ein Computer ohne Software allerdings würde per se keinen Sinn machen, weil seine zentrale Funktion alleine darin besteht, Programme auszuführen.

Die einfachen Beispiele der Waschmaschine und des Autos zeigen, dass Software immer mehr Geräte erfasst, die zuvor ohne Software betrieben worden sind. Diese zunehmende Digitalisierung der realen Welt erlaubt es nicht nur, immer mehr Funktionen auf den Geräten verfügbar zu machen oder bereits früher vorhandene Funktionen, präziser, sicherer, schneller oder autonomer auszuführen, sondern auch, unterschiedliche Geräte miteinander zu vernetzen. *Internet of Things*, das Internet der Dinge, ist hier das moderne Schlagwort. Wenn Ihr Kühlschrank erkennt, dass die Milch bald aufgebraucht ist und er vollkommen selbstständig online beim Laden um die Ecke zwei Tüten Milch bestellt, weil er genau weiß, dass Sie jeden Morgen Milch benötigen, um Ihr Müsli zuzubereiten, oder wenn Ihr Haus automatisch die Lamellen-Rollläden hochzieht, weil die Wettervorhersage starken Wind ankündigt, dann spüren Sie die im Verborgenen wirkende Software, die scheinbar vollkommen unterschiedliche Geräte dazu bringt, auf wundersame und scheinbar intelligente Weise zu unserem Nutzen zusammenzuarbeiten.

Trotz aller Vernetzung und aller vermeintlichen Intelligenz von Alltagsgeräten, denkt man natürlich beim Thema „Software“ vor allem am ehesten noch an den klassischen Computer und die darauf fest installierte Software, wie das Betriebssystem, die Textverarbeitung oder den Web-Browser. Tatsächlich aber ist auch fast alles, was uns täglich im Internet begegnet, letztlich Software. Selbst *statische* Webseiten, wie man sie aus der Anfangszeit des World Wide Webs kennt und mit denen man nichts anderes tun kann, als sie zu betrachten (deshalb auch als *statisch* bezeichnet), sind letztlich mit einer speziellen, beschreibenden Programmiersprache gestaltet (dazu weiter unter mehr). Erst recht steckt Software hinter den *dynamischen* Websites, die auf Eingaben der Benutzer reagieren. Wenn Sie eine Nachricht bei Facebook posten oder eine Suchanfrage bei Google starten, läuft Software, teilweise auf Ihrem eigenen Computer, teilweise auf dem Webserver, der die Seite bereitstellt.

Mit der zunehmenden Anwendungsbreite und ihrer Fähigkeit, komplexe Probleme zu lösen, hat von Software hat in den vergangenen Jahrzehnten eine rasante Entwicklung genommen. Nicht ganz so rasant ging es dagegen in der in der Frühzeit der Software zu: Das erste Gerät, auf dem ein richtiges Programm laufen konnte, war die *Analytical Engine* des englischen Universalgelehrten *Charles Babbage* in den 1830er-Jahren, ein rein mechanisch arbeitender Vorgänger des Computers. Für dieses System schrieben unter anderem der italienische Mathematiker *Federico Luigi Menabrea* und die britische Gelehrte *Augusta Ada Byron* die ersten Programme. Diese Tätigkeit brachte Byron den Ruf als erste Programmiererin der Welt ein und machte sie zu einem Gegenstand der Popkultur. Bemerkenswert an der Software, die die beiden damals entwickelten, war vor allem, dass zu dieser Zeit die passende Hardware noch gar nicht existierte. Die Arbeit der beiden frühen Programmierer fand mit Papier und Tinte statt, denn Babbages *Analytical Engine*

wurde zu seinen Lebzeiten nie gebaut, weil der *British Association for the Advancement of Science*, die das Vorhaben hätte finanzieren sollen, die Kosten zu unsicher und die Erfolgsaussichten zu nebulös waren. Tatsächlich wurde eine voll funktionsfähige Analytical Engine *Analytical Engine* erst in den 1990er-Jahren gebaut, als bereits seit Jahrzehnten viel leistungsfähigere Computer zur Verfügung standen.

Zum Speichern der Programme (und ebenso der Daten, mit denen sie arbeiteten) waren Lochkarten vorgesehen. Diese Technik fand erstmals Anfang des 19. Jahrhunderts im sogenannten *Jacquard-Webstuhl* Anwendung, einer Webmaschine, die mit Hilfe der Lochkarten auf das Weben bestimmter Muster „programmiert“ werden konnte.

Hier sehen Sie auch die Parallele zu unserer Diskussion um Waschmaschine und Computer: Babbages *Analytical Engine* war ein System, das ausschließlich dazu entwickelt worden ist, Software auszuführen, der Webstuhl des Seidenwebers *Joseph-Marie Jacquard* dagegen war wie die Waschmaschine ein Gerät, dass auch grundsätzlich auch ohne Software seine Funktion erfüllen kann, durch Programme aber vielseitiger einsetzbar und einfacher zu bedienen wird.

## 1.2 Algorithmen

---

Software ist heutzutage allgegenwärtig. Sie steuert die Geräte und sagt ihnen, was sie tun sollen.

Das ist ziemlich offensichtlich bei physischen Geräten wie der Waschmaschine, die eine ganze Reihe automatischer Waschprogramme beherrscht, oder Ihrem Auto, das dank entsprechender Assistenzfunktion vollkommen selbstständig in eine Parklücke manövriert. Aber auch bei Computersoftware ist es so: Die Software weist den Computer an, auf dem Bildschirm etwa ein YouTube-Video anzuzeigen und den passenden Ton über die Lautsprecher abzuspielen. Software ermöglicht es Ihnen, über die Tastatur Ihr Twitter-Passwort einzugeben, oder in der Textverarbeitung durch Klick auf einen Button einige Wörter in fette Schrift zu setzen.

Die Software erteilt den Geräten also Anweisungen. Einen Satz von mehreren solchen Anweisungen bezeichnet man als *Algorithmus*.

Ein Algorithmus ist nichts weiter als eine Arbeitsvorschrift, die erklärt, wie man ein bestimmtes Ziel erreicht. Algorithmen kennen wir aus unserem Alltag, auch wenn wir die Arbeitsvorschriften und -abläufe, die uns dort begegnen, üblicherweise nicht als Algorithmen bezeichnen.

Ein Beispiel eines solchen Alltagsalgorithmus, der es zum Ziel hat, ein gutes Pesto herzustellen, lautet zum Beispiel:

1. 80 g Pinienkerne, 4–5 Bund Basilikum (pesto alla genovese) und/oder Rucola, eine Knoblauchzehe, 50 g Parmesan, 50 g Pecorino sardo oder Pecorino romano, 180 ml Olivenöl, sowie Salz (vorzugsweise grobes) bereitstellen.
2. Basilikum waschen, die Blätter abzupfen und trockentupfen.
3. Die Knoblauchzehe schälen, den Keim entfernen und grob hacken.
4. Parmesan und Pecorino reiben (keinen bereits geriebenen Käse verwenden).

5. Pinienkerne, Knoblauch und Basilikumblätter mit Salz in einen Mörser geben und fein zerstoßen.
6. Den Käse untermengen.
7. Langsam das Olivenöl zugeben, nicht zu viel auf einmal, damit man es gut unter das Gemisch arbeiten kann.

Wenn Sie diesen Anweisungen folgen, werden Sie am Ende aus den im ersten Schritt genannten Zutaten ein gutes Pesto hergestellt haben.

Ein anderes Beispiel für Algorithmen, wie sie uns im Alltag ständig begegnen ist das folgende:

1. Auf Dienerstraße nach Osten Richtung Marienplatz (230 m).
2. Weiter auf Residenzstraße (300 m).
3. Links Richtung Theatinerstraße abbiegen (47 m).
4. Rechts abbiegen auf Theatinerstraße (130 m).

Dieser Algorithmus beschreibt, wie man als Fußgänger in München vom Odeonsplatz zum Marienplatz gelangt.

Als letztes Beispiel noch ein Alltagsalgorithmus, der das Aufstehen und Fertigmachen an einem normalen Werktag beschreibt:

5. Aufstehen.
6. Zähneputzen.
7. Duschen.
8. Anziehen.
9. Instant-Kaffee in Tasse anrühren.
10. Solange Schluck aus Kaffeetasse trinken, bis Tasse leer.
11. Schuhe anziehen.
12. Wenn es kalt ist, Jacke anziehen.
13. Wohnung verlassen.
14. Tür abschließen.

Von den beiden vorangegangenen Algorithmen unterscheidet sich dieser hier zum einen dadurch, dass nicht alle Schritte immer durchlaufen werden; manche Schritte („... Jacke anziehen“) werden nur durchlaufen, wenn bestimmte Bedingungen erfüllt sind („Wenn es kalt ist ...“). Zum anderem werden manche Schritte wiederholt: Schritt 6 besagt, dass wiederholt ein Schluck aus der Kaffeetasse genommen wird, und zwar so lange, bis die Tasse leer ist. Bedingungen und Wiederholungen (oder „Schleifen“, wie der Programmierer sagen würde) sind wichtige Konzepte der Programmierung, mit denen wir uns später noch ausführlicher beschäftigen werden.

Sie sehen am Beispiel dieser einfachen Anweisung für das Aufstehen, dass wir auch in unserem Alltagsalgorithmen oft Elemente wie Bedingungen und Schleifen benutzen, die ebenso beim Programmieren Anwendung finden. Grundsätzlich scheinen die Alltagsalgorithmen und die Algorithmen, die Programmierer entwickeln, nicht so verschieden zu sein. Trotzdem gibt es einige – mitunter sehr wichtige – Unterschiede.

Was sind aber nun die Unterschiede zwischen diesen Alltagsalgorithmen und den Algorithmen, die von Computern verarbeitet werden? Wir hatten bereits gesagt, dass die Alltagsalgorithmen üblicherweise nicht als „Algorithmen“ bezeichnet werden – achten Sie einmal auf das Gesicht des Chefkochs in einem Restaurant, wenn Sie ihm nach einem vorzüglichen Essen freudestrahlend ein Kompliment machen und ihn wissen lassen, dass er „da wirklich einen ganz hervorragenden Tartuffo-Algorithmus eingesetzt hat“.

Die Unterschiede zwischen Alltagsalgorithmen und Computer-Algorithmen sind aber tatsächlich noch tiefgreifender. Computer-Algorithmen laufen auf dem Gerät Computer. Alltagsalgorithmen laufen auf dem „Gerät Mensch“. Dieses System ist intelligent und füllt Lücken, die der Algorithmus möglicherweise hat, automatisch mit dem auf, was an dieser Stelle Sinn macht. Lässt der Algorithmus etwa ganz offensichtlich einen (vielleicht trivialen) Zwischenschritt weg, der eigentlich notwendig ist, dann erkennen wir diese Unvollständigkeit und führen den fehlenden Schritt einfach trotzdem aus, obwohl er im Algorithmus nicht angegeben ist. Das tun Computer nicht. Computer sind Maschinen, die genau das tun, was man ihnen sagt. Sie denken nicht mit und verfügen über keinerlei Intelligenz, die es ihnen erlauben würde, Unvollständigkeiten und Fehler im Algorithmus selbstständig zu erkennen und mit dem, was an dieser Stelle Sinn macht, aufzufüllen bzw. zu ersetzen. Deswegen dürfen Computer-Algorithmen keine Lücken und keine Fehler haben. Sie müssen ungleich präziser sein als Algorithmen, die dafür gedacht sind, auf dem „Gerät Mensch“ zu laufen. Alles muss haarklein und korrekt beschrieben sind, damit der Computer den Algorithmus tatsächlich ausführen kann.

Wenngleich sich auch Alltagsalgorithmen und Computer-Algorithmen in diesem Punkt ganz fundamental voneinander unterscheiden, so haben sie doch eines gemein: Sie sind letztlich nur Abfolgen von Arbeitsschritten, um ein bestimmtes Ziel zu erreichen, ganz unabhängig davon, worin dieses Ziel besteht – ob es darum geht ein fabelhaftes Pesto zu zaubern oder einen Facebook-Post abzusetzen.

Damit lässt sich nun auch die Grundfrage, die wir uns in diesem Kapitel stellen, beantworten: Programmieren ist nicht mehr und nicht weniger als der Vorgang, Algorithmen zu entwickeln und so aufzuschreiben, dass der Computer sie ausführen kann.

## 1.3 Grenzen von klassischen Algorithmen: Das Spielfeld der künstlichen Intelligenz

---

### 1.3.1 Nur scheinbar intelligent

---

Computer können viele Dinge sehr gut, die uns Menschen schwerfallen oder bei denen wir schnell an unsere Grenzen stoßen, wie der deutschstämmige Silicon-Valley-Milliardär und Gründer von PayPal, Peter Thiel, argumentiert. Komplizierte Berechnungen ausführen beispielsweise, und das in einer atemberaubenden Geschwindigkeit und ohne dabei auch nach Stunden nur ansatzweise zu ermüden.

Trotzdem, so Thiel weiter, scheitern Computer regelmäßig an Aufgaben, die selbst Kleinkinder mühelos bewältigen, etwa an der Aufgabe, zu erkennen, ob das in einem YouTube-Video dargestellte Tier eine Katze ist, oder nicht. Warum ist dieses scheinbar simple Problem für Computer so schwer zu bewältigen?

Versuchen Sie einmal, einen Algorithmus aufzuschreiben, der prüft, ob ein Bild eine Katze zeigt. Sie werden vermutlich mit den offensichtlichen Eigenschaften einer Katze beginnen, zum Beispiel den Ohren oder den Schnurrbarthaaren. Sie können nun aber dem Computer nicht einfach befehlen, „Erkenne, ob auf dem Bild spitze Ohren zu sehen sind!“. Woher soll er wissen, was ein spitzes Ohr ist? Also werden Sie anfangen, spitze Ohren zu beschreiben. Vielleicht nähern Sie sich dem Beschreibungsproblem mit der geometrischen Form eines Dreiecks. Auch die Farbe ist sicher ein gutes Unterscheidungsmerkmal. Sie werden aber schnell feststellen, dass Katzenohren nicht wirklich ganz dreieckig sind, erst recht nicht, wenn die Katze von der Seite gefilmt wird. Außerdem wird Ihr Computer den Dachgiebel eines grau-braunen Hauses mit Reetdach sehr schnell für ein Katzenohr halten. Je mehr sie sich darin vertiefen, zu beschreiben, welche Merkmale eine Katze hat und wie sich diese von ähnlichen Merkmalen anderer Gegenstände, die keine Katze sind, unterscheiden, werden Sie feststellen, wie unglaublich schwer es ist, einen echten Algorithmus zu entwickeln, der erkennt, ob wir es mit einer Katze zu tun haben, oder nicht.

Warum aber fällt uns Menschen das Erkennen einer Katze so leicht? Wir erkennen die Katze immer, selbst wenn wir sie nur von hinten oder im Gegenlicht sehen. Die Antwort ist einfach: Unser Gehirn arbeitet nicht algorithmisch. Es führt nicht schrittweise eine Reihe von Befehlen aus, um das Problem „Katze erkennen“ systematisch zu lösen. Es arbeitet vollkommen anders.

Ein ganzer Zweig der Informatik beschäftigt sich mit dem abstrakten Nachbilden dieser Funktionsweise des menschlichen Gehirns im Computer. Die Fähigkeit eines Computers, Leistungen zu reproduzieren, von denen wir typischerweise ausgehen, dass sie Intelligenz voraussetzen, nennt man konsequenterweise *künstliche Intelligenz* (oder kurz auch einfach nur KI oder AI, nach dem englischen Begriff *artificial intelligence*).

Künstliche Intelligenz ist mittlerweile ein Modewort geworden, überall und in den verschiedensten Zusammenhängen hört man davon. Vielen Menschen macht die Vorstellung, Computer könnten „intelligent“ sein, Angst. Ein umfangreiches Angebot spannender Science-Fiction-Filme lehrt uns, dass wir aufpassen sollten, was wir im Bereich künstliche Intelligenz entwickeln, immerhin könnten unsere Schöpfungen eines Tages die Kontrolle übernehmen und uns überflüssig werden lassen.

Die Realität ist indes um einiges trister und weniger aufregend. Von der Vorstellung, wir könnten Systeme entwickeln, die selbstbewusst (also sich *ihrer eigenen Existenz bewusst*) sind und die wie ein Mensch *denken*, haben sich die meisten ernstzunehmenden Wissenschaftler längst verabschiedet. Diese Form der künstlichen Intelligenz, sogenannte *starke KI*, über die etwa der Androide *Data* in der erfolgreichen Fernsehserie *Star Trek – Das Nächste Jahrhundert* mit seinem Positronenhirn zweifelsfrei verfügt, bleibt weiterhin den Science-Fiction-Autoren vorbehalten.

Sehr viel realistischer dagegen ist es, Software zu entwickeln, die *auf einzelnen Gebieten* Leistungen erzielt, die mit den Leistungen von Menschen auf denselben Gebieten vergleichbar sind. In diesem Zusammenhang spricht man in Abgrenzung zur starken KI von *schwacher KI*. Anwendungen solcher künstlichen Intelligenz gibt es viele. Die fortgeschrittenen Schach- oder Go-Computer schlagen jeden noch so brillanten menschlichen Spieler um Längen. Das (teil-)autonome Autofahren wäre ohne die von künstlicher Intelligenz gestützte Bilderkennung und die darauf aufbauende Erkennung von Straßenverhältnissen, Straßenverläufen und anderen Verkehrsteilnehmern nicht möglich. Die Spracherkennung, wie sie etwa Apples Assistenzsystem Siri bietet, basiert ebenso auf künstlicher Intelligenz wie Googles Bildersuche oder der Mechanismus, der Ihnen YouTube-Videos vorschlägt, die Sie interessieren könnten.

Einige dieser Anwendungen von künstlicher Intelligenz sind in der gesellschaftlichen Diskussion durchaus umstritten, selbst, wenn nicht die Gefahr besteht, dass die Weltherrschaft übermorgen von hochintelligenten Maschinen übernommen und der Mensch zu unnützer Biomaterie degradiert wird.

Dafür gibt es mindestens vier Gründe:

- Weil es künstliche Intelligenz erlaubt, menschliche Leistungen zu reproduzieren, die wir typischerweise mit Intelligenz in Verbindung bringen, öffnet sich ein breites Feld von Anwendungen, die bisher allein menschlichem Entscheiden, Bewerten und Urteilen vorbehalten waren. Wir fühlen uns nicht wohl dabei, diese Entscheidungen an (im Grunde doch vollkommen unintelligente) Systeme zu delegieren, die das Wirken menschlicher Intelligenz letztlich nur simulieren. Es ist ja eben gerade der Kerngedanke der schwachen künstlichen Intelligenz, dass menschliche Intelligenzleistungen auf einem eng beschränkten Gebiet reproduziert werden, ohne dabei wirkliches *Denken* in der Maschine hervorzurufen. So verwundert es nicht, wenn Menschen skeptisch sind, die Ihr Leben plötzlich einem autonom fahrenden Auto anvertrauen sollen, sich auf Diagnosen verlassen sollen, die nicht ein Arzt, sondern eine „intelligente“ Software gestellt hat, oder die Richtersprüche akzeptieren sollen, die kein menschlicher Richter, sondern ein darauf spezialisiertes Programm verfasst hat. Ein wenig relativiert sich die Situation aber, wenn man sich andere Situationen in der Geschichte anschaut, in denen Technologie plötzlich die Tätigkeit von Menschen übernahm. Im Rahmen der Apollo-11-Mission der NASA kam 1969 erstmals ein Steuercomputer zum Einsatz, anfänglich sehr zum Missfallen der betroffenen Astronauten, die sich partout nicht vorstellen konnten, ihr Leben in die Hände des von einer jungen Mathematikerin namens *Margaret Hamilton* und ihrem Team entwickelten Steuerungsprogramms zu legen. Tatsächlich steuerte die Software die Raumfähre dann auch nicht vollautomatisch als Autopilot, sondern fungierte eher wie ein Assistenzsystem für die menschlichen Piloten, die sich an die Arbeit mit der neuen Technologie gewöhnten. Es tritt also eine Gewöhnung ein, wenn wir merken, dass die Systeme ihren Zweck erfüllen und wir Vertrauen in ihre *Funktionsüchtigkeit* entwickeln, selbst, wenn wir ihre Arbeitsweise nicht wirklich verstehen.
- Ein weiteres Problem von Anwendungen künstlicher Intelligenz stellen neben dem notwendigen Vertrauen in ihre Funktionsfähigkeit auch *ethische Erwägungen*.

gen dar. Ein klassisches, hypothetisches Beispiel ist das autonom fahrende Auto, dass in einer kritischen Verkehrssituation nur die Auswahl zwischen zwei moralisch strenggenommen inakzeptablen Situationen hat, von denen jede den Tod oder die schwere Verletzung eines anderen Verkehrsteilnehmers zur Folge hätte. Dieser Fall lässt sich noch einigermaßen gut dadurch „lösen“, dass das Auto einfach die Variante wählt, die dem „betroffenen“ Verkehrsteilnehmer die höchsten Überlebenschancen einräumt. Der Autopilot schätzt die Situation im Zweifel erheblich schneller und besser ein als der geschockte und vollkommen überforderte menschliche Fahrer. Trotzdem gibt es nicht wenige, die aus prinzipiellen Erwägungen heraus postulieren, solche Entscheidungen müssten grundsätzlich dem Menschen vorbehalten sein. Gleichermaßen wird mitunter für Entscheidungen im sozialen Kontext, wie etwa Gerichtsurteilen, gefordert.

- Darüber hinaus erlaubt künstliche Intelligenz Anwendungen, die dazu verwendet werden können, *Kontrolle über andere* auszuüben, entweder durch Überwachung oder dadurch, dass unser Verhalten beeinflusst wird. In den falschen Händen, würden solche Technologien mächtige Werkzeuge für Despoten und selbst für eigentlich demokratische Politiker in demokratischen politischen Systemen darstellen. Die gesellschaftliche Diskussion über vermeintliche Wahlmanipulationen bei den Präsidentschaftswahlen in den Vereinigten Staaten im Jahr 2016 und die Rolle sozialer Medien in der demokratischen Auseinandersetzung sind noch vergleichsweise harmlose Beispiele für solcherlei Phänomene.
- Schließlich ist es wahrscheinlich, dass künstliche Intelligenz in manchen Bereichen den Menschen tatsächlich ersetzen wird, was zu einem sozialen Problem führt, da Menschen ihre ursprüngliche Tätigkeit, für die sie auch ausgebildet worden sind, nicht mehr ausüben können. Dies wird vor allem deshalb als erhebliches Problem betrachtet, weil die ersten Bereiche, in denen künstliche Intelligenz den Menschen tatsächlich vollständig ersetzen wird, sicherlich jene sein werden, die heute von Menschen mit eher geringer Qualifikation bearbeitet werden. Diese Menschen wiederum werden es schwerer haben, eine alternative Beschäftigung zu finden als Höherqualifizierte.

Auch wenn also manche Anwendungen von künstlicher Intelligenz nicht umstritten sind, wird ihre Verbreitung und Zahl der Einsatzbereiche weiter zunehmen. Die Möglichkeiten, die künstliche Intelligenz bietet, sind einfach zu interessant und zu verlockend.

### 1.3.2 Katze oder nicht Katze – das ist hier die Frage

---

Am Beispiel der Erkennung einer Katz auf einem Bild haben wir festgestellt, dass künstliche Intelligenz offenbar ganz anders arbeitet als herkömmliche Algorithmen. Und in der Tat arbeiten viele KI-Ansätze eben nicht mit einer algorithmischen Abfolge von Arbeitsschritten, sondern mit einer Form der Mustererkennung, die der Funktionsweise des menschlichen Gehirns nachempfunden ist. Dabei werden Signale über mehrere Schichten von künstlichen „Neuronen“ geleitet. Wegen der „Neuronen“ spricht man auch von einem *neuronalen Netz*. Diese Neuronen

sind wie kleine Netzwerknoten mit anderen Neuronen verbunden. An die Neuronen, mit denen sie verbunden sind, können sie einen Signalimpuls weitergeben. Ob das geschieht oder nicht, und wenn ja, wie stark, hängt zunächst einmal davon ab, ob zwischen den betreffenden Neuronen überhaupt eine Verbindung besteht. Falls dies der Fall ist, ist die Stärke des Impulses, den das eine Neuron an das nächste Neuron weitergibt, davon abhängig, wie „dick“ die Verbindung zwischen den Neuronen und wie stark das Signal, das das erste Neuron selbst von seinem jeweiligen „Vorgänger“ empfangen hat, ist. Die letzte Schicht von Neuronen ist die Ausgabeschicht. Hier wird das Ergebnis dieser mehrschichtigen Verarbeitung „angezeigt“.

In unserem Katzenbeispiel treffen die Informationen über das Bild, also die einzelnen Bildpunkte mit ihrer Position im Bild und ihren Farbwerten auf die Eingangsschicht von Neuronen. Diese geben die Impulse an die nächste Schicht weiter, entsprechend der Dicke der Verbindungen und der Stärke des Impulses, den sie selbst bekommen haben. In der letzten Schicht gibt es nur noch zwei Neuronen, die die Ergebnisse „Katze“ und „nicht Katze“ repräsentieren. Das Ergebnis, also die Signalstärke, die am Ende bei den beiden Ausgangsneuronen ankommt, hängt also offenbar von der Verteilung der Signale in der Eingangsschicht (das heißt also dem Bild, das wir analysieren) und der Dicke der Verbindungen zwischen den Neuronen ab. Aber woher weiß das neuronale Netz eigentlich, wie diese Verbindungsstärken beschaffen sein müssen, damit am Ende wirklich erkannt werden kann, ob das Bild, das wir in die Eingangsschicht geben, eine Katze darstellt, oder nicht? Antwort: Gar nicht. Wir müssen es ihm sagen. Etwas präziser gesagt: Wir „trainieren“ das Netzwerk mit Katzenbildern und Bildern, die keine Katzen zeigen. Wir wissen ja, welches Ergebnis am Ende das richtige ist. Mit Hilfe des Ergebnisses, den das bis dahin untrainierte (oder zumindest noch nicht voll trainierte) Netzwerk liefert, und einem speziellen Algorithmus lassen sich die Verbindungsichten zwischen den Neuronen optimal anpassen. Danach geht es zum nächsten Trainingsbild. Mit der Zeit und tausenden von Bildern wird das neuronale Netz auf diese Weise immer besser darin, die Katzenbilder zu erkennen.

Das bemerkenswerte an dieser Technik ist, dass wir am Ende nicht sagen können, warum genau das neuronale Netz überhaupt in der Lage ist, die Katzenbilder von den Nicht-Katzenbildern zu unterscheiden. Das Netzwerk besteht aus einer riesigen Menge von Neuronen und Verbindungen zwischen diesen Neuronen sowie den Dicken dieser Verbindungen (sogenannte Gewichte). Schaut man sich diese Parameter an, sieht man dem neuronalen Netz nicht an, dass es dafür geeignet ist, eine Katze zu erkennen. Tatsächlich haben wir nicht die geringste Ahnung, *warum* das Netzwerk funktioniert. Die Parameter haben sich einfach durch die Trainings-Sitzungen ergeben, sie wurden von einem ganz klassischen Algorithmus auf Basis des Unterschieds zwischen dem vom Netzwerk errechneten Ergebnis und dem erwünschten, das heißt, richtigen Ergebnis, ermittelt bzw. schrittweise mit jedem Trainingsbild nachjustiert. Das neuronale Netz ist also eine Black Box. Wo wir beim herkömmlichen Algorithmus genau nachvollziehen können, wie er zu diesem oder jenem Ergebnis gekommen ist, sehen wir beim neuronalen Netz nur eine verwirrende Menge von Parametern, die nicht sinnvoll interpretiert werden können.

Der Vollständigkeit halber sei erwähnt, dass nicht alle Ansätze künstlicher Intelligenz mit neuronalen Netzen arbeiten. So gibt es schon seit Jahrzehnten Sys-

### 1.3 · Grenzen von klassischen Algorithmen: Das Spielfeld der...

teme, die auf Basis von Wenn-Dann-Regeln und damit von herkömmlichen Algorithmen, das Wissen eines Experten auf einem bestimmten Gebiet zur Verfügung stellen und praktisch in einem Frage-Antwort-Spiel Menschen etwa dabei unterstützen, eine komplizierte medizinische Diagnose zu stellen oder die Fehlfunktion an einem Motor zu verstehen. Diese Systeme werden, weil sie über das in expliziten Regeln dokumentierte Wissen eines menschlichen Experten auf diesem Gebiet verfügen, auch als *Expertensysteme* bezeichnet.



# Warum programmieren lernen?

## Inhaltsverzeichnis

- 2.1      Viele gute Gründe – 16**
- 2.2      Klischees und Vorurteile – 20**

## Übersicht

Sie haben sich dazu entschieden, dieses Buch zu lesen, also müssen Sie offenbar nicht mehr davon überzeugt werden, sich mit dem Programmieren zu beschäftigen. Trotzdem lohnt es sich, sich in diesem Kapitel noch einmal klar zu vergegenwärtigen, warum es Sinn macht, Programmieren zu lernen.

Auch, wenn (berufsmäßige) Programmierer in unserer modernen Welt manchmal wie die Superstars unter den Wissensarbeitern erscheinen mögen, schlagen ihnen doch häufig Vorurteile entgegen, die nicht selten von überzeichneten Klischees geprägt sind. Einige dieser Vorurteile begegnen Ihnen manchmal auch als Hobby-Programmierer. Mit solcherlei Vorurteilen und Klischees beschäftigen wir uns in diesem Kapitel ebenfalls.

### 2.1 Viele gute Gründe

Einige Gründe dafür, dass es sich lohnt, Programmieren zu lernen, sind die folgenden:

- **Machen Sie Ihren Alltag einfacher!**

Programmieren zu können, erlaubt es Ihnen, Dinge zu automatisieren, die Sie anderenfalls mühsam und fehleranfällig von Hand erledigen müssten. Das steigert nicht nur Ihre Produktivität, sondern macht das Arbeiten auch entspannter, denn es sind natürlich in besonderem Maße die langweiligen, monotonen, repetitiven, kurzum: nervigen Aufgaben, die sich besonders gut durch Programme automatisieren lassen.

Microsoft Excel ist heute das Schweizer Taschenmesser des Büroarbeiters. Mit einigen Ausnahmen wie etwa Rechtsanwaltskanzleien, deren Geschäftsmodell nicht zuletzt im Abfassen elaborierter Schriftsätze besteht, wird Excel (und in erheblich geringerem Maße auch Tabellenkalkulationsprogramme anderer Hersteller, zum Beispiel Google) heute praktisch überall und zu den unterschiedlichsten Zwecken eingesetzt, von der Budgetplanung über Business Cases bis hin zu Projektplänen und Aufgabenlisten. Vielen Benutzern ist dabei gar nicht bewusst, dass Excel (wie auch die übrigen Microsoft-Office-Anwendungen) von Haus aus mit einer leistungsfähigen Programmiersprache, *Visual Basic for Applications* oder kurz *VBA*, daherkommt. Diese Programmiersprache erlaubt es, auch komplizierte Vorgänge zu automatisieren. Angenommen etwa, Sie haben mehrere Fragebögen einer Kunden- oder Mitarbeiterbefragung in unterschiedlichen Excel-Dateien vorliegen und wollen diese nun in einer einzigen Arbeitsmappe konsolidieren. Das können Sie natürlich von Hand tun, indem Sie jede Datei öffnen, die Werte ablesen oder kopieren und in Ihrer konsolidierten Arbeitsmappe wieder einfügen. Schneller und sicherer (also mit weniger Fehlerrisiko) geht es, wenn Sie sich ein kleines VBA-Programm schreiben, dass die Aufgabe einfach automatisch ausführt. Nun mögen Sie vielleicht einwenden, dass es ja auch Zeit kostet, das VBA-Programm zu entwickeln und zu testen, bis es richtig funktioniert. Und in der Tat dauert das Ent-

## 2.1 · Viele gute Gründe

wickeln so eines kleinen Makros, wie man die VBA-Programme auch nennt, oftmals länger, als man es ursprünglich veranschlagt hat. Trotzdem ist die Herausforderung, die Aufgabe zu automatisieren um Längen spannender als das stumpfsinnige, händische Hin- und Herkopieren der Daten. Vollends deutlich wird der Vorteil des VBA-Programms, wenn Sie die Aufgabe später einmal wiederholen müssen, zum Beispiel, weil neue Befragungsergebnisse vorliegen.

Versuchen Sie doch mal, Ihre Kollegen und nicht zuletzt auch Ihren Chef damit zu beeindrucken, wie schnell und mühelos Sie viele Aufgaben erledigen, die den meisten als zeitintensive Horror-Vorhaben erscheinen. Drew Houston, einer der Gründer des Cloud-Dienstes Dropbox hat einmal gesagt: „Programming is the closest thing we have to a super power“. Auch in den Aussagen vieler anderer Programmierer findet sich das Motiv wieder, Programmieren habe etwas Magisches. Doch natürlich kommt es nur den Nicht-Eingeweihten magisch vor. Die „Magier“ selbst wissen, wie die „Magie“ funktioniert, und dass ihre „Kräfte“ alles andere als übernatürlich sind.

Dass Programmieren-Können das Leben manchmal einfach etwas angenehmer macht, hat bereits Microsoft-Gründer Bill Gates in jungen Jahren unter Beweis gestellt. Er und sein Freund und späterer Microsoft-Mitgründer Paul Allen griffen dergestalt in die Stundenplansoftware von Gates' High School ein, dass sich dieser auf einmal in einer Klasse wiederfand, die fast nur aus Mädchen bestand. Dass er mit keiner der Mitschülerinnen richtig anbandeln konnte, wie er selbst berichtet, lag zumindest nicht an seinen Programmier-Künsten.

### ■ Sprechen Sie „IT“!

Es ist eine mittlerweile tausendfach wiederholte, aber deswegen nicht minder zutreffende Beobachtung, dass IT in Unternehmen und anderen Organisationen, einschließlich des Öffentlichen Dienstes, immer wichtiger wird. Kaum eine moderne Organisation, deren Art, ihr Geschäft zu betreiben, von der fortschreitenden Digitalisierung nicht beeinflusst wird. Manchmal ändert sich dabei sogar das Geschäftsmodell selbst. Die Folge ist natürlich, dass die IT-Abteilung intern an Bedeutung gewinnt, eben weil es immer mehr Themen und Probleme gibt, die nur in Zusammenarbeit mit der internen IT oder externen IT-Beratern, die zu diesem Zweck an Bord genommen werden, bearbeitet und gelöst werden können. Auch wenn Sie nicht selbst in der IT-Abteilung arbeiten, sondern in einer klassischen „Fachabteilung“, im Vertrieb, zum Beispiel, oder im Controlling, Sie werden mit Projekten und Vorhaben in Berührung kommen, zu denen die IT einen maßgeblichen Beitrag liefert. Deshalb ist ein Verständnis davon, wie „die IT“ in den Grundzügen funktioniert, was technisch lösbar ist und was nicht, und welche Schritte grob zur Lösung eines Problems notwendig sind, von ungemeinem Vorteil. Das gilt erst recht und in besonderem Maße, wenn Sie eng in Projekte eingebunden sind, die eine IT-Komponente haben. Die Zeit, in der hunderte Seiten umfassende, komplizierte Fachkonzepte verfasst wurden, die dann von der IT mehr schlecht als recht interpretiert und in eine technische Lösung umgesetzt wurden, sind vorbei. Auf dem Vormarsch sind sogenannte agile Methoden wie SCRUM, bei denen die Entwicklungszyklen kürzer und die (internen) „Kunden“, also die fachlichen Anforderer einer technischen Lösung, viel enger als früher in deren Konzeption, Umset-

zung und Test einbezogen sind. Anders ausgedrückt: Die Zusammenarbeit mit IT wird intensiver und wichtiger und dementsprechend auch das gegenseitige Verständnis. Davon bringen Sie als Programmier-Kundiger erheblich mehr mit als andere, und es wird Ihnen mit diesem Verständnis viel leichter fallen, mit Ihren IT-Kollegen auf Augenhöhe zu diskutieren, selbst dann, wenn Sie von den konkreten Technologien, die diese einsetzen, überhaupt keine Ahnung haben. Viel wichtiger nämlich, als eine bestimmte Technologie zu beherrschen, ist es, zu erkennen, was lösbar (zum Beispiel im Sinne von Automatisierung) ist, und was nicht, die Herangehensweise an die Problemlösung zu verstehen und die wichtigsten Schritte vorausdenken zu können. So werden Sie zum geschätzten fachlichen Partner Ihrer IT-Kollegen und bringen Ihre fachlichen Themen wirkungsvoll voran!

Und übrigens, falls Sie nach höheren Weihen streben: Die Zeiten, in denen sich Angehörige der mittleren und höheren Führungsebene, einschließlich der Geschäftsführung, beim Ansprechen von IT-Themen ob der Belästigung mit vermeintlichen operativen Unwichtigkeiten pikiert abwenden oder gar unverhohlen mit Ihrer Unwissenheit kokettieren konnten, neigen sich dem Ende entgegen. Nicht mehr viele werden wohl in Zukunft damit auskommen, gänzlich ohne IT-Verständnis wilde Strategien auf schicke Präsentationsfolien zu bannen.

### ■ Verdienen Sie Geld!

Wir haben soeben argumentiert, dass IT immer wichtig wird und bereits heute immense Bedeutung hat. Das spiegelt sich auch in den Arbeitsmarktdaten wider: Laut Institut für Arbeitsmarkt- und Berufsforschung der Bundesanstalt für Arbeit waren 2017 allein in der Berufsgruppe „Softwareentwicklung und Programmieren“, jener Berufsgruppe, für die das, was Sie in diesem Buch lernen, sicherlich am relevantesten ist, 198.000 Menschen sozialversicherungspflichtig beschäftigt, ein knappes Drittel mehr als 2013 (12,7 % der Beschäftigten waren übrigens Frauen, ein Anteil der über die vergangenen Jahre verhältnismäßig konstant ist). Mit einem Durchschnittseinkommen von 4658 EUR brutto im Monat oder 55.896 EUR im Jahr (2017) gehören die Jobs in dieser Berufsgruppe auch nicht gerade zu den am schlechtesten vergüteten in Deutschland. Offensichtlich also ein attraktives Betätigungsfeld, wenn Sie sich beruflich noch nicht festgelegt haben, oder sich umorientieren wollen.

Aber natürlich muss man nicht eine sozialversicherungspflichtige Festanstellung anstreben, wenn man seine Programmierkenntnisse zu Geld machen möchte. Auf verschiedenen Internet-Plattformen bieten Freelancer ihre Dienste an. Die populäre Plattform *Upwork* zum Beispiel weist zum Zeitpunkt, zu dem diese Zeilen geschrieben werden, in den Kategorien „Desktop Software Development“, „Mobile Development“ und „Web Development“ 22.559 Projekte aus, die von den jeweiligen Auftraggebern an Freelancer vergeben werden sollen. Auf diesen Plattformen tummeln sich natürlich viele Softwareentwickler aus Ländern mit niedrigerem Lohnniveau, die dementsprechend überschaubare Vergütungen zu akzeptieren bereit sind, obwohl sie mitunter gut ausgebildet sind. Um erfolgreich auf Plattformen wie Upwork zu sein, müssen Sie sich eine gute Reputation aufbauen, denn Sie akquirieren neue Kunden nicht zuletzt mit Hilfe der Bewertungen, die Sie von früheren Kunden erhalten haben. Das macht die Sache besonders am

## 2.1 · Viele gute Gründe

Anfang natürlich schwer. Hier mag aber ein Weg sein, zunächst mit niedrigeren Honoraren zu punkten und Ihre Fähigkeiten auf andere Art als durch vergangene Kunden-Projekte unter Beweis zu stellen, etwa durch private Programmier-Vorhaben, die Sie über Plattformen wie *GitHub* veröffentlichen und so Ihren potentiellen Kunden zeigen können. Trotz aller Schwierigkeiten, die Freelancing über solche Internet-Plattformen insbesondere zu Beginn mit sich bringen mag, ist es aber trotzdem eine exzellente Möglichkeit, sich praktisch risikolos und ohne Ihren Hauptberuf an den Nagel hängen zu müssen, auszuprobieren und Ihre neu erworbenen Fähigkeiten am Markt zu testen.

Schließlich können Sie sich natürlich auch von der Auftragsarbeit für Einzelkunden lösen und eigene Software entwickeln, die Sie an einen Kundenkreis vertrieben. Ob das, selbst wenn Sie eine gute Idee haben und die notwendigen Fähigkeiten mitbringen, noch neben Ihrem eigentlichen Beruf möglich ist, kann sicher nur im Einzelfall und unter Berücksichtigung der eigenen Risikovorliebe entschieden werden. Etliche Beispiele zeigen allerdings, dass es durchaus möglich ist, auch in einem überschaubaren Rahmen (es muss ja nicht gleich das nächste Facebook sein) mit eigener unternehmerischer Tätigkeit im Software- und Internetgeschäft Erfolg zu haben.

### ■ Verstehen Sie, was die Welt im Innersten zusammenhält!

Bisher haben wir eher von den profanen Dingen des Lebens gesprochen. Davon, wie Sie sich mit Programmieren das Leben leichter machen, ihre beruflichen Projekte voranbringen und Geld verdienen können. Wenn Sie das alles nicht wirklich interessiert, und Sie stattdessen das hehre Ziel verfolgen, mehr darüber zu lernen, wie unsere komplexe Welt funktioniert, kommen Sie wiederum am Programmieren nicht vorbei.

Würde Goethes Dr. Faust in unserer Zeit leben, er würde keinen Pakt mit dem Teufel schließen. Er würde programmieren lernen.

Unsere Welt wird zunehmend durch die vermeintlich geheimnisvolle Macht der Algorithmen gelenkt. Obwohl das so ist, fehlt vielen Menschen ein Verständnis davon, was Algorithmen sind und wie sie funktionieren (ein Verständnis, das Sie nach der Lektüre des letzten Kapitels hoffentlich schon aufgebaut haben!). Laut der Bertelsmann-Studie *Was Deutschland über Algorithmen denkt und weiß* aus dem Jahr 2018 erinnern sich zwar 72 % der Befragten, den Begriff schon einmal gehört zu haben (das heißt im Umkehrschluss: jeder vierte hat noch nichts von Algorithmen gehört!), 56 % geben aber gleichzeitig an, sie wüssten kaum etwas über Algorithmen. Dafür assoziieren 37 % der Befragten mit Algorithmen Begriffe wie „unheimlich“ und „unverständlich“, 50 % immerhin auch mit „Fortschritt“. Chancen von Algorithmen werden interessanterweise eher von denen gesehen, die angeben, eine genauere Vorstellung davon zu haben, was Algorithmen sind, die Einschätzung der Risiken dagegen ist unter den Befragten vergleichbar, unabhängig von den (selbst eingeschätzten) Vorkenntnissen.

Wir leben also in einer Welt, die zunehmend von Technologien und Problemlösungsansätzen geprägt ist, die viele nicht verstehen. Das war früher anders. Die Schule hat jedem ein mehr oder weniger solides Wissen darüber vermittelt, wie die wichtigsten Technologien der jeweiligen Zeit funktionierten. Wer in der Schule auf-

gepasst hat, wusste in den Grundzügen, was ein Auto antreibt, woher die Elektrizität kommt, und warum arbeitsteilige Industrieproduktion Effizienzvorteile bietet. Mittlerweile hat die öffentliche Bildung aber den Anschluss verloren. Wir kommen auf diese Thematik weiter unten noch einmal zurück.

Wenn Sie sich eine fundierte Meinung über Chancen und Risiken von Algorithmen bilden und an der gesellschaftlichen Diskussion darüber teilnehmen wollen, müssen Sie mehr davon verstehen, als es viele Deutsche laut der Bertelsmann-Befragung tun. Ist es dazu notwendig, dass Sie selbst programmieren können? Nein, mit Sicherheit nicht. Aber ein weit besseres Verständnis hat auf jeden Fall derjenige, der schon mal selbst ein kleines Programm entwickelt hat.

### ■ Schulen Sie Ihr logisches, problemlösendes Denken!

Es wird oft gesagt, dass man ein besseres Verständnis von Grammatik entwickelt, wenn man in der Schule Latein lernt, dessen rein kommunikativer Nutzen (außer vielleicht im Vatikan, wo man wahrscheinlich auch gut mit Italienisch durchkommt) eher überschaubar ist. Dass das Studium des Lateinischen ein besseres Verständnis von Grammatik als solcher bedingt, liegt daran, dass man sich, wenn man lateinische Texte übersetzen will, tatsächlich mehr mit der Grammatik beschäftigen muss als in anderen Sprachen, etwa dem Englischen. Man lernt auf diese Weise die Grundkonzepte von Grammatik kennen, zum Beispiel die unterschiedlichen Fälle und ihre Verwendung, die sich dann auch auf andere Sprache anwenden lassen. So hat das Lateinlernen sogar dann einen Nutzen, wenn Sie nicht zu denjenigen gehören, die berufsbedingt wenigstens ein Minimum an Lateinkenntnissen vorweisen können müssen, wie etwa Historiker oder Mediziner.

Ähnlich verhält es sich mit dem Programmieren. Bei kaum einer anderen Beschäftigung werden Sie so gut wie beim Programmieren lernen, Probleme systematisch anzugehen, sie in einfachere Teilprobleme zu zerlegen, einen Ansatz für jedes der Teilprobleme zu entwickeln und schließlich aus den Lösungen der Teilprobleme die Lösung für Ihre ursprüngliche Fragestellung zusammenzusetzen. Die Fähigkeit, Probleme systematisch anzugehen, hilft Ihnen nicht nur beim Programmieren selbst, sondern überall im Leben. Wer programmieren lernt, sieht die Welt anders und geht Probleme gleich welcher Art strukturierter und lösungsorientierter an. Vielleicht ist diese Schule für das logische Denken sogar der wichtigste Grund überhaupt, sich mit Programmieren zu beschäftigen.

## 2.2 Klischees und Vorurteile

---

Nachdem wir einige gute Gründe diskutiert haben, dererwegen es Sinn macht, sich mit dem Programmieren zu beschäftigen, sollten wir uns an dieser Stelle nun auch noch kurz die Zeit nehmen, um mit einigen Klischees und Vorurteilen aufzuräumen, die dem Programmieren in den Augen vieler anhaften.

### ■ Programmieren ist nur etwas für Nerds

Manch einer hat noch immer das Bild des verpickelten, Brille tragenden, männlichen Teenagers im Kopf, dessen Grundnahrungsmittel Pizza und Cola sind und

der wie besessen und ohne Unterlass die ganze Nacht auf seine Tastatur einhämmt. Ja, diese Art von Programmierern gibt es mit Sicherheit auch. Und es ist nicht schwer, zu erkennen, warum dieses zugespitzte Klischee natürlich auch einen Funken Wahrheit beinhaltet. Zum einen ist da nämlich das „Suchtpotential“. Man kann, wenn man entsprechend veranlagt ist, schnell ganz und gar in ein Programmierproblem „hineingezogen“ werden und alles um sich herum vergessen, weil man vollkommen konzentriert alles daransetzt, dieses eine Problem zu lösen, das partout nicht weichen will. In dieser Situation kann Zeit schnell zweitrangig werden. Man will es einfach lösen, weil es einem vollkommen unbefriedigend vorkäme, das Problem ungelöst zurückzulassen, obwohl es, wie der Erfahrung zeigt, im Sinne der Problemlösung vielfach besser wäre, sich erst einmal auszuruhen und sich am folgenden Tag mit frischem Verstand das Problem noch einmal vorzunehmen.

Neben dem Suchtpotential gibt es noch einen zweiten Aspekt, der implizit in dem oben beschriebenen Klischee enthalten ist: Programmieren als Ersatz für echte soziale Interaktion. Das Tolle am Programmieren ist, dass man etwas entwickelt und sodann vom Computer ein Feedback bekommt: Das Programm läuft, oder es läuft eben nicht. Es ist leicht vorstellbar, dass dies für Menschen, die eher introvertiert und in sozialen Situationen zurückhaltend sind, ein interessanter Anreiz ist. Hinzu kommt die perfekte Kontrollierbarkeit: Wenn man das Programm richtig schreibt, tut der Computer genau das, was man ihm befohlen hat, und nichts anderes. Soziale Interaktionen sind erheblich weniger kontrollierbar, haben immer Elemente von Unvorhergesehenem, Überraschendem, vielleicht sogar Zufälligem. Diese Kernelemente sozialer Interaktion kann man beim Programmieren einfach ausschalten. Man befindet sich in einer perfekt kontrollierbaren Situation.

Die Aspekte Suchtpotential und Kontrollierbarkeit spielen mit Sicherheit nur für einen kleinen Teil aller Programmierer eine Rolle. Die meisten Programmierer sind ganz normale Menschen. Der kaufmännische Angestellte, der sich mit Hilfe von VBA einige Arbeitsschritte in Excel automatisiert, um seinen Arbeitsalltag zu vereinfachen, steht mitten im Leben und wird sich von Ihnen vielleicht kaum unterscheiden. Gleiches gilt für diejenigen, die von Berufs wegen mit der Entwicklung von Software beschäftigt sind. Sie müssen also keineswegs befürchten, Mitglied einer gesellschaftlichen Randgruppe zu werden, wenn Sie sich intensiver mit dem Programmieren beschäftigen.

#### ■ **Software und ihre Programmierung ist nur eine Mode, ein Hype**

Mancher mag denken, dass die Softwarebranche nur die Ausgeburt eines modernen Hypes ist. Am Ende zählen nicht Programme, sondern immer noch ein gutes, brauchbares physisches Produkt, so das Argument. Wer aber so denkt, unterschätzt die bedeutende Rolle, die Software mittlerweile in unser aller Leben spielt. Nicht nur sind viele Produkte entstanden, die ausschließlich aus Software bestehen (Büroanwendungen und soziale Netzwerke sind Beispiele dafür), sondern die Art, wie physische Produkte konzipiert, hergestellt, vertrieben und monetarisiert werden, ändert sich durch Software in tiefgreifender Weise; denken Sie etwa an die Wirkungen, die der Online-Versandhändler Amazon auf den Buchhandel ausübt. In einigen Branchen wie der Automobilindustrie, gibt es starke Anzeichen dafür, dass Software als Teil des Gesamtprodukts relativ zu seinen physischen Kompo-

nenten erheblich an Bedeutung gewinnt. Vielleicht ist es in Zukunft interessanter, was Ihr Auto dank der Software, die es betreibt, so alles kann, als die Frage, ob es Alufelgen und Sportsitze hat. Insbesondere für diejenigen, für die das Auto primär ein Fortbewegungsmittel ist (nicht natürlich die Autoenthusiasten) könnte das physische Auto mehr und mehr austauschbar werden als eine Plattform, auf der die wirklich interessanten Features in Form von Software laufen.

Ja, physische Produkte (und nicht unmittelbar software-bezogene Dienstleistungen natürlich auch) werden weiterhin Bedeutung haben, aber sie werden stärker von Software abhängen, und so ihren Benutzern mehr Funktionen und damit Nutzen anbieten zu können. Auch die Art, physische Produkte zu erzeugen und zu vertrieben, ändert sich mitunter radikal, ebenso wie die Bedeutung von reinen Software- und Internetprodukten erheblich zunehmen wird.

### ■ Programmieren ist nur etwas für Männer

Es mag zwar sein, dass Männer im Durchschnitt ein größeres Interesse an technischen Themen mitbringen (woran auch immer das liegen mag), aber Interesse am Programmieren vorausgesetzt, gibt es natürlich keinen Grund zu der Annahme, dass Frauen nicht ebenso begabte Programmierer sein können, oder dass es für die berufliche Entwicklung von Frauen weniger nützlich wäre, des Programmierens mächtig zu sein.

Tatsächlich haben Frauen in der Geschichte der Software-Programmierung immer wieder eine besondere Rolle gespielt. Die englische Mathematikerin *Augusta Ada Byron* beschrieb im 19. Jahrhundert als erste ein vollständiges Programm für die von Charles Babbage erdachte Lochkartenmaschine, weshalb sie weithin als die erste Programmiererin der Welt betrachtet wird (Babbages *Analytical Machine* wurde jedoch zu Byrons und Babbages Lebzeiten wegen Finanzierungsschwierigkeiten nie gebaut). Sie erkannte auch – und das ist vermutlich noch weit bedeutsamer – dass analytische Maschinen wie Babbages System grundsätzlich zu mehr als nur dem Umgang mit Zahlen fähig sind und stattdessen auch mit anderen Arten von Symbolen (wie etwa Buchstaben) arbeiten könnten. Weiterhin erdachte sie Konzepte wie die Schleife (Wiederholungen von Programmanweisungen), die heute Standard in praktischen allen modernen Programmiersprachen sind. Als Ehrerweisung trägt die Programmiersprache *Ada* ihren Namen.

Neben Ada Lovelace haben noch weitere Frauen wichtige Beiträge zur Entwicklung des Programmierens geleistet. Beispielhaft sei hier die Computerpionierin *Grace Hopper* genannt, die bereits in den 1940er-Jahren die bahnbrechende Idee hatte, Programme in einer für den Menschen unmittelbar verständlichen Sprache zu verfassen und wesentlich an der Entwicklung von *COBOL* (*Common Business Oriented Language*) beteiligt war, einer der ersten modernen Hochsprachen. Sie gilt außerdem als Erfinderin des Compilers, der das in einer (eben für den Menschen verständlichen) Programmiersprache verfasste Programm in die Maschinensprache übersetzt, die der Computer unmittelbar ausführen kann (mehr dazu im nächsten Kapitel).

Oder die amerikanische Mathematikerin *Margaret Hamilton*, die mit ihrem Team die Steuerungssoftware für das Apollo-11-Programm entwickelte und dabei

## 2.2 · Klischees und Vorurteile

in gewissem Sinne das Assistenzsystem erfand, mit dem Neil Armstrong und Buzz Aldrin 1969 sicher auf dem Mond landeten und wieder heimkehrten.

Frauen haben einen ganz entscheidenden Beitrag zur Entwicklung des Programmierens geleistet und vieles von dem, was wir heute als ganz normal hinnehmen, überhaupt erst möglich gemacht. In der Geschichte war Programmieren immer auch Frauensache. Warum sollte das heute anders sein?



# Was ist eine Programmiersprache?

## Inhaltsverzeichnis

- 3.1 Sprachen für Menschen,  
Sprachen für Maschinen – 26
- 3.2 Übersetzung und  
Ausführung von  
Programmiersprachen – 28
- 3.3 Von der Maschinensprache  
bis zur Hochsprache – 30

## Übersicht

Programmieren bedeutet letztlich, dem Computer Anweisungen zu erteilen. Das geschieht in einer speziellen Sprache, die sowohl wir als Programmierer als auch der Computer versteht: einer Programmiersprache. Programmiersprachen und natürliche Sprachen haben sehr viele Gemeinsamkeiten, weisen aber auch einige wichtige Unterschiede auf.

In diesem Kapitel beschäftigen wir uns mit Programmiersprachen und damit mit der entscheidenden Frage, wie wir uns dem Computer gegenüber verständlich machen, und wie der Computer es schafft, uns zu verstehen und unsere Anweisungen auszuführen.

### 3.1 Sprachen für Menschen, Sprachen für Maschinen

Bisher haben wir uns damit beschäftigt, *was* Programmieren eigentlich ist und *warum* es Sinn macht, es zu lernen. Nur die Frage des *Wie* haben wir bisher ausgelassen: Wie programmiert man denn nun eigentlich? Wie genau teilen wir dem Computer mit, wie der Algorithmus, also die Folge von Arbeitsschritten, die er abarbeiten soll, aussieht?

Irgendwie muss der Algorithmus in einer dem Computer verständlichen Art und Weise aufgeschrieben werden. Nun arbeiten Computer ja bekanntlich in einem binären Modus, sie unterscheiden letztlich nur zwei Zustände, Null und Eins, An und Aus, Spannung vorhanden und Spannung nicht vorhanden. Es liegt daher nahe, zu vermuten, dass wir Programme in dieser binären Weise aufschreiben müssen, damit der Computer sie versteht. Das ist glücklicherweise nicht der Fall.

Wenn man einem Programmierer über die Schulter schaut, sieht man, dass er Texte in einer für den Menschen grundsätzlich lesbaren Sprache verfasst. Im Text erkennt man viele englische Begriffe. Eine typische Zeile eines solchen Programmtextes könnte etwa lauten:

```
if value < 0 then print "Wert ist kleiner als null"
```

Auch wenn man viele der verwendeten Begriffe aus dem Englischen kennt, so ist doch die Sprache, die der Programmierer benutzt, eine *künstliche Sprache*, eine *Programmiersprache*. Es gibt auch andere künstliche Sprachen, die keine Programmiersprachen sind, zum Beispiel die von dem polnischen Augenarzt Ludwik Lejzer Zamenhof entwickelte Verkehrssprache Esperanto, oder Klingonisch, die MutterSprache der aus Star Trek bekannten außerirdischen Rasse der Klingonen.

Wie menschliche Sprachen besitzen auch Programmiersprachen eine Grammatik, die sogenannte *Syntax*, die beschreibt, welche Formulierungen zulässig sind und welche nicht. Trotzdem unterscheiden sich Programmiersprachen deutlich von menschlichen Sprachen:

## 3.1 · Sprachen für Menschen, Sprachen für Maschinen

1. Programmiersprachen sind weniger komplex, sowohl in Bezug auf die Grammatik/Syntax als auch in Bezug auf die verwendeten Begriffe und deren Bedeutung (die sogenannte *Semantik*). Die gute Nachricht für Sie: Sie müssen nicht so viele Vokabeln lernen!
2. Die grammatischen Regeln müssen strikt eingehalten werden, sonst wird man nicht verstanden. Das ist die schlechte Nachricht. In normalen Gesprächssituationen versteht das menschliche Gegenüber einen Satz auch dann, wenn er grammatisch nicht ganz korrekt formuliert ist. Das ist wichtig, angesichts der Tatsache, dass nur Muttersprachler und Sprecher mit sehr langer Erfahrung in der betreffenden Fremdsprache die Sprache wirklich fehlerfrei beherrschen können. Diese Art der internen Fehlerkorrektur bzw. fehlertolerantem Verstehen setzt aber mentale Leistungen voraus, die wir gemeinhin mit Intelligenz assoziieren. Das Problem besteht nun – wie Sie sich schon denken können – darin, dass Computer einfach nicht intelligent sind. Sie haben keine Fehlertoleranz und akzeptieren und verstehen nur korrekt gebildete Sätze. Das ist es, was das Programmieren schwierig und von Zeit zu Zeit auch frustrierend macht, insbesondere dann, wenn einen der Computer partout nicht verstehen will (bzw. kann, denn einen freien Willen hat er natürlich auch nicht). Weil die Sprache so präzise sein muss, wirken ihre Formulierungen oft umständlich, was sie als Alltagssprache vollkommen unbrauchbar macht. Damit wären wir beim nächsten Punkt.
3. Programmiersprachen sind nicht dafür gemacht, gesprochen zu werden. Zwei Programmierer werden sich nicht in einer Programmiersprache unterhalten, wenn sie sich zum Mittagessen verabreden. Aber natürlich sprechen Programmierer manchmal seltsam, weil sie über Teile eines Programms, das in einer Programmiersprache verfasst ist, sprechen und dazu Teile des Programms zitieren.
4. Programmiersprachen im engeren Sinne (es gibt einige deskriptive Sprachen, die man als Ausnahmen betrachten könnte) sind nicht dafür gemacht, Informationen zu übertragen, sondern ausschließlich dazu gedacht, dem Computer Anweisungen zu erteilen.

Obwohl die Programmiersprachen als künstliche Sprachen also sowohl in Bezug auf ihre Grammatik als auch auf ihren Wortschatz erheblich einfacher angelegt sind als natürliche Sprachen, sind die Werke der Autoren, also der Programmierer, die sich dieser Sprachen bedienen, durchaus von erheblicher Komplexität und Umfang. Einige Beispiele:

- Die Programme, die die Apollo-11-Mondlandefähre steuerten und Neil Armstrong und Buzz Aldrin als erste Menschen sicher auf den Mond brachten, hatten ungefähr einen Umfang von 40 Tausend Zeilen; das sind 645 normal beschriebene DIN A4-Seiten.
- Das Betriebssystem Microsoft Windows 3.1 hatte ca. 2,5 Millionen Zeilen; etwa 40 Tausend DIN-A4-Seiten.
- Die Büroanwendungssuite Microsoft Office 2013 zieht ihre Funktionalität aus ungefähr 44 Millionen Zeilen Programmcode, die sich auf ca. 694 Tausend DIN-A4-Seiten drucken ließen.

- Das soziale Netzwerk Facebook basiert auf ca. 61 Millionen Zeilen Programmcode und damit ca. 984 Tausend Seiten.
- Alle Google-Anwendungen zusammen haben nach Auskunft einer Google-Entwicklerin ungefähr 2 Milliarden (!) Zeilen; ausgedruckt wären das ungefähr 32,3 Mio. DIN-A4-Seiten.

**3**

Zum Vergleich: Goethes *Faust* hat 12.111 Zeilen, das entspricht 195 normal bedruckten DIN-A4-Seiten.

Auch wenn Vergleiche zwischen diesen verschiedenen Programmen nicht ganz leicht sind, weil unterschiedliche Programmiersprachen für dieselbe Operation unterschiedlich viele Zeilen benötigen und auch der persönliche Programmierstil sowie die Art, wie der Programmcode formatiert ist (was vor allem Einfluss auf die Lesbarkeit und damit „Wartbarkeit“ hat) den Umfang mitbestimmt, so sieht man aber doch, dass Softwareprogramme oftmals sehr umfangreiche (und mitunter auch außerordentlich komplexe) Werke sind.

## 3.2 Übersetzung und Ausführung von Programmiersprachen

---

Im vorangegangenen Kapitel hatten wir festgestellt, dass die meisten Programmiersprachen englische Begriffe verwenden. Das liegt vor allem daran, dass die Entwicklung der Programmiersprachen vor allem im englischsprachigen Raum vorangetrieben worden ist, obwohl die erste richtige Programmiersprache die Erfindung des Deutschen *Konrad Zuse* war, auf den wir an späterer Stelle noch einmal zurückkommen. Natürlich könnte man genauso gut eine Programmiersprache entwerfen, deren Befehle allesamt einer anderen Sprache als dem Englischen, etwa dem Spanischen, Französischen oder Deutschen entlehnt sind. Es ist aber nahe liegend, dass das – außer möglicherweise zu pädagogischen Zwecken – normalerweise nicht geschieht, angesichts der dominanten Rolle des Englischen als Weltverkehrssprache und des nachvollziehbaren Wunsches jedes Schöpfers einer neuen Programmiersprache, sein Werk möge möglichst weite Verbreitung finden.

Nicht gelöst haben wir bisher jedoch das Problem, dass der Computer nun mal weder mit Englisch noch irgendeiner anderen menschlichen Sprache umgehen kann. Wenn der Computer aber nur Nullen und Einsen versteht, wie kommt er dann mit den an das Englische angelehnten Programmiersprachen klar? Was offensichtlich fehlt, ist noch irgendein Schritt zwischen dem Schreiben des Programms, also der Anweisungen, in der menschenlesbaren Programmiersprache und dessen Ausführung durch den Computer.

Diese Aufgabe übernimmt der sogenannte *Compiler*. Der Compiler („Übersetzer“) übersetzt das Programm in die Sprache, die der Computer tatsächlich direkt versteht, die *Maschinensprache*. Diese Maschinensprache kennt nur eine sehr begrenzte Menge an grundlegenden Befehlen, etwa zum Laden einzelner Werte in die Register des Prozessors, also die kleinen Speicherelemente, mit deren Inhalten der Prozessor arbeitet.

Es ist ein wenig so, als würden Sie einem *Ortskundigen* eine Route erklären müssen. Während der *Ortskundige* mit der Anweisung „Gehen Sie zum Odeon-

splatz, dort treffen wir uns dann“ von seinem jetzigen Standort aus ohne weiteres selbständig den Weg zum Treffpunkt finden würde, müssen Sie die Wegbeschreibung für den Ortsunkundigen auf elementarere Bestandteile herunterbrechen, zum Beispiel, indem Sie sagen „Gehen Sie geradeaus bis zur zweiten Ampel. Dann links. Dann gehen Sie gerade aus bis zur nächsten Kreuzung. Dort biegen Sie rechts ab und gehen geradeaus, bis Sie auf den Platz stoßen“. Genauso verhält es sich auch mit der Maschinensprache. Sie ist der auf elementare Prozessoroperationen heruntergebrochene Strom von Anweisungen, die sich aus dem in der Programmiersprache geschriebenen Programm ergeben. Das Programm mag viele „Abkürzungen“ und Verweise auf bereits bekannte Anweisungsfolgen beinhalten, aber für den Prozessor muss all das auf die grundlegenden Operationen reduziert werden, die er tatsächlich ausführen kann. Dazu bedarf es dann auch grundsätzlich keines weiteren Programms mehr. Der Maschinencode läuft direkt auf dem Prozessor des Computers. Wenn Sie mit Windows arbeiten, sind Ihnen wahrscheinlich die .exe- und .com-Dateien bekannt. Das sind die in Maschinencode übersetzten Programme, die der Computer direkt versteht und verarbeiten kann.

Etwas anders arbeitet ein *Interpreter*. Der Interpreter (der selbst wieder ein Programm ist) führt das Programm Schritt für Schritt aus, er „interpretiert“ den Programmcode. Es ist klar, dass der Interpreter letztlich natürlich wiederum das auszuführende Programm dem Computer in Maschinensprache übergeben muss, sonst könnte dieser es ja gar nicht verstehen. Nur findet diese Übersetzung im Fall des Interpreters eben statt, während das Programm läuft („zur Laufzeit“, wie man auch sagt). Das Programm läuft also nicht *direkt* auf dem Computer, sondern vermittelt durch den Interpreter. Das bedeutet aber auch: Man kann das Programm nicht einfach ohne weiteres überall ausführen, stets benötigt man ein zusätzliches Programm, eben den Interpreter, um den Quellcode des Programms auf einem Computer zum Laufen zu bringen.

Compilieren, also das Übersetzen des menschenlesbaren Programmcodes in die für den Computer verständliche Maschinensprach durch einen Compiler, hat normalerweise Geschwindigkeitsvorteile, weil das Übersetzen selbst Zeit in Anspruch nimmt und durch den Compiler im Vorhinein geleistet wird. Zur Laufzeit selbst muss dann keine Zeit mehr für die Übersetzung aufgewendet werden. Für unsere Zwecke sind Erwägungen der Ausführungsgeschwindigkeit des Codes nicht ganz so wichtig, aber wenn es um sehr rechenintensive oder zeitkritische Anwendungen geht (man denke etwa an Überwachungssysteme in der Intensivmedizin), sind solche Überlegungen durchaus von hoher Relevanz.

Es gibt noch einen Mittelweg zwischen Compiler und Interpreter. Der funktioniert so: Zunächst erzeugt ein sogenannter *Bytecode-Compiler* aus dem Quellcode des Programms einen „Zwischencode“, den sogenannten *Bytecode*. Der ähnelt stark der Maschinensprache ist aber unabhängig von der Prozessor-Architektur. Das Problem mit der Maschinensprache ist nämlich, dass sie davon abhängig ist, wie genau der Prozessor aufgebaut ist. Unterschiedliche Prozessortypen verstehen jeweils nämlich nur „ihre“ Maschinensprache. Der Bytecode ist dagegen maschinennunabhängig und kann so leicht auf einem (Typ von) System erzeugt und dann auf anderen (Typen von) Systemen ausgeführt werden. Man sagt auch, der Code sei „portabel“. Die Erzeugung des Bytecodes erfolgt im Rahmen der Entwicklung

des Programms. Zur Laufzeit dann erzeugt ein sogenannter *Just-in-time-Compiler* (JIT-Compiler) eben „just-in-time“ den maschinenspezifischen Code aus dem Bytecode. Der Just-in-time-Compiler ist gewissermaßen ein Bytecode-Interpreter. Er liest den „vorcompilierten“ Bytecode ein und gibt Anweisungen in der jeweiligen Maschinensprache des zur Ausführung verwendeten Systems an den Prozessor weiter. Das ist schneller als die direkte Ausführung des unübersetzten Quellcodes, wie sie ein klassischer Interpreter vornehmen würde. Prominentestes Beispiel für diesen Ansatz der Verwendung einer Kombination von Bytecode-Compiler (während der Entwicklung) und Just-in-time-Compiler (zur Laufzeit) ist die populäre Programmiersprache Java. Dank ihrer an das jeweilige System angepassten Just-in-time-Compiler (die gerne auch als „Laufzeitumgebungen“ bezeichnet werden, weil sie die Umgebung bilden, in der das Java-Programm ausgeführt wird), läuft sie nicht nur auf Computern mit ganz unterschiedlichen Betriebssystemen wie Windows, MacOS oder Linux, sondern auch auf verschiedenen anderen Endgeräten wie etwa Autos oder Kühlschränken.

Typischerweise wird es als eine Eigenschaft einer Programmiersprache betrachtet, ob sie compiliert oder interpretiert wird. Es gibt aber auch Sprachen, die grundsätzlich interpretiert werden, für die es dann aber trotzdem die Möglichkeit einer Bytecode-Compilierung gibt. Auch Interpreter compilieren unter Umständen Teile des Codes, die besonders rechenintensiv sind oder oft durchlaufen werden, und manchmal sogar den gesamten Code, direkt in ausführbaren Maschinencode; auch das wird als JIT-Compilierung bezeichnet. Die Welt ist also tatsächlich noch etwas bunter und der Compilerbau in der Informatik nicht ganz ohne Grund eine Kunst (und Disziplin) für sich.

### 3.3 Von der Maschinensprache bis zur Hochsprache

---

Alles schön und gut mit Compilieren und Interpretieren. Aber es stellt sich doch die Frage: Könnte man nicht ein Programm auch einfach direkt in Maschinensprache schreiben? Das ginge natürlich, „einfach“ wäre es aber keineswegs, im Gegen teil: Es wäre äußerst mühsam und das Ergebnis für den Entwickler und für andere, die das Programm später nachvollziehen wollen, sehr schwer zu verstehen. Zudem müsste man das Programm genau so schreiben, dass es auf die Eigenheiten des Prozessors, auf dem es laufen soll, Rücksicht nimmt. Soll es plötzlich auf einer ganz anderen Systemarchitektur laufen, müsste man es unter Umständen an vielen Stellen erheblich anpassen. Programmieren in Maschinensprache ist also nicht zu empfehlen. In der Anfangszeit der Computer war aber genau das nötig. Letztlich sind auch die Lockkarten-Programme von Charles Babbage und Kollegen Software, die in der Maschinensprache geschrieben worden sind, die Babbages *Analytical Engine* direkt verstanden hätte, wäre sie zu seinen Lebzeiten je gebaut worden.

Um sich die Mühen, direkt in Maschinensprache zu schreiben, zumindest zu vereinfachen, kamen ab den 1940er-Jahren sogenannte *Assembler*-Sprachen auf. Assemblersprachen sind letztlich Maschinencode in lesbarem Antlitz. Für die (überschaubar) zahlreichen Maschinenbefehle wurden kurze, an das Englische angelehnte Befehle (sog. *Mnemonics*) geschaffen, die sich eins zu eins in Maschinen-

### 3.3 · Von der Maschinensprache bis zur Hochsprache

code übersetzen lassen. Anders ausgedrückt: die Assemblerbefehle entsprechen exakt dem Befehlssatz des Prozessors. Auch in Assembler zu programmieren, ist noch immer ungemein mühsam. Ein Assemblerbefehl lautet beispielsweise:

```
MOV AL, 6Fh
```

Das sagt dem Computer: Lade den Wert 111 (6Fh bedeutet im hexadezimalen Zahlensystem, also im Sechzehner-basierten Zahlensystem, den Wert 111) in das Prozessorregister AL. Um einen einfachen, in einer Hochsprache verfassten Befehl wie etwa

```
print "Hallo Welt!"
```

in Assembler darzustellen, benötigt es eine ganze Reihe solcher Operationen. Vor allem aber ist man mit Assembler immer noch abhängig von der jeweiligen Prozessorarchitektur.

Einfacher ist es mit den sogenannten *Hochsprachen*. Hochsprachen abstrahieren von der Prozessorarchitektur, denn ihre Interpreter oder Compiler übernehmen die Berücksichtigung der Spezifika des Zielprozessors, sodass der Entwickler von dieser Aufgabe entlastet wird. Er schreibt sein Programm nur einmal; danach kann es in die Maschinensprache ganz verschiedener Prozessorarchitekturen übersetzt werden.

Die Hochsprachen sind, anders als Assembler-Code und (erst recht Maschinencode) einfacher lesbar, weil sie englische Begriffe für die Befehle verwenden. Viele dieser Befehle fassen eine ganze Reihe von Maschinencodebefehlen zusammen, das heißt, eine Zeile Hochsprachen-Code erzeugt unter Umständen bei der Compilierung viele verschiedene Maschinencode-Befehle. Der Programmierer braucht sich um diese Details freilich nicht mehr zu kümmern. Kein Wunder also, dass praktisch alle heute verwendeten Programmiersprachen solche Hochsprachen sind.

Die erste einfache Hochsprache wurde in den 1940er-Jahren von dem deutschen Ingenieur *Konrad Zuse* entwickelt und hieß *Plankalkül*. Zuse gelang mit *Plankalkül* eigentlich eine bahnbrechende Entwicklung, die der modernen Softwareentwicklung überhaupt erst den Weg hätte ebnen sollen. Bedingt durch den 2. Weltkrieg bliebt Zuses Arbeit aber lange unbeachtet. Die erste Implementierung (also nutzbare technische Umsetzung) von Zuses *Plankalkül* erfolgte erst 1975 durch Joachim Hohmann im Rahmen von dessen Dissertation.

Der Durchbruch der Hochsprachen gelang dagegen mit *ALGOL* (Kurzform für *Algorithmic Language*), das Ende der 1950er-Jahre/Anfang der 1960er-Jahre hauptsächlich für akademische Zwecke und weitestgehend ohne Bezug zu den Pionierarbeiten Konrad Zuses in den USA entwickelt wurde. Bereits zuvor, in den 1950er-Jahren, hatte die amerikanische Informatikern *Grace Hopper* den ersten funktionstüchtigen Compiler entworfen. *Hopper*, die 1992 starb und posthum mit der Presidential Medal of Freedom, der höchsten zivilen Auszeichnung der Verei-

nigten Staaten geehrt wurde, war außerdem an der Entwicklung der Programmiersprache COBOL (Kurzform für *Common Business-Oriented Language*) beteiligt, einer vor allem im geschäftlichen Kontext weitverbreiteten Sprache, die zusammen mit dem von *John Backus* 1957 entwickelten und hauptsächlich im technischen und wissenschaftlichen Bereich angewendeten *FORTAN* (Kurzform für *Formula Translation*) die Sammlung früher, wirklich einsatzfähiger Hochsprachen komplettiert.



# Warum gibt es so viele Program- miersprachen?

## Übersicht

Es gibt buchstäblich hunderte unterschiedlicher Programmiersprachen und ständig werden neue Sprachen entwickelt. Warum ist das so? Würde es nicht genügen, eine einzige Programmiersprache zu haben, die alle Programmierer verstehen und mit der alle Computer programmiert werden können?

In diesem Kapitel gehen wir der Frage nach, warum die Welt der Programmiersprachen so bunt, facettenreich, aber leider auch übersichtlich ist.

Die drei frühen Sprachen ALGOL, COBOL und FORTRAN bilden tatsächlich nur einen kleinen Teil des Universums heute existierender Programmiersprachen. Über deren genaue Zahl gibt es widersprüchliche Angaben. Das liegt auch daran, dass es eine Definitionsfrage ist, wann genau zwei Sprachen als „verschieden“ betrachtet werden. Soll etwa eine leichte Abwandlung einer Sprache (ein „Dialekt“) bereits als neue Programmiersprache gezählt werden? Und was ist mit den sogenannten deklarativen (beschreibenden) Programmiersprachen (wie etwa die beiden beliebten Sprachen HTML und CSS, mit denen u. a. Webseiten gestaltet werden), also Sprachen, die gar keine richtige Ablauflogik enthalten? Es ist also nicht ganz einfach, das, was man zählen will, überhaupt exakt greifbar zu machen.

Wikipedia jedenfalls listet zu dem Zeitpunkt, zu dem diese Zeilen geschrieben werden, 714 verschiedene Programmiersprachen auf. Das stellt jeden, der Programmieren lernen will, vor die Frage, welche Sprache er oder sie denn nun lernen sollte. Damit beschäftigen wir uns im folgenden Kapitel. Hier gehen wir erst mal der Frage nach, warum es überhaupt so viele Programmiersprachen gibt. Die Antwort auf diese Frage liegt vor allem in den unterschiedlichen *Anwendungsfeldern*, für die die Sprachen entworfen worden sind, den unterschiedlichen Arten, wie die Sprachen fundamental funktionieren (den sogenannten *Programmier-Paradigmen*) sowie der *Weiterentwicklung* bestehender Programmiersprachen in einem eigenen Zweig.

### ■ Unterschiedliche Anwendungsfelder

Programmiersprachen sind für unterschiedliche Zwecke entwickelt worden und eignen sich deshalb für „ihren“ Zweck besonders gut, für andere Zwecke möglicherweise weniger. Beispiele sind etwa die besonders im statistischen Bereich starke Sprache R, Apples Swift, das speziell für die App-Programmierung entwickelt wurde, die Structured Query Language (SQL), die der Standard für Datenbankabfragen ist oder PHP, dessen Hauptzweck darin besteht, dynamische Webseiten zu entwickeln, also Webseiten, die zum Beispiel auf Benutzereingaben reagieren.

Die Sprachen bringen für ihren jeweiligen „Hauptzweck“ spezielle Features mit, die es erlauben, Probleme in diesem Bereich besonders einfach zu lösen. Die Statistiksprache R kann zum Beispiel sehr gut mit Tabellen und Datenspalten (sog. Vektoren) umgehen, was in anderen Sprachen u. U. schwer zu bewerkstelligen ist. In der Datenbankabfragesprache SQL ist es sehr einfach, Datenbanken nach Informationen zu „befragen“ und beispielsweise „alle Kunden, die in den letzten 12

Monaten wenigstens zweimal unser neues Produkt über die Homepage gekauft haben“ aus dem riesigen Bestand aller Kundentransaktionsdaten zu ermitteln. Die Websprache PHP wiederum erlaubt es sehr einfach, zum Beispiel die Ergebnisse der Suche in einem Produktkatalog auszugeben oder die Login-Daten des Benutzers einer Social Media-Plattform zu verarbeiten und dem Benutzer sodann Zugang zu gewähren (oder eben auch nicht).

Nicht alle Programmiersprachen haben ein spezielles Anwendungsfeld, für das sie geschaffen worden sind. Es handelt sich dann nicht um sogenannte *Special-Purpose-Sprachen*, sondern um *General-Purpose-Sprachen*. Dazu gehören unter anderem die bekannteren Schwergewichte Java, C/C++, Python und VisualBasic. Trotzdem haben sich aufgrund der Eigenschaften dieser Sprachen dennoch oftmals Bereiche entwickelt, in denen sie besonders populär geworden sind: so ist C/C++ der Standard für systemnahe Programmierung (also etwa die Entwicklung von Betriebssystemen oder Gerätetreibern), Python für Data Science (also komplexe Auswertungen von Daten, inklusive Anwendung von Methoden künstlicher Intelligenz).

Der Übergang von den Special-Purpose- zu den General-Purpose-Sprachen ist aber einigermaßen fließend: Der weit verbreiteten Programmiersprache Java zum Beispiel liegt die Idee zugrunde, den gleichen Code einmal zu compilieren und dank Bytecode dann auf allen möglichen Geräten ausführen zu können. Das ist in gewissem Sinne auch ein spezieller Anwendungszweck. Trotzdem wird Java heute vielerorts eingesetzt, wo es auf die Portabilität der entwickelten Programme von einer System-Architektur auf eine andere gar nicht ankommt.

Die Zwecke und Anwendungsfelder, für die Programmiersprachen benötigt werden, ändern sich, und damit auch das Angebot an Sprachen, die für die aktuell populären Zwecke besonders günstig ist. In der ersten Hälfte der 90er-Jahre spielten Webanwendungen eine vollkommen untergeordnete Rolle. Wichtig waren stattdessen Sprachen für Programme, die man, zum Beispiel von Diskette oder CD-ROM, fest auf dem Computer installierte. Auf einfache Art (am besten per Drag & Drop) attraktiv aussehende und gut zu bedienende Programm-Oberflächen zu gestalten und dann ereignisgesteuert auf das Verhalten des Benutzers auf diesen Oberflächen reagieren zu können, waren zentrale Anforderungen; dazu vielleicht noch, ohne große Mühe ein komfortables Installationsprogramm bereitzustellen zu können. Damit ließen sich Entwickler zu dieser Zeit glücklich machen. Dann wurden Internetanwendungen wichtiger und mit ihnen ganz neue Sprachen, die in Webbrowsern laufen mussten, die in der Lage sein mussten, HTML-Seiten zu generieren oder zu verändern und mit Webservern zu kommunizieren, um beispielsweise Daten aus Datenbanken auf dem Server auszulesen und hübsch aufbereitet für den Benutzer anzuzeigen. Damit verloren Sprachen für fest installierte Anwendungen relativ zu den neuen „Emporkömmlingen“ an Bedeutung. Das gilt in gewissem Maße auch für Java, dessen plattformübergreifender Ansatz nicht mehr so entscheidend war, jetzt, da die neuen Sprachen ohnehin direkt vom Browser interpretiert wurden oder auf dem Server liefen und einfach eine vollständige, dynamisch erzeugte Website an den Browser zurückgaben.

## ■ Unterschiedliche Paradigmen

Programmiersprachen liegen unterschiedliche Programmier-*Paradigmen* zugrunde. Ein Paradigma in diesem Sinne ist das grundsätzliche Prinzip, wie Programme in der jeweiligen Sprache aufgebaut sind und funktionieren. Die Paradigmen, die in der Programmierung Anwendung finden, sind nicht überschneidungsfrei, auch wenn sich manche Kombinationen ausschließen; tatsächlich folgen die meisten Sprachen mehreren Paradigmen.

4

Ein besonders wichtiges (weil von vielen Programmiersprachen befolgtes) Paradigma ist die objektorientierte Programmierung (OOP). Vereinfacht gesagt erlaubt es OOP, auf einfache Weise Gegenstände, wie sie in der realen Welt auftreten, mit ihren Eigenschaften in Programmen nachzubilden und damit zu arbeiten, zum Beispiel neue Objekte dieser Art zu erzeugen oder ihre Eigenschaften zu ändern. Denken Sie etwa an ein Auto: Ein Auto ist ein Objekt mit verschiedenen Eigenschaften wie Farbe, Marke, Beschleunigung und so weiter. In einer objektorientierten Sprache könnten Sie sehr einfach ein Datenkonstrukt, das ein Auto repräsentiert, erzeugen und dabei seine Eigenschaften setzen, zum Beispiel auf Farbe = nachtblau, Marke = BMW, Beschleunigung = 9,1 s von 0 auf 100 km/h. Auch könnte es Eigenschaften geben, die den aktuellen Fahrstatus des Autos widerspiegeln, zum Beispiel seine aktuelle Geschwindigkeit, wie viele Personen sich im Fahrzeug befinden und ob der Blinker gerade an ist. Für viele Zwecke ist eine solche Repräsentation von Objekten sehr nützlich. Wir werden uns an späterer Stelle noch ausführlicher mit objektorientierter Programmierung beschäftigen.

Ein weiteres Beispiel für ein (vollkommen anderes) Programmier-Paradigma ist die sogenannte *logische Programmierung*, die etwa die Sprache *Prolog* („Programmation en logique“) verfolgt: Im logischen Programmier-Paradigma definiert man Regeln, die das Wissen über ein Teilgebiet der Welt beinhalten. So könnte man zum Beispiel als Regel definieren, dass zwei Kinder genau dann Geschwister sind, wenn sie dieselben Eltern haben. Dann könnte man dem Prolog-Interpreter mitteilen, dass Pauls Vater Markus, und seine Mutter Julia ist; und weiterhin, dass Petras Vater Markus und Petras Mutter Julia ist. Gefüttert mit diesem Wissen und der zuvor definierten Regel über das Verhältnis von Eltern und Kindern, würde der Prolog-Interpreter jetzt vollkommen selbstständig und ohne, dass irgendwie weitere Programmierung nötig wäre, den logischen Schluss ziehen könne, das Paul und Petra Geschwister sind.

Wie so vieles in der Welt sind auch Programmier-Paradigmen Moden und Trends unterworfen. Kommt ein Paradigma in Mode entstehen mehr Sprachen, die diesem Paradigma folgen.

## ■ Weiterentwicklungen und Dialekte

Ein Teil der Vielfalt an Programmiersprachen erklärt sich auch daraus, dass „Dialekte“ von Sprachen entstanden sind, also neue Sprachen, die eng an eine andere Sprache angelehnt sind, aber Weiterentwicklungen bzw. – zumindest aus Sicht des Entwicklers – Verbesserungen gegenüber der Originalsprache darstellen. Beispiele hierfür sind C++ und C# (gesprochen „C sharp“), die beide Fortentwicklungen

## Warum gibt es so viele Programmiersprachen?

der Programmiersprache C sind (und die Sprache auf das objektorientierte Programmier-Paradigma umgestellt haben).

Aber warum passt man nicht einfach die Originalsprache an, statt mit dem Dialekt einen neuen „Zweig“ aufzumachen? Gründe hierfür können sein, dass der Entwickler der Originalsprache die angedachten Änderungen nicht mitträgt; oder man will sicherstellen, dass Code, der in der Originalsprache geschrieben ist, nach wie vor lauffähig ist, selbst, wenn man mit der neuen Sprache gravierende Änderungen einführt, die dazu führen würden, dass alle bisherigen Programme in der Originalsprache nicht mehr fehlerfrei ausgeführt werden können.

Praktisch alle Programmiersprachen sind (selbst wenn sie keine Weiterentwicklungen anderer Sprachen darstellen) von anderen Sprachen beeinflusst. Das sieht man recht schön an den Wikipedia-Artikeln der Sprachen, in denen sehr systematisch dargestellt ist, von welchen anderen Sprache die betreffende Sprache beeinflusst wurde, und welche anderen Sprachen sie selbst beeinflusst hat.

Ist eine Sprache ein Dialekt einer anderen Sprache, so hat sie oftmals einen Namen, der an den Namen der Originalsprache angelehnt ist, wie wir es eben bei C++ und C# gesehen haben. Andere Beispiele für dieses Phänomen sind VisualBasic (eine Weiterentwicklung von BASIC) und Object Pascal (eine objektorientierte Variante von Pascal). Letzteres ist ein gutes Beispiel dafür, dass eine Umstellung der Originalsprache auf das objektorientierte Programmier-Paradigma oft zu Erweiterung des Namens um „Object“ oder „++“ (angelehnt an das „Originalpärchen“, C/C++) führt.

Ansonsten ist die Namenslandschaft der Programmiersprachen sehr bunt abwechslungsreich. Manche Programmiersprachen haben Namen, die an Eigennamen von Personen angelehnt sind (den Namen des Entwicklers oder von Personen, die mit der Benennung geehrt werden), zum Beispiel Ada (nach der Programmierungspionierin Augusta Ada Byron), Eiffel (nach französischen Ingenieur und Erbauer des Eiffel-Turms), Gödel (nach dem österreichischen Mathematiker) oder Wolfram Language (nach dem britischen Mathematiker und Informatiker Stephen Wolfram). Die Namen weiterer Sprachen dagegen sind Abkürzungen, wie etwa in den Fällen von SQL („Structured Query Language“), VBA („Visual Basic for Applications“), Fortran („Formula Translation“), COBOL (Common Business Oriented Language) oder Prolog („programmation en logique“), andere wiederum reine Phantasienamen wie etwa Delphi oder Python.

Viele (insbesondere professionelle) Programmierer sind heute einigermaßen sprach-agnostisch: Sie verwenden einfach die Sprache, die sich für ihr aktuelles Vorhaben am besten eignet. Dabei schauen sie sich das zu lösende Problem an und entscheiden dann, womit sie arbeiten wollen. Sie können zwar nicht alle Sprachen, aber doch eine ganze Reihe, aus denen sie dann wählen können. Und diese Sprachpalette ist in der Regel sehr viel breiter als die von professionellen Übersetzern. Das funktioniert, weil die Programmierer die Grundkonzepte der Programmierung verinnerlicht haben und sich deshalb neue Sprache leicht aneignen können. Das wollen wir mit diesem Buch auch lernen. Zuvor aber schauen wir uns im nächsten Kapitel an, wie man denn nun eine geeignete Sprache auswählt.



# Welche Programmiersprachen sollte man lernen?

## Übersicht

Im letzten Kapitel haben wir gesehen, warum es so viele unterschiedliche Programmiersprachen gibt. Diese Vielfalt stellt uns aber vor das Problem, zu entscheiden, welche Sprache(n) wir lernen sollten. In den Teilen 3 und 4 dieses Buches haben Sie die Möglichkeit, sich mit zwei äußerst populären Programmiersprachen vertraut zu machen, Python und JavaScript; insofern hat der Autor bereits eine Vorauswahl für Sie getroffen. Unabhängig davon wollen wir uns aber in diesem Kapitel mit der Frage beschäftigen, nach welchen Kriterien Sie Sprachen, die für Sie von Interesse sein könnten, auswählen können. Denn wenn Sie erst mit dem Programmieren anfangen, wird es langfristig mit Sicherheit nicht bei der ersten Sprache, mit der Sie in die Welt der Programmierung eingestiegen sind, bleiben.

Angesichts des immensen Angebots von Programmiersprachen stellt sich natürlich die Frage, mit welchen davon man sich beschäftigen sollte. Wie Sie sich wahrscheinlich schon gedacht oder gar befürchtet haben, fällt die Antwort salomonisch aus: Kommt darauf an.

### ■ Gegenstand Ihres Vorhabens

Wir hatten im vorangegangenen Kapitel gesehen, dass Sprachen für unterschiedliche Anwendungsgebiete unterschiedlich gut geeignet sind. Also macht es Sinn, für Ihr(e) Vorhaben eine Sprache zu wählen, mit der sich das Ziel komfortabel erreichen lässt. Wenn Sie vorwiegend Websites mit dynamischen Elementen bauen wollen (zum Beispiel, wenn Ihre Website die Adressdaten beim Anlegen eines neuen Benutzerkontos durch einen Kunden validieren soll), dann werden Sie wohl am ehesten auf JavaScript zurückgreifen, weil JavaScript im Webbrowser läuft und es sich hervorragend dazu eignet, mit den Eingaben des Benutzer zu arbeiten, bevor diese an den Webserver geschickt werden. Im nächsten Schritt werden Sie aber nicht umhinkommen, auch den Webserver selbst in die Arbeit einzuspannen, schließlich sollen ja die Kundendaten auch in eine Datenbank geschrieben werden. Dazu läuft auf dem Webserver ein Skript, für das vermutlich PHP die Sprache Ihrer Wahl sein wird. PHP ist eben genau zu diesem Zweck entwickelt worden. Es läuft auf dem Webserver und macht es leicht, im Hintergrund mit Datenbanken zu arbeiten und die Ergebnisse von Datenbankabfrage dynamisch als Website im Browser des Benutzers anzuzeigen. Wenn Sie dagegen eine echte Windows-Anwendung entwickeln wollen, werden weder PHP noch JavaScript in der engeren Auswahl sein, die allesamt dafür gemacht sind, im Web-Umfeld ausgeführt zu werden. Für eine echte Windows-Anwendung werden Sie wohl eher auf Visual C/C++, VisualBasic, Delphi oder etwas Ähnliches zurückgreifen. Diese Sprachen erlauben es, klassische Stand-Alone-Anwendungen zu entwickeln, dazu hübsche Oberflächen zu gestalten und auf die Aktionen des Benutzers zu reagieren. Wollen Sie Programme für die Auswertung von großen Datenmengen entwickeln, werden Sie wohl auf R oder Python zurückgreifen. Und für Apps werden Sie vielleicht Objective C oder das verhältnismäßige neue, von Apple entwickelte Swift einsetzen.

Wir haben immer davon gesprochen, eine Sprache sei für einen bestimmten Zweck besonders gut *geeignet*. Aber was macht Sprachen besser oder weniger geeignet? Es sind vor allem zwei Dinge:

1. Die Umgebung, in der die Sprache läuft, also etwa als Stand-Alone-Anwendung auf einem bestimmten Betriebssystem, im Rahmen einer Website im Webbrowsert des Benutzers, auf dem Webserver, oder als Handy-/Tablet-App.
2. Die Werkzeuge, die die Sprache bietet, um die Aufgabe, die Sie sich vorgenommen haben, zu bewältigen. Für viele Sprache gibt es sogenannten Bibliotheken und Frameworks, das sind letztlich Erweiterungen, die bestimmte der Sprach bestimmte Funktionalitäten verleihen, so könnte es zum Beispiel eine Bibliothek für das Webscrapen geben, also für das Durchsuchen anderer Webseiten nach Inhalten und deren Extraktion. Wenn Sie nun ein Programm schreiben wollen, dass zum Beispiel den Preis eines bestimmten Amazon-Artikels im Auge behält und Sie informiert, wenn sich dieser ändert, dann werden Sie gut beraten sein, eine Sprache zu wählen, die mit einer starken Webscraping-Bibliothek daherkommt. In Bezug auf Bibliotheken und Frameworks sind Open-Source-Sprachen mit einer aktiven, engagierten Community regelmäßig im Vorteil gegenüber ihren kommerziellen Konkurrenten.

Aber auch von Haus aus bringen die Sprachen oft wichtige Features mit, die die Sprache für Ihren Zweck mehr oder weniger gut geeignet erscheinen lassen. Das kann beispielsweise mit den Datenstrukturen der Sprache zusammenhängen, also mit der Art und Weise, wie Daten in der Sprache abgelegt und bearbeitet werden. Wollen Sie etwa mit Datentabellen und Spalten aus diesen Tabellen arbeiten, ist die hauptsächlich für statistische Zwecke eingesetzte Sprache R eine gute Wahl, biete Sie doch out-of-the-box mit den Dataframes und Vektoren direkt umfangreiche Unterstützung für diese Datenstrukturen.

## ■ Kundenwunsch und Arbeitgeberpräferenz

Wenn Sie auf Auftrag arbeiten, etwa als Freelancer, hat der Kunde vielleicht Vorstellungen, welche Programmiersprache verwendet werden soll. Das mag daher kommen, dass er Ihr Programm in seine größere Gesamtsoftware, die in eben jener Sprache entwickelt ist, reibungslos integrieren will. Vielleicht kennen sich er und seine internen Entwickler aber auch einfach am besten mit der betreffenden Sprache aus. Versteht ihr Kunde nicht, wie das, was Sie da entwickelt haben, funktioniert, kann er es auch schlecht weiterentwickeln.

Auch wenn Sie eine Anstellung als Entwickler suchen, ist es wichtig, eine oder einige derjenigen Sprachen zu beherrschen, die am Markt nachgefragt sind. Nur wie findet man die?

Eine Möglichkeit besteht darin, sich am sogenannten *TIOBE-Index* zu orientieren. Diese Aufstellung des gleichnamigen Softwareunternehmens bewertet die Relevanz von Programmiersprachen anhand der Anzahl der Treffer, die man in verschiedenen Suchmaschinen erzielt, wenn man nach dem Namen der Sprache sucht.

Auch die Anzahl der Frage- und Antwortbeiträge in dem bei Programmierern äußerst populären Forum *Stack Overflow* ist ein guter Indikator. Quantitative Auswertungen über die Programmiersprachen, auf die sich die dort geposteten Fragen

beziehen, veröffentlichen die Stack-Overflow-Betreiber immer wieder. Zudem bieten die regelmäßigen Nutzerbefragungen von Stack Overflow einen Anhaltspunkt. Hier wird unter anderem regelmäßig nach den verwendeten Programmiersprachen gefragt.

Was aber, wenn Sie all diesen Statistiken nicht trauen? Immerhin gibt es durchaus Gründe, die Aussagekraft solcher Zahlen zumindest kritisch zu hinterfragen: Könnte es nicht etwa sein, dass die vielen Fragen zu einer Programmiersprache auf *Stack Overflow* nur daher kommen, dass die Sprache sehr kompliziert und deshalb möglicherweise einigermaßen unbrauchbar ist? Mit dem gleichen Argument lässt sich die Aussagekraft der Suchtreffer-basierten *TIOBE*-Popularitätsstatistiken in Frage stellen. Weiterhin könnte man bezweifeln, dass die Nutzerbefragungen von Stack Overflow repräsentativ ist – vielleicht neigen bestimmte Gruppen, wie etwa Nerds, die sich anderen Gründen als ihrer praktischen Anwendbarkeit mit einer Sprache beschäftigen, eher dazu, an solchen Befragungen teilzunehmen.

Das alles sind legitime Überlegungen. Immerhin jedoch zeigen hohe Anzahlen von Fragen und Suchtreffern wenigstens, dass es einige Leute gibt, die sich mit der Sprache beschäftigen. Möchte man aber tatsächlich auf Statistik verzichten, bleibt nur noch, anekdotische Hinweise für die Bedeutung bestimmter Programmiersprachen zu suchen. Gehen Sie auf die großen Job-Portale, suchen Sie nach Jobs für Programmierer/Entwickler und schauen Sie, welche Sprachen in den Jobanzeigen gefordert werden (wenn Sie sich auch hierzu lieber auf Statistik verlassen: Untersuchungen wie etwa die des amerikanischen Programmierkurs-Anbieter *Coding Dojo* untersuchen Stellenanzeigen auf systematische Weise, im Falle von *Coding Dojo* die auf der Job-Suchmaschine *indeed.com*). Wenn Ihnen das als anekdotische Evidenz immer noch nicht ausreicht und Sie doch lieber die persönliche Meinung von (zumeist selbst ernannten) Experten hören wollen, gehen Sie auf *YouTube* und suchen Sie nach „Programming languages to learn in [hier Jahreszahl einsetzen]“ und sie werden ganz schnell dutzende YouTuber mit mehr oder weniger guten Einblicken in die Marktlage und Anwendbarkeit von Programmiersprachen finden, die ihren Zuschauern bereitwillig und ausführlich erklären, welche Sprachen sie aus welchen Gründen präferieren.

### ■ Pädagogische Gesichtspunkte

Insbesondere zu Beginn könnten Sie bewusst Sprachen auswählen, die hinsichtlich des grundlegenden Ansatzes, wie man mit ihnen programmiert (also ihrer *Programmier-Paradigmen*, mit denen wir uns im dritten Teil des Buchs noch näher beschäftigen werden) unterschiedliche Wege gehen, um ein breites Spektrum von Möglichkeiten kennenzulernen. Das klingt etwas einfacher als es in der Realität ist, denn die meisten Sprachen verfolgen keineswegs nur ein einziges Paradigma, sondern sind *multiparadigmatisch*, picken sich also Elemente aus verschiedenen Paradigmen heraus. Ein guter Mix schafft aber trotzdem ein fundiertes Verständnis der unterschiedlichen Herangehensweisen.

Genauso könnten Sie zunächst mit Sprachen beginnen, die verhältnismäßig leicht zu erlernen sind. Schnell erste Lernerfolge zu verzeichnen ist immerhin eine zentrale Motivation dafür, dranzubleiben und weiterzumachen.

Diesen pädagogischen Gesichtspunkten folgen wir auch im vorliegenden Buch.

Ihnen ist wahrscheinlich aufgefallen, dass wir hier immer davon ausgehen, dass Sie letztlich nicht nur eine, sondern mehrere Programmiersprachen lernen. Aber ist es realistisch, wirklich mehrere Sprachen zu beherrschen, und das auch ausreichend gut?

Wie ist Ihr Englisch? Wie Ihr Französisch? Wie steht es um Ihre Spanischkenntnisse? Können Sie sich in allen drei Sprachen einem Muttersprachler verständlich machen? Wahrscheinlich nicht (es sei denn, Sie sind wirklich sehr sprachbegabt). Es ist offenbar recht schwer, natürliche Sprachen gut zu beherrschen. Bei Programmiersprachen sieht das etwas anders aus. Wenn Sie die Grundkonzepte der Programmierung verstanden haben, können Sie mit überschaubarem Aufwand weitere Sprachen erlernen.

Wie bei den natürlichen Sprachen auch, wird Ihnen natürlich ein „Sprecher“ mit langjähriger (insbesondere professioneller) praktischer Erfahrung, ein „Muttersprachler“ sozusagen, bei weitem überlegen sein und Tricks und Kniffe kennen, von denen Sie noch nie gehört haben. Auf dieser Weise wird er eine gegebene Aufgabe mit einem Programm lösen können, das kürzer ist und schneller läuft als eines, das Sie schreiben können. Dennoch: Sich die Grundlagen einer neuen Sprache anzueignen, auf einem Niveau, das Sie in der Lage versetzt, erfolgreich Programme zu schreiben, ist nicht schwer.

Diese Aussage mag Ihnen zwar an dieser Stelle als eine einigermaßen kühne Behauptung erscheinen, wir werden aber in diesem Buch den Beweis dafür antreten.

Nachdem wir uns im dritten Teil des Buchs ausgiebig mit den Grundkonzepten der Programmierung befasst haben, werden Sie im vierten und fünften Teil *zwei* Programmiersprachen lernen, *Python* und *JavaScript*.

Diese beiden Sprachen sind nicht nur sehr nützlich, sondern nach allen Maßstäben auch äußerst populär. Im *TIOBE-Index* vom September 2018 belegen sie die Plätze 4 (Python) und 8 (JavaScript). Die führenden Sprachen im Index – Java, C und C++ – sind etwas komplizierter und für den Einstieg nicht ganz so gut geeignet wie Python und JavaScript. Letztere beiden sind auf Stack Overflow zu dem Zeitpunkt, zu dem diese Zeilen geschrieben werden, die beiden am stärksten nachgefragten Sprachen, das heißt, die meisten Befragten, die diese Sprachen noch nicht sprechen, wollen sie lernen. Im Juni 2017 waren es zudem die Sprachen mit dem höchsten Anteil an Fragen auf Stack Overflow, wobei insbesondere der Anteil der Python-bezogenen Fragen in den letzten Jahren erheblich gestiegen ist.



# Einige Tipps

## Übersicht

Bevor wir im nächsten Teil des Buchs in die Grundkonzepte der Programmierung einsteigen, finden Sie in diesem Kapitel noch einige aufmunternde Tipps für den Einstieg in die faszinierende Welt der Programmierung.

6

### ■ Lernen Sie erst mal das Wesentliche!

Fangen Sie beim Lernen einer neuen Sprache klein an. Mit ganz wenigen Grundelementen kann man in der Regel bereits lauffähige Programme entwickeln. Man muss dazu nicht jedes Detail kennen. Sie werden die Sprache ohnehin nie „komplett“ können, es gibt immer noch mehr Features, noch mehr Bibliotheken etc., die sie noch nicht kennen. Wichtig ist, sich ein Grundverständnis der Sprache zu erarbeiten und das ist in der Regel viel einfacher, als es zunächst erscheint. Fokussieren Sie Ihre Bemühungen deshalb erst einmal auf die zentralen Konzepte der Sprache. Dabei helfen die Grundfragen, die wir in Teil 2 dieses Buches behandeln. Versuchen sich nicht, tonnenweise Literatur zu lesen, um jeden Winkel der Sprache theoretisch verstanden zu haben, bevor Sie sich an Ihr erstes Programm trauen. Ganz im Gegenteil! Fangen Sie möglichst früh an, eigene kleine Programme zu schreiben. Ihr Wissen wird sich mit der Zeit ganz natürlich erweitern.

### ■ Spielen Sie!

Trauen Sie sich, Dinge auszuprobieren. Anders als in der Fahrschule kann beim Programmieren nichts kaputt gehen, wenn Sie ein wenig herumspielen. Wenn Sie Dinge ausprobieren, lernen Sie schnell, was funktioniert und was nicht. Bekanntlich lernt man aus Fehlern besonders gut. Durch Ausprobieren erfahren Sie Dinge, die in keinem Buch genau so beschrieben sind.

### ■ Lassen Sie sich nicht entmutigen!

Lassen Sie nicht entmutigen, wenn Ihr Programm am Anfang nicht das tut, was es tun soll, oder wenn Sie ständig eine Fehlermeldung erhalten und nicht die geringste Ahnung haben, woran das liegen könnte. Wir versuchen natürlich, so etwas mit der Heranführung in diesem Buch zu vermeiden. Aber die Wahrheit ist auch: Die besten Programmierer machen ständig Fehler. Haben Sie sich mal gefragt, warum sich die Apps auf Ihrem Handy so oft aktualisieren? Viele dieser Updates sind reine *Bugfixes*, also Fehlerkorrekturen. Fehler machen, finden und ausbügeln ist Teil des Programmierhandwerks, und bei weitem nicht der unwichtigste. Das ist manchmal mühsam und nervenraubend, aber kein Grund, die Flinte ins Korn zu werfen. Übrigens: Eine kleine Pause wirkt manchmal Wunder.

### ■ Fangen Sie klein an, und lassen Sie Ihr Programm schrittweise wachsen!

Wenn Sie ein Programm schreiben, überlegen Sie sich, was die eigentliche Aufgabe ist, die das Programm bewältigen soll, und welche Features Ihr Programm dazu wirklich benötigt. Entwickeln Sie diese zuerst. Wenn die Grundfunktionalität

steht, können Sie Schritt für Schritt mehr Komplexität hinzufügen, etwa um die Benutzerfreundlichkeit zu erhöhen oder das Programm robuster gegenüber Fehleingaben des Anwenders zu machen.

#### ■ **Lassen Sie die Schönheit Schönheit sein!**

Es ist wichtig, dass Ihr Programm funktioniert, also das tut, was es soll, und dass es robust ist, das heißt, nicht so leicht aus der Fassung zu bringen ist, wenn etwa der Benutzer einer Fehleingabe macht. Weit weniger wichtig ist erst mal, dass das Programm ein Musterbeispiel von Eleganz und Effizienz ist. Man hört manchmal Programmierer davon sprechen, dass ein Stück Programmcode „elegant“ oder „schön“ ist. Solche Genießer-Aussagen sind Ausdruck der häufig geäußerten Auffassung, Programmieren sei Wissenschaft und (Handwerks-)Kunst zugleich. Übertreiben Sie es aber nicht mit der Kunst. Ein Profi mit Jahren von Erfahrung wird Ihre Programme vielleicht etwas ungelassen finden und einige Optimierungsmöglichkeiten sehen. Das ist aber nicht weiter schlimm. Besser, als dass sie ewig an einem eigentlich bereits funktionsfähigen Programm herumtüfteln, um es noch eleganter zu machen, ist allemal, etwas Neues auszuprobieren. Auf diese Weise lernen Sie erheblich mehr dazu!

#### ■ **Dokumentieren Sie!**

Gewöhnen Sie sich gleich zu Beginn Ihres Programmierdaseins an, Ihren Programmcode zu dokumentieren, insbesondere, indem Sie ihn mit Kommentaren versehen, die erläutern, wie der Code funktioniert. Das ist wichtig, damit Sie Ihr Programm später noch verstehen. Kommentieren ist wahrscheinlich die meist gehasste aber zugleich eine der wertvollsten Aktivitäten beim Programmieren. Mit dem Kommentieren werden wir uns deshalb in ► Kap. 10 noch genauer befassen.

# Die Grundkonzepte des Programmierens

## Inhaltsverzeichnis

Kapitel 7	Neun Fragen – 51
Kapitel 8	Was brauche ich zum Programmieren? – 55
Kapitel 9	Was muss ich tun, um ein Programm zum Laufen zu bringen? – 69
Kapitel 10	Wie stelle ich sicher, dass ich (und andere) mein Programm später noch verstehe? – 75
Kapitel 11	Wie speichere ich Daten, um mit ihnen zu arbeiten? – 87
Kapitel 12	Wie lasse ich Daten ein- und ausgeben? – 123

- Kapitel 13** Wie arbeite ich mit Programmfunctionen, um Daten zu bearbeiten und Aktionen auszulösen? – 151
- Kapitel 14** Wie steuere ich den Programmablauf und lasse das Programm auf Benutzeraktionen und andere Ereignisse reagieren? – 169
- Kapitel 15** Wie wiederhole ich Programmanweisungen effizient? – 195
- Kapitel 16** Wie suche und behebe ich Fehler auf strukturierte Art und Weise? – 213



# Neun Fragen

## Übersicht

In diesem Kapitel verschaffen wir uns einen Überblick über die 9 Fragen, anhand derer wir in diesem Teil des Buches die Grundkonzepte des Programmierens kennenlernen.

Auf Basis dieser Grundkonzepte werden wir uns dann in den folgenden beiden Teilen des Buches die Programmiersprachen Python und JavaScript erarbeiten. Und auch darüber hinaus sind diese Grundkonzepte – zusammengefasst in Form der 9 Fragen – ein äußerst nützliches Denkschema, mit dem Sie sich jede beliebige Programmiersprache erschließen können, die Sie erlernen wollen.

7

Dieser zweite Teil des Buches widmet sich den *Grundkonzepten* des Programmierens. Die Grundkonzepte sind in praktisch allen Programmiersprachen auf die eine oder andere Weise umgesetzt. Wenn Sie diese Grundkonzepte verstanden haben, werden Sie viele Ähnlichkeiten zwischen unterschiedlichen Programmiersprachen entdecken. Diese Ähnlichkeiten sind es, die das Erlernen neuer Programmiersprachen erheblich vereinfachen.

Wir werden in diesem Teil die Grundkonzepte des Programmierens unter 9 großen *Fragen* zusammenfassen. Wenn Sie eine neue Programmiersprache lernen, können Sie sich in Ihrem Lernprozess an diesen Fragen orientieren. Natürlich können Sie dabei in einer anderen Reihenfolge als jener vorgehen, die durch die 9 Fragen vorgegeben ist. Und tatsächlich müssen Sie an manchen Stellen, wenn Sie sich mit einer Frage beschäftigen, teilweise einer anderen Frage gewissermaßen „vorgreifen“; wenn Sie sich etwa mit der Abbildung von Daten in Ihrer Sprache auseinandersetzen, müssen Sie – zumindest sehr rudimentär – auch in der Lage sein, Daten auszugeben (was Gegenstand einer anderen Frage ist), um überhaupt praktisch etwas ausprobieren zu können. Auch, wenn die Fragen also nicht ganz in sich abgeschlossen sind (und sein können), so bilden sie doch ein nützliches Denkraster für Ihren Fahrplan, die neue Sprache von Grund auf zu verstehen.

Selbst aber, wenn Sie sich nicht an den Ablauf der 9 Fragen halten: Wollen Sie eine für Sie neue Programmiersprache in den Grundzügen verstanden haben, müssen Sie am Ende diese 9 Fragen beantworten können.

Wir werden nun in diesem Teil zunächst die Grundkonzepte des Programmierens anhand der 9 Fragen kennenlernen. Am Ende jedes Kapitels finden Sie einen Abschnitt *Ihr Fahrplan zum Erlernen einer neuen Programmiersprache*, der Ihnen die wichtigsten Punkte des jeweiligen Kapitels, mit denen Sie sich vertraut machen sollten, wenn Sie eine neue Programmiersprache lernen, zusammenfasst.

In den nächsten beiden Teilen werden wir die Grundkonzepte dann anwenden, um die Grundzüge von Python und JavaScript zu erlernen. Diese beiden Teile des Buches sind folgerichtig auch anhand der 9 Fragen strukturiert. Sie können daher jederzeit, während Sie sich mit Python und JavaScript beschäftigen, auf das zugehörige Grundkonzepte-Kapitel zurückblättern und sich nochmal die ein oder andere Grundüberlegung in Erinnerung rufen.

Die 9 Fragen, die es zu beantworten gilt, sind die folgenden.

### ■ Was brauche ich zum Programmieren?

Hier geht es zunächst um die *Werkzeuge*, die Tools, die Sie benötigen, um Programme in der Programmiersprache zu schreiben und auszuführen. Außerdem beschäftigen wir uns damit, wo Sie sie weitere Informationen und Hilfe erhalten können, wenn Sie einmal nicht mehr weiterkommen.

### ■ Was muss ich tun, um ein Programm zum Laufen zu bringen?

Als nächstes entwickeln wir ein allererstes, sehr einfaches Programm. Dabei lernen wir bereits gewisse *Grundregeln des Programmaufbaus* und der *Grammatik* der Programmiersprache kennen und verstehen, wie Programme *geschrieben* und *ausgeführt* werden. Dies zu können, ist eine Grundvoraussetzung für alle weiteren Lernschritte, in deren Zuge wir ja auch ganz praktisch lauffähige Programme schreiben wollen.

### ■ Wie stelle ich sicher, dass ich (und andere) mein Programm später noch verstehe?

Ihre Programme müssen mindestens für Sie, manchmal aber auch für andere, verständlich sein. Dabei hilft es, bestimmte *Konventionen*, wie Programmcode aussehen sollte, einzuhalten und Ihr Programm zu *kommentieren*, das heißt, mit Erläuterungen zu versehen. Wenn Ihr Programmcode auch von anderen verwendet werden soll, müssen Sie zudem *dokumentieren*, wie genau das geschehen kann.

### ■ Wie speichere ich Daten, um mit ihnen zu arbeiten?

Alle Software arbeitet mit Daten. Deshalb beschäftigen wir uns als nächstes damit, wie Daten in Programmen vorgehalten und bearbeitet werden. Dabei lernen wir das zentrale Konzept der *Variablen* kennen und sehen, wie man mit *Objekten* Gegenstände und Sachverhalte elegant abbilden kann.

### ■ Wie lasse ich Daten ein- und ausgeben?

Die Daten, die im Programm zur Verarbeitung vorgehalten werden, müssen aber auch irgendwo herkommen. Außerdem sollen die Verarbeitungsergebnisse wieder irgendwie „nach außen“ kommuniziert werden. Um diese beiden Themen es in dieser Frage. Daten- und Ausgabe in allen ihren Formen, sei es durch den Benutzer über die *Oberfläche* des Programms, sei es mit Hilfe von *Dateien* oder mit *Datenbanken*, ist eine Kernanforderung, die alle Programme auf die eine oder andere Weise bedienen müssen.

### ■ Wie arbeite ich mit Programmfunktionen, um Daten zu bearbeiten und Aktionen auszulösen?

Im Rahmen dieser Frage beschäftigen wir uns mit der eigentlichen *Verarbeitung der Daten*. Diese wird in den meisten Programmiersprachen durch sogenannte *Funktionen* geleistet. Auch die Ein- und Ausgabe von Daten (vorherige Frage) bedient sich solcher Funktionen. Funktionen können wir selbst entwickeln, oder wir benutzen Funktionen, die die Programmiersprache selbst oder die Community der Entwickler für uns bereitstellt. Mit Funktionen sicher zu arbeiten zu können, ist ein Schlüssel zum Erfolg bei der Arbeit mit jeder Programmiersprache.

- **Wie steuere ich den Programmablauf und lasse das Programm auf Benutzeraktionen und andere Ereignisse reagieren?**

Unsere Programme sollen nicht starr immer auf die gleiche Art und Weise ablaufen, sondern auf ihre Umwelt reagieren, zum Beispiel auf die Wünsche des Benutzers. Im Rahmen dieser Frage werden uns damit befassen, wie wir unsere Programme *auf äußere Einflüsse und Ereignisse reagieren* lassen und je nach Ereignis im Programmablauf in unterschiedliche alternative Äste verzweigen können.

- **Wie wiederhole ich Programmanweisungen effizient?**

Oft sind Teile eines Programms *Wiederholungen* des immer gleichen Musters. Solche Wiederholungen lassen sich in den meisten Programmiersprachen elegant und effizient mit sogenannten *Schleifen* realisieren. Mit dem in der Praxis äußerst wichtigen Konzept der Schleifen befassen wir uns im Rahmen dieser Frage.

- **Wie suche und behebe ich Fehler auf strukturierte Art und Weise?**

7

Ein (nicht vollkommen triviales) Programm zu schreiben und es gleich beim ersten Versuch zur Perfektion zu führen, ist eine Illusion – auch für Profis. *Fehler* gehören leider unabdingbar zum Programmieralltag. Deshalb beschäftigen wir uns zum Abschluss unserer Tour durch die Grundkonzepte der Programmierung noch damit, wie wir systematisch Fehler finden, diagnostizieren und eliminieren können, und welche Werkzeuge uns dabei helfen können.



# Was brauche ich zum Programmieren?

## Inhaltsverzeichnis

- 8.1 Werkzeuge – 56**
  - 8.1.1 Compiler und Interpreter – 56
  - 8.1.2 Code-Editoren – 57
  - 8.1.3 Integrierte Entwicklungsumgebungen (IDEs) – 58
  - 8.1.4 Einfache Online-Entwicklungs-umgebungen – 64
- 8.2 Hilfe und Informationen – 65**
- 8.3 Ihr Fahrplan zum Erlernen einer neuen Programmiersprache – 67**

## Übersicht

Bevor wir damit beginnen können, Programme zu schreiben, müssen wir uns zunächst die richtigen (Software-)Werkzeuge bereitlegen.

Dazu gehören neben dem Compiler bzw. Interpreter, die den in der Programmiersprache verfassten Programmquelltext in Maschinensprache übersetzen und so für den Computer ausführbar machen, auch Code-Editoren, mit denen der Quelltext des Programms überhaupt erst geschrieben wird. Sogenannte integrierte Entwicklungsumgebungen fassen diese und weitere Werkzeuge unter dem Dach einer gemeinsamen Benutzeroberfläche zusammen.

Neben solchen eher „technischen“ Werkzeugen werden Sie aber von Zeit zu Zeit auch inhaltlich Hilfe beim Programmieren benötigen. Deshalb beschäftigen wir uns in diesem Kapitel auch damit, wie und wo Sie Informationen und Unterstützung rund um Ihre Programmiersprache finden können.

In diesem Kapitel werden Sie folgendes lernen:

- wie Sie sich einen Compiler/Interpreter für Ihre Programmiersprache beschaffen
- welche Funktionen Code-Editoren bieten, und wie sie sich von „normalen“ Texteditoren unterscheiden
- welche Funktionen integrierte Entwicklungsumgebungen bieten, und wie sie sich von reinen Code-Editoren unterscheiden
- welche beliebte Code-Editoren und integrierte Entwicklungsumgebungen es gibt
- wie und wo Sie Informationen und Unterstützung zu Ihrer Programmiersprache im Internet finden.

8

## 8.1 Werkzeuge

» „Wie oft habe ich Euch gesagt, dass Ihr für jede Arbeit das richtige Werkzeug verwenden sollt!“

(Montgomery „Scotty“ Scott in „Star Trek VI – Das unentdeckte Land“)

### 8.1.1 Compiler und Interpreter

Aus dem vorangegangenen Kapitel wissen Sie bereits, dass Sie – je nach Programmiersprache – einen Interpreter bzw. Compiler benötigen, um Ihre Programme in den für den Computer verständlichen, ausführbaren Maschinencode zu übersetzen bzw. ihren Programmquelltext direkt interpretieren und ausführen zu lassen (sollte Ihnen der Unterschied nicht mehr präsent sein, blättern Sie nochmal einige Seiten zu ► Abschn. 3.2 zurück). Für viele Programmiersprachen lassen sich Compiler bzw. Interpreter kostenlos aus dem Internet herunterladen. Das gilt insbesondere für Sprachen, bei denen die Weiterentwicklung von einer de facto gemeinnützigen Organisation betrieben wird, wie es beispielsweise bei Python oder R der Fall ist. Selbst aber für proprietäre Sprachen, die nur von einem bestimmten, kommerziellen Anbieter zur Verfügung gestellt werden (wie etwa der Object Pascal-Dialekt

Delphi, den das amerikanische Unternehmen Embarcadero entwickelt und vertreibt), gibt es oft eine sogenannte *Community Edition*, also eine kostenfreie Version mit einem etwas eingeschränkten, aber für den privaten Anwender – zumal den Anfänger – absolut ausreichenden Funktionsumfang. Die Verwendung der Community Edition zur Entwicklung kommerzieller Anwendungen könnte allerdings Restriktionen unterliegen. Haben Sie also vor, Ihre selbst entwickelte Software zu verkaufen, informieren Sie sich im Vorfeld über die Lizenzbedingungen. Für manche Programmiersprachen ist aber überhaupt kein separater Interpreter erforderlich. Wollen Sie etwa mit JavaScript programmieren, übernimmt der Webbrowswer die Interpretation und Ausführung Ihres Programmcodes. Wollen Sie serverseitige Anwendungen mit PHP entwickeln, läuft die Interpretation des Codes direkt auf dem Server (mit Hilfe eines dort installierten Interpreters) ab, nur deren Ergebnisse werden an den Client zurückgegeben und im Browser sichtbar gemacht.

### 8.1.2 Code-Editoren

---

Um Compiler bzw. Interpreter überhaupt sinnvoll einsetzen zu können, müssen Sie aber zunächst mal ein Programm *schreiben*. Wie wir bereits gesehen haben, ist ein Programm letztlich nicht mehr als ein Text aus Anweisungen, die in einer speziellen Sprache verfasst sind. Die Betonung liegt dabei auf *Text*. Weil der Quelltext Ihres Programms eben einfach ein Text ist, können Sie zu seiner Bearbeitung jedes Programm verwenden, das es erlaubt, unformatierte (also nicht mit speziellen Formatanweisungen wie Fett- oder Kursivsatz versehene) Texte zu editieren. Wenn Sie Windows als Betriebssystem verwenden, steht Ihnen beispielsweise der sehr einfache *Windows Editor* zu Verfügung. Der kann allerdings kaum viel mehr als eine Datei zu öffnen oder neu zu erzeugen, Text hineinzuschreiben oder zu ändern und die Datei nach der Bearbeitung abzuspeichern. Auch wenn diese rudimentären Funktionen im Prinzip bereits ausreichen, um Computerprogramme zu schreiben, so empfiehlt es sich gleichwohl, einen Editor zu verwenden, der speziell für das Programmieren entwickelt worden ist oder doch zumindest von Haus aus Funktionen mitbringt, die das Programmieren erleichtern.

Eine besonders wichtige solche Funktion ist das sogenannte *Syntax Highlighting*. Wie Sie sich erinnern werden, ist die Syntax gewissermaßen die Grammatik der Programmiersprache. Syntax Highlighting ist, wie man sich mit etwas Phantasie aus dem Namen bereits erschließen kann, eine Funktionalität, die bestimmte Teile des Programmcodes farblich hervorhebt, um den Code lesbarer zu gestalten. So werden zum Beispiel Schlüsselwörter (also besondere, reservierte „Wörter“ der Programmiersprache) oder die Bezeichnungen von Variablen jeweils durch unterschiedliche Schriftfarben und/oder Schriftstile (etwa Fettsatz) markiert. Programmieren ohne die visuelle Unterstützung durch Syntax Highlighting ist natürlich allemal möglich, aber erheblich weniger komfortabel, weil es ohne die Hervorhebungen schwerer fällt, den Blick direkt auf die richtigen Stellen im Programmcode zu richten und bestimmte Strukturen im Code schnell zu erfassen. Da die Syntax natürlich von Programmiersprache zu Programmiersprache variiert, muss auch

das Syntax Highlighting bei jeder Sprache anders funktionieren. Etliche Texteditoren unterstützen ab Werk oder durch entsprechende Erweiterungspakete eine Vielzahl unterschiedlicher Programmiersprachen mit Syntax Highlighting. Editoren, die in diese Kategorie fallen, sind zum Beispiel *Atom*, *Notepad++*, *Sublime Text*, *Vim* oder *Visual Studio Code*. Viele der Editoren sind entweder vollkommen oder zumindest in einer funktional etwas eingeschränkten Version, mit der sich sehr gut arbeiten lässt, kostenfrei verfügbar, wie etwa der populäre Sublime Text.

Angesichts der Vielzahl von Texteditoren auf dem Markt, die spezielle Features zur Arbeit mit Programmcode mitbringen, kann eine solche Liste natürlich nicht erschöpfend sein. Und so verwundert es nicht, dass es im Internet eine Unmenge an Artikeln, Blog-Beiträgen und Videos gibt, die sich mit der Frage beschäftigen, welcher denn nun der beste Code-Editor ist. Diese Entscheidung ist natürlich abhängig von persönlichen Vorlieben, insbesondere in Hinblick auf die angebotenen Funktionen und sicherlich auch auf die Optik; das Auge programmiert schließlich mit. Ein besonderer Aspekt der Optik sind die *dark themes*: Vielleicht haben Sie schon einmal einem erfahrenen Programmierer über die Schulter geschaut und dabei gesehen, dass er vor einem Editor mit dunklem, fast schwarzem Hintergrund sitzt, von dem sich der durch das Syntax Highlighting bunt eingefärbte Programmcode deutlich abhebt. Warum ist dieser dunkle Hintergrund so populär? Vielleicht liegt es daran, dass es einfach „cool“ ist, seinem Editor so einzustellen, zeigt es doch, dass man zur geheimnisvollen Community der Programmierer gehört, die unentwegt für den Normalmenschen einigermaßen unverständliche Codes in ihre Tastatur hämmern. Diese Vermutung mag teilweise auch zutreffen. Viel wichtiger allerdings ist ein anderer Faktor: Der dunkle Hintergrund ist für die Augen erheblich angenehmer als ein heller, gar weißer Hintergrund. Wenn Sie sich stundenlang auf Ihren Programmcode konzentrieren müssen, werden Sie rasch den dezenten und weniger blendenden Hintergrund sehr zu schätzen lernen, vor dem sich der Programmcode kontrastreich in den Vordergrund schiebt. In diesem Buch sind die Bildschirmfotos von Code-Editoren immer hell gehalten, aber nur deshalb, weil sie sich so besser im Druck abbilden lassen. Tatsächlich arbeitet der Autor meist mit einem dunklen Hintergrund, der sich bei den meisten Editoren einstellen lässt und bei manchen (etwa Sublime) sogar als Standard voreingestellt ist.

### 8.1.3 Integrierte Entwicklungsumgebungen (IDEs)

Neben (Code-)Editoren gibt es noch weitere Gruppe von Werkzeugen, die *Integrierten Entwicklungsumgebungen*, englisch *Integrated Development Environments* oder kurz *IDEs*. Diese Tools gehen in ihrer Funktionalität über einen reinen *Code-Editor* hinaus.

Wichtige Features, die die meisten IDEs unter einem Dach vereinen, sind:

- Direkter Aufruf des Compilers/Interpreters
- Funktionen zur effizienten Quellcode-Bearbeitung, die über Syntax Highlighting erheblich hinausreichen

## 8.1 · Werkzeuge

- Funktionen zum graphischen Aufbau von Benutzeroberflächen, sofern die Sprache grafische Benutzeroberflächen unterstützt
- Funktionen zur Fehlersuche und -behebung (*Debugging*)
- Funktionen zur Verwaltung ganzer Projekte mit mehreren Dateien

Eine häufig vorzufindende Funktion im Bereich der Quellcode-Bearbeitung ist zum Beispiel die Autovervollständigung: Beim Tippen einiger Buchstaben erscheinen als Vorschlag direkt mögliche Anweisungen/Schlüsselwörter, die zu Ihrer Eingabe passen. So lässt sich die Geschwindigkeit des Code-Schreibens erhöhen und zugleich die Fehleranfälligkeit verringern. Auch eine automatische Syntaxprüfung ist oftmals vorhanden, die Ihnen bereits mögliche Fehler anzeigt, während Sie noch an Ihrem Programmcode arbeiten, Sie also zum Beispiel darauf aufmerksam macht, wenn Sie eine Klammer zwar geöffnet, später aber nicht wieder geschlossen haben. Auf diese Weise können Sie Fehler im Programm frühzeitig erkennen und beheben und werden nicht erst beim Aufruf des Compilers oder Interpreters von entsprechenden Fehlermeldungen überrascht. Zu praktischen Features im Bereich der Quellcode-Bearbeitung gehört oftmals auch eine tiefe Integration der Hilfe. So ist es häufig möglich, direkt aus der IDE heraus Hilfeinformationen zu der Anweisung, mit der Sie gerade arbeiten, aufzurufen.

Nicht selten müssen Sie aber für Ihr Programm nicht nur Code schreiben, sondern auch eine *Benutzeroberfläche* (englisch *graphical user interface*, kurz *GUI*) gestalten. Manche IDEs bieten dem Entwickler dabei umfangreiche Unterstützung. Oft können Sie Ihre Oberfläche aus Standardelementen wie Buttons, Eingabefeldern oder Listenboxen per Drag & Drop zusammenklicken und dann zahlreiche Eigenschaften der Standardelemente nach Ihren Wünschen anpassen, zum Beispiel einem Button eine neue Beschriftung geben. Danach müssen Sie lediglich noch die Oberflächenelemente mit ihrem Programmcode verbinden und aus dem Programm heraus ansprechen, damit auch etwas geschieht, wenn der Benutzer zum Beispiel auf den Button klickt. Integrierte Entwicklungsumgebungen, die die Oberflächengestaltung in dieser Weise unterstützen, sind zum Beispiel Embarcaderos *RAD Studio* oder Microsofts *Visual Studio*.

Neben dem direkten Aufruf von Compiler oder Interpreter und praktischen Funktionen zur Quellcode-Bearbeitung und Oberflächengestaltung bieten IDEs typischerweise auch Funktionen zum Debugging, also der systematischen Fehlersuche und -behebung. Wichtige Features in diesem Bereich sind zum Beispiel die Einrichtung von Haltepunkten (englisch *breakpoints*) im Quellcode. Startet man, nachdem man einen Haltepunkt eingerichtet hat, dann ein Programm, so läuft es erst mal nur bis zu der Stelle im Code, an der sich der Haltepunkt befindet; später kann man es dann manuell über den Haltepunkt hinaus weiterlaufen lassen. Die Arbeit mit Haltepunkten erlaubt es zum Beispiel, zu prüfen, ob ein Programm bis zu der durch den Haltepunkt markierten Stelle fehlerfrei durchläuft. Auch kann man sich den Inhalt von Variablen an der Stelle des Haltepunkts anschauen. Diese Überwachung des Inhalts von Variablen ist ein weiteres wichtiges Debugging-Feature. Es gestattet, während das Programm läuft in Variablen „hineinzuschauen“ und ihren aktuellen Inhalt zu sehen, ggf. sogar zu verändern.

Wenn Ihr Programm aus mehreren Quelltext-Dateien besteht, können Sie diese in einer IDE oft als zusammenhängendes Projekt speichern. Damit genügt es, das Projekt zu öffnen, und schon stehen Ihnen alle Code-Datei zur Verfügung. Auch können Sie bestimmte Einstellungen, etwa des Compilers, projektspezifisch speichern.

Wie im Fall der Code-Editoren auch, gibt es am Markt auch eine schier unüber- schaubare Menge von IDEs, manche kostenlos (Open Source oder als Community Edition), manche kostenpflichtig. Einige unterstützen nur eine Programmiersprache (zum Beispiel *RStudio* der gleichnamigen Firma für die Statistik-Sprache R oder *PyCharm* von JetBrains für Python), andere können, manchmal durch ent- sprechende Plug-ins mit unterschiedlichen Sprachen umgehen (zum Beispiel Mic- rosofts *Visual Studio* oder die Open-Source-Lösung *NetBeans*).

IDEs für mobile Anwendungen, wie etwa Googles *Android Studio*, erlauben es zudem, den Betrieb der entwickelten App auf einer mobilen Umgebung mit be- stimmten Parametern (zum Beispiel Hardware-Ausstattung, Konfiguration) zu si- mulieren und die Beanspruchung der System-Ressourcen (wie etwa Prozessoraus- lastung oder Mobildatentransfer) abzuschätzen. Diese IDEs sind also eher zu einem *bestimmten Zweck*, nämlich der Entwicklung mobiler Anwendungen, als um eine *bestimmte Programmiersprache* herum designt und unterstützen daher oft auch unterschiedliche Sprachen; im Falle von Android Studio etwa C/C++, Java und Kotlin.

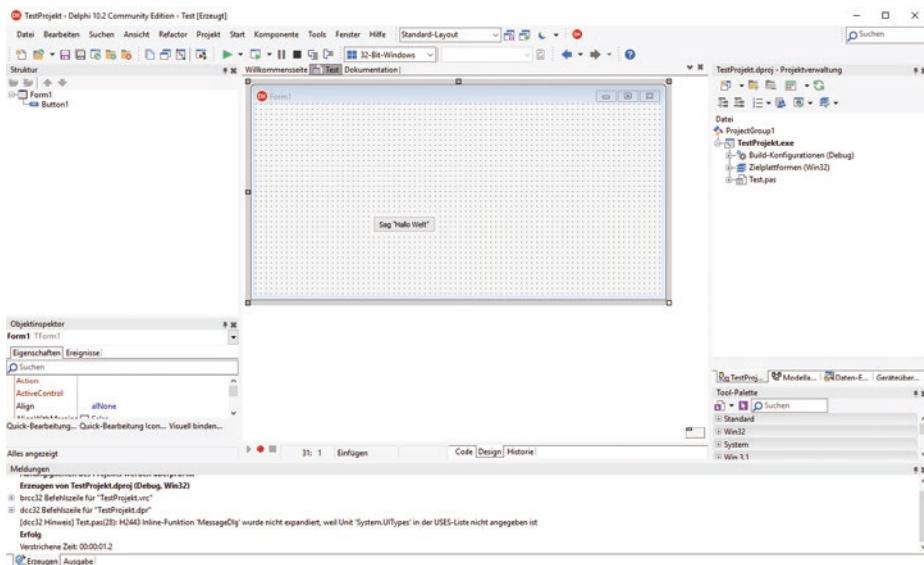
Der Übergang zwischen Code-Editoren und IDEs ist einigermaßen fließend. Viele Code-Editoren erlauben es, einen Compiler oder Interpreter anzubinden und besitzen damit bereits die Kernfunktionalität einer IDE, bieten aber mit Ausnahme von Syntax Highlighting keine Unterstützung im Bereich der sprachspezifischen Code-Bearbeitung, des Debuggings oder des Oberflächen-Designs.

Die Abbildungen □ Abb. 8.1, 8.2, 8.3 und 8.4 zeigen unterschiedliche IDEs. In Abbildung □ Abb. 8.4 sehen Sie eine sehr alte IDE für C/C++, die noch unter MS-DOS lief. Sehr schön erkennt man hier aber wichtige IDE-Funktionen in der Me- nüleiste, wie zum Beispiel Ausführen („Run“), Kompilieren („Compile“), Debug- gen („Debug“) und Features zur Projektverwaltung („Project“).

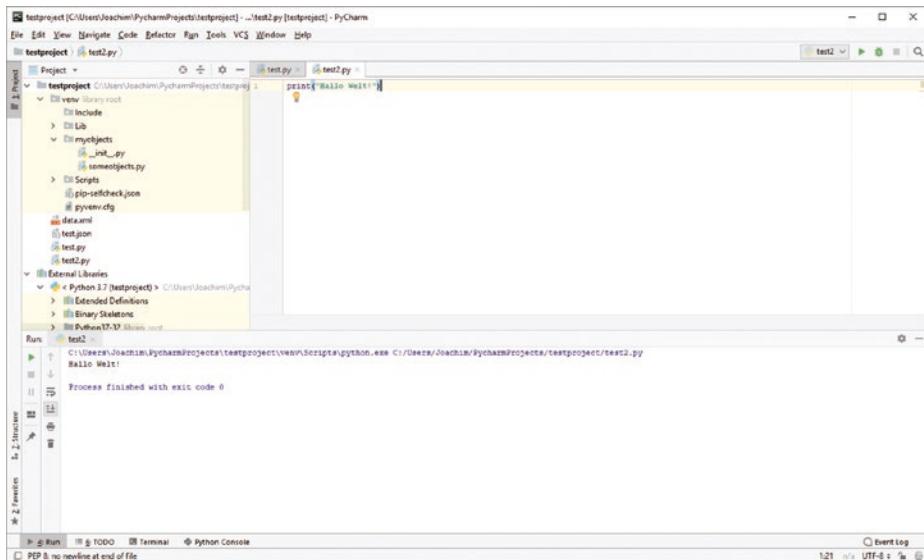
Tabelle □ Tab. 8.1 zeigt für einige gebräuchliche Programmiersprache eine Auswahl von IDEs, die diese Sprachen von Haus aus („nativ“) unterstützen. Etli- che IDEs, wie zum Beispiel *Eclipse* oder *NetBeans*, lassen sich durch Add-ins so erweitern, dass sie eine ganze Reihe von Sprachen unterstützen.

Bei der Entscheidung, ob man einen Code-Editor oder eine IDE verwendet, spielt natürliche eine Rolle, wie stark man Werkzeuge, die nur die IDEs bereitstel- len (etwa Debugging-Features oder Funktionen für das Design von Oberflächen) tatsächlich nutzen will. Selbst aber, wenn man das nicht vorhat, ist der große Vor- teil der IDEs, alle Funktionen unter einem Dach anbieten zu können, immer noch ein gewichtiger, gerade wenn man an die zentralen Werkzeuge wie Compiler oder Interpreter denkt, die man während des Entwicklungsprozesses ständig benötigt. Andererseits sind die IDEs oft selbst recht komplexe Programme, mit einer Un- menge an Buttons, unterschiedlichen Toolbars/Ribbons, Fenstern und Register-

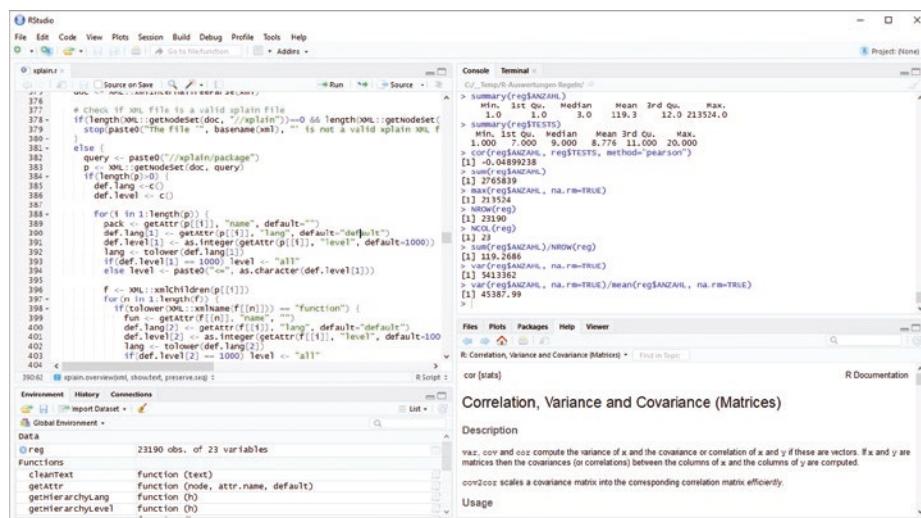
## 8.1 · Werkzeuge



■ Abb. 8.1 Die integrierte Entwicklungsumgebung (IDE) von Delphi



■ Abb. 8.2 Die integrierte Entwicklungsumgebung (IDE) PyCharm für Python



8

Abb. 8.3 Die integrierte Entwicklungsumgebung (IDE) RStudio für R

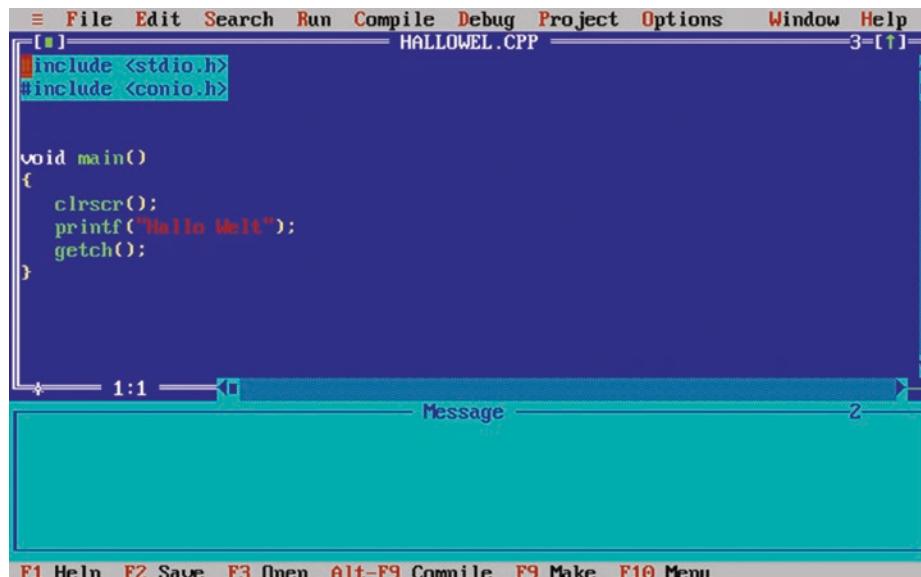


Abb. 8.4 Die integrierte Entwicklungsumgebung (IDE) TurboC für C/C++

■ Tab. 8.1 Ausgewählte integrierte Entwicklungsumgebungen (IDEs)

Programmiersprache	Ausgewählte von IDEs
C#	SharpDevelop, Visual Studio
C/C++	AppCode, C++ Builder, CLion, NetBeans, QT Creator, Visual Studio
Java	AppCode, Eclipse, IntelliJ IDEA, JBuilder, Net Beans
JavaScript	AppCode, Aptana Studio, NetBeans, RubyMine, Visual Studio, WebStorm
Perl	Komodo IDE, Padre
PHP	Aptana Studio, Komodo IDE, NetBeans, PhpStorm, Zend Studio
Python	Aptana Studio, <i>PyCharm</i> , Rodeo, Spyder, Thonny
R	Rcommander, RStudio
Ruby	Aptana Studio, Komodo IDE, RubyMine
Swift	AppCode, Xcode
VBA	Microsoft Office, Visual Studio Tools for Office (VSTO)

karten, in denen man sich erst einmal zurechtfinden muss. Einen guten Eindruck von der Komplexität vermitteln die Abbildungen ■ Abb. 8.1 bis ■ Abb. 8.3. Da die IDEs für Profis entwickelt worden sind, kommt es den Herstellern erkennbar nicht so sehr darauf an, dass man in wenigen Minuten alle Funktionen und Möglichkeiten verstanden hat und anwenden kann; schließlich soll die IDE ja nicht temporär oder nur ab und zu, sondern ständig als Schaltzentrale aller Programmierarbeiten verwendet werden. Deshalb darf es ruhig etwas dauern, bis man die Möglichkeiten des neuen Werkzeugs vollends überblickt. Hat man sich aber erst einmal in der IDE zurechtgefunden, kann man sehr produktiv arbeiten, denn genau dafür werden diese Tools entwickelt.

### Tipp

Probieren Sie unterschiedliche Code-Editoren und IDEs aus und schauen Sie, womit Sie am besten zurechtkommen. Lassen Sie sich nicht von den vielen Funktionen verwirren. Um Programme zu schreiben und laufen zu lassen, brauchen Sie nur sehr wenige dieser Features. Sie werden mit der Zeit mehr und mehr Funktionen der Tools kennenlernen und sicher auch schätzen lernen. Versuchen Sie nicht gleich zu Beginn, alles auszuprobieren und zu verstehen, sondern konzentrieren Sie sich zunächst im ersten Schritt auf die wirklich essentiellen Features, also die zum Bearbeiten und Kompilieren/Ausführen des Codes.

Letztlich müssen Sie selbst für sich entscheiden, welche Werkzeuge Sie verwenden wollen. Dabei ist es durchaus empfehlenswert, unterschiedliche Code-Editoren/IDEs auszuprobieren und sich erst dann festlegen. Im Rahmen dieses Buches werden wir sowohl mit einer vollwertigen IDE arbeiten, nämlich mit *PyCharm* für Python, als auch mit einem klassischen Code-Editor, *Sublime Text*, wenn wir uns mit JavaScript beschäftigen. Beim Schreiben dieser Absätze fällt dem Autor auf, dass er selbst offenbar unbewusst einer einfachen Regel folgt: Wenn die Programmiersprache keinen speziellen, eigenständigen Compiler oder Interpreter auf dem eigenen Computer benötigt (also etwa bei JavaScript oder PHP) ist ein Code-Editor das Werkzeug der Wahl, bei Sprachen, die einen installierten Compiler oder Interpreter voraussetzen, kommt eine (jeweils sprachspezifische) IDE zum Einsatz. Letztlich gibt es aber keine goldene Regel, die immer und für jeden zutrifft. Es hilft nur eines: Ausprobieren!

### 8.1.4 Einfache Online-Entwicklungsumgebungen

---

## 8

Wenn Sie eine Sprache zunächst einfach einmal ausprobieren wollen, ohne gleich alle dafür nötigen Werkzeuge auf Ihrem Computer zu installieren, können Sie in vielen Fällen auf spezielle Webseiten zurückgreifen, die es erlauben, Code direkt einzugeben und auszuführen. Alle zur Ausführung notwendigen Features wie Kompilieren und Interpretieren des Codes werden durch die Webseite bereitgestellt. Einige Beispiele solcher „Online-IDEs“ sind ► <http://cpp.sh/> für die Programmiersprache C++, ► <https://www.compilejava.net/> für Java, ► <https://js.do/> für JavaScript, ► <http://phptester.net/> für PHP, ► <https://www.pythonanywhere.com/> für Python und ► <https://rextester.com/>. Letztere erlaubt es, gleich eine ganze Reihe von Sprachen auszuprobieren, darunter auch die bislang noch nicht genannten C#, Haskell, Kotlin, Ruby, Pascal und Visual Basic.

Weitere solcher Angebote lassen sich sehr einfach finden, indem man „try programmiersprache online“ (wobei *programmiersprache* dann durch den Namen der interessierenden Sprache ersetzt wird) in eine Suchmaschine eingibt.

Meist bedarf es nicht einmal des Anlegens eines Accounts, sondern man kann direkt damit beginnen, Code zu schreiben. Seiten, die einen (kostenlosen) Account voraussetzen, wie etwa ► <https://www.pythonanywhere.com/> erlauben es meist auch, Dateien in der Cloud abzuspeichern und später wiederzuverwenden.

Webangebote wie die genannten sind natürlich kein Ersatz für eine richtige Entwicklungsumgebung, da sie üblicherweise eine sehr beschränkte Funktionalität besitzen und auch die Ausführung von Programmen unter Umständen Restriktionen unterliegt (so darf ein Programm in einigen Fällen zum Beispiel nur fünf Sekunden Rechenzeit in Anspruch nehmen). Will man sich ernsthaft mit der Sprache beschäftigen, führt kein Weg daran vorbei, die notwendigen Werkzeuge auf dem eigenen Computer zu installieren. Nichtsdestotrotz sind diese Seiten eine interessante Möglichkeit, einmal eine Sprache ohne Risiko und unnötigen Aufwand auszuprobieren.

## 8.2 Hilfe und Informationen

---

Die richtigen Werkzeuge zur Verfügung zu haben, ist aber noch nicht alles. Von Zeit zu Zeit werden Sie *Hilfe* benötigen. Deshalb macht es Sinn, sich bereits im Vorfeld Gedanken darüber zu machen, wo Sie weitere Informationen zu Ihrer Programmiersprache erhalten, beispielsweise, wenn Sie keine Vorstellung haben, wie Sie ein bestimmtes Problem überhaupt angehen sollen, nicht wissen, was bestimmte Befehle Ihrer Programmiersprache tun oder wie man sie einsetzt, oder aber Sie die mitunter kryptischen Fehlermeldungen des Interpreters bzw. Compilers nicht verstehen. In all diesen Fällen ist es hilfreich, sofort eine Anlaufstelle parat zu haben, bei der Sie Unterstützung finden können.

Solche Anlaufstellen sind in der praktischen Arbeit eine bedeutende Ressource, nicht nur für Programmieranfänger. Neben Büchern wie diesem bietet natürlich das Internet eine Unmenge von Quellen, die kaum ein Informationsbedürfnis unbefriedigt zurücklassen – vorausgesetzt natürlich, man findet sie.

Viele Programmiersprachen, sowohl Open-Source- als auch proprietäre (herstellergebundene) Sprachen bringen eine umfangreiche Web-Dokumentation mit, in der man vor allem ausführliche Informationen zu spezifischen Befehlen der Programmiersprache findet. Beispiele für diese Hilfsangebote sind die *Function Reference* von PHP, die *Library Reference* von Python oder Microsofts *VBA-Referenz* für Visual for Applications (VBA).

Bestandteil der offiziellen Dokumentationen ist meist aber nicht nur eine solche *Funktionsreferenz*, also ein wörterbuch-ähnliches Nachschlagewerk, das beschreibt, was bestimmte Befehle der Programmiersprache tun und wie sie verwendet werden, sondern vielfach auch eine *Sprachreferenz*. Sprachreferenzen erläutern die Grammatik der jeweiligen Sprache, die Syntax, und beschreiben so, wie man korrekt Sätze, also Anweisungen, in der Sprache formuliert. Solche Sprachreferenzen sind allerdings nicht immer für programmierunfahrene Einsteiger gut bekommlich.

Deshalb bieten manche offiziellen Dokumentationen auch Tutorials für Einsteiger und „Getting started“-Artikel als ergänzende Komponente.

Sich ein Browser-Lesezeichen auf die offizielle Sprachdokumentation, insbesondere auf die Funktionsreferenz, der jeweiligen Sprache zu setzen, ist jedem zu empfehlen, der sich ernsthaft mit einer Programmiersprache befassen möchte. Es ist meist die erste Anlaufstelle, wenn man verstehen möchte, was ein bestimmter Befehl tut und wie genau er zu verwenden ist.

Neben den offiziellen Dokumentationen gibt es natürlich auch inoffizielle Informations- und Hilfekanäle, die nicht von der Organisation, die für die Programmiersprache verantwortlich zeichnet, betrieben werden. Der in der Praxis für viele Programmiersprachen wichtigste solche Kanal ist das englischsprachige Internetforum *Stack Overflow* (► <https://StackOverflow.com/>). Mit über 17 Millionen Fragen zu den unterschiedlichsten Programmiersprachen lässt es kaum Wünsche offen. Sucht man dort nach einer Lösung für ein konkretes Problem, bekommt man tatsächlich oftmals schnell den Eindruck, dass jede nur erdenkliche Frage schon

mal von irgendjemandem zuvor gefragt worden ist. Aber nicht nur die große Zahl von beantworteten Fragen und die hohe Abdeckung unterschiedlicher Programmiersprachen machen Stack Overflow zu einer ungemein nützlichen Informationsquelle. Auch die Qualität der Antworten ist im Allgemeinen sehr hoch.

Erreicht werden die hohe Quantität und Qualität zum einen durch Gamification, also das spielerische Setzen von Anreizen, indem Forumteilnehmern für bestimmte Aktionen Punkte („Reputation“) gutgeschrieben werden. Diese Punkte prangen nicht nur als Kompetenzausweis für jedermann sichtbar neben dem jeweiligen Username, sondern erlauben es auch, nach und nach bestimmte Funktionen zu nutzen, die nicht allen Benutzern zur Verfügung stehen. Eine wichtige solche Funktion (die man sich bereits mit einem verhältnismäßig niedrigem Punktestand erschließen kann) ist das Up- und Down-Voten von Antworten, also die Bewertung der Antworten anderer User. Das hilft den Lesern, die Qualität der Antworten besser einzuschätzen. Die Verfasser von hoch-gevoteten Antworten erhalten wiederum Reputationspunkte, was den Anreiz, qualitativ hochwertige Antworten zu liefern, erhöht. Auch kann der Fragesteller eine der Antworten als die beste markieren. Andere Benutzer erkennen dann an dem großen grünen Haken neben einer Antwort, dass es sich um diejenige Antwort handelt, die dem Fragesteller letztlich geholfen hat, sein Problem zu lösen. Auch hierfür erhält der Autor der Antwort Reputationspunkte gutgeschrieben. Auf diese Weise wird nicht nur ein System gegenseitiger Qualitätskontrolle betrieben, sondern auch ein starker Anreiz dafür gesetzt, Arbeit für andere zu investieren, von der man bei nüchterner Betrachtung eigentlich nichts hat als das gute Gefühl, einem anderen Benutzer geholfen zu haben. Bei Stack Overflow gewinnt man außer dem guten Gefühl zusätzlich Reputation und Rechte, was dem Ego vieler Teilnehmer erkennbar sehr zuträglich ist.

Neben der Gamification spielen für die hohe Qualität der Informationen auch die strikten Regeln im Forum eine Rolle. So werden etwa Fragen, die Duplikate zu anderen Fragen sind, oder die im jeweiligen Forenbereich off topic sind, von den Moderatoren sofort geschlossen. Vom Fragesteller wird gefordert, zu seinem Problem ein minimales, aber lauffähiges Code-Beispiel zu liefern und seine Frage bereits im Titel des Posts genau zu formulieren. Obwohl der Stack Overflow-eigene *Code of Conduct* einen freundlichen Umgang miteinander fordert, mutet der Ton im Forum manchmal etwas rüde an, auch gegenüber Stack Overflow-Neulingen.

Wenngleich der Stil bisweilen ein wenig gewöhnungsbedürftig ist, ist Stack Overflow eine erstklassige Informationsquelle. Auch zu unverständlichen Compiler- oder Interpreter-Fehlermeldungen – ein häufiges Ärgernis beim Programmieren – wird man bei Stack Overflow meist fündig. Da Stack Overflow viele unterschiedliche Programmiersprachen unterstützt, ist es wichtig, den Namen der Sprache immer in die Suchanfrage mit aufzunehmen.

Sucht man nach einer Fragestellung auf Google, so sind die Suchtreffer von Stack Overflow regelmäßig weit oben gelistet. Durch das Hinzufügen von „site:StackOverflow.com“ zur Google-Suchanfrage werden sogar ausschließlich Ergebnisse von Stack Overflow gelistet; natürlich hat StackOverflow.com aber auch eine eigene Suche.

### 8.3 · Ihr Fahrplan zum Erlernen einer neuen Programmiersprache

Neben Stack Overflow gibt es eine Vielzahl weiterer Foren, unter anderem auf bekannten Plattformen wie Facebook oder Reddit. Auch zahlreiche Blogs liefern immer wieder gute Hinweise und How-Tos für spezifische Probleme.

Wann aber soll man nun die offizielle Dokumentation verwenden, wann andere Quellen, wie etwa Stack Overflow?

Wenn Sie bereits wissen, welchen Befehl Sie verwenden müssen und es jetzt nur noch darum geht, zu verstehen, wie man das genau macht, dann ist in der Regel die offizielle Funktionsreferenz der Sprache die beste Anlaufstelle. Wenn Sie aber noch gar nicht wissen, mit welchem Befehl oder mit welchem Ansatz Sie ein Problem angehen sollen, dann ist Stack Overflow (oder ein inhaltlich ähnliches Forum) die erste Wahl. Hier finden Sie zu ganz vielen häufig vorkommenden Fragestellungen gute Lösungsansätze. Ähnliches gilt, wenn Sie die Fehlermeldungen von Compiler oder Interpreter nicht dechiffrieren können oder partout nicht verstehen, warum sich ein Befehl, den Sie in der Funktionsreferenz nachgeschlagen haben, gerade so verhält, wie er sich verhält. Das Schöne an Stack Overflow ist dabei: Aufgrund der Vielzahl an bereits gestellten Fragen wird mit einiger Wahrscheinlichkeit auch Ihre dabei sein und Sie bekommen Ihr Problem sofort gelöst, ohne selbst eine Frage posten und dann stunden- oder gar tagelang auf eine adäquate Antwort warten zu müssen.

## 8.3 Ihr Fahrplan zum Erlernen einer neuen Programmiersprache

- **Wenn Sie eine neue Programmiersprache lernen ...**
  - beschaffen Sie sich zunächst die benötigten Werkzeuge, insbesondere den benötigten Compiler bzw. Interpreter und ein Tool (Code-Editor bzw. IDE), um Ihren Code zu bearbeiten,
  - Wenn Sie bereits für eine andere Sprache einen Code-Editor bzw. eine IDE verwenden, überlegen Sie, ob Sie diese nicht auch für die neue Sprache nutzen können; immerhin sind Sie bereits mit der Bedienung vertraut. Für Ihr Tool gibt es möglicherweise ein Erweiterungspaket, dass es auch für die neue Sprache fit macht.
  - Wenn Sie noch keinen Code-Editor bzw. keine IDE im Einsatz haben, probieren Sie durchaus unterschiedliche aus. Erfordert die Arbeit mit Ihrer Programmiersprache besondere Features, wie etwa für die Gestaltung grafischer Oberflächen, oder wollen Sie intensiv mit Werkzeugen zur Fehlerbehebung und Beseitigung (Debugging) arbeiten, sollten Sie eine IDE einem reinen (Code-)Editor vorzuziehen. Ansonsten genügt ein guter Code-Editor, aus dem heraus Sie aber (sofern im Falle ihrer Sprache notwendig) Compiler bzw. Interpreter auch aufrufen können sollten.
  - Wenn Sie ein neues Tool zur Code-Bearbeitung installieren, versuchen Sie nicht gleich, die ganze Komplexität an Features zu verstehen. Sie brauchen ohnehin – zumindest zu Beginn – nur einen relativ kleinen Teil davon, um

wirklich arbeiten zu können. Konzentrieren Sie sich erst einmal darauf, zu verstehen, wie Sie Code-Dateien öffnen, bearbeiten und speichern können, und wie Sie Ihr Programm ausführen können. Danach können Sie direkt mit dem Programmieren starten. Das meiste Übrige über Ihre integrierte Entwicklungsumgebung bzw. Ihren Code-Editor werden Sie *en passant* lernen.

- informieren Sie sich, bevor Sie richtig in die Programmiersprache einsteigen, welche offizielle Hilfe- und Informationskanäle es gibt; setzen Sie sich insbesondere Browser-Bookmarks zur offiziellen Funktions- und Sprachreferenz.

# Was muss ich tun, um ein Programm zum Laufen zu bringen?

## Inhaltsverzeichnis

- 9.1      Aller Anfang ist leicht – 70**
- 9.2      Hallo, Welt! – 72**
- 9.3      Ihr Fahrplan zum Erlernen  
einer neuen Programmiersprache – 74**

## Übersicht

Nachdem Sie die notwendigen Werkzeuge beschafft sowie Hilfe- und Informationsquellen eruiert haben, wird es Zeit, herauszufinden, wie genau Sie eigentlich ein Programm zum Laufen bringen. Nichts ist frustrierender, als zwar hervorragend mit umfangreichem theoretischem Wissen über Ihre Programmiersprache präpariert zu sein, aber letztlich daran zu scheitern, Ihr erstes kleines Programm überhaupt zu starten.

Um dieses unangenehme Erlebnis zu vermeiden, sollten Sie möglichst zeitig bereits ein erstes, richtiges Programm zu schreiben, selbst, wenn Sie noch fast gar nichts über Ihre Programmiersprache wissen.

In diesem Kapitel werden Sie folgendes lernen:

- warum es wichtig ist, bereits früh ein erstes lauffähiges Programm zu produzieren (und damit Ihren ersten kleinen Erfolg als Programmierer in der neuen Programmiersprache zu feiern)
- was das berühmte „Hallo Welt“-Programm ist und warum es sich gut als allererster Schritt in die neue Programmiersprache eignet
- was der Unterschied zwischen interaktivem und Skript-/Batch-Modus bei der Ausführung der Programmiersprache ist und wann welcher Modus zu bevorzugen ist.

9

### 9.1 Aller Anfang ist leicht

In den folgenden Kapiteln werden wir uns mit vielen Grundkonzepten befassen, deren Umsetzung in Ihrer neuen Programmiersprache Sie verstehen müssen, um wirklich gute, sinnvoll anwendbare Programme entwickeln zu können. Nichtsdestotrotz ist es hilfreich, schon jetzt mit einem sehr simplen Programm zu starten.

Nicht nur können Sie dieses einfache Stück Code als Grundlage für schrittweise Erweiterungen nutzen, mit denen Sie jedes zusätzlich erlernte Element der Sprache direkt praktisch ausprobieren können. Auch lernen Sie auf diese Weise sofort einige Spezifika Ihrer neuen Programmiersprache kennen, die Ihnen viele Kopfschmerzen bereiten können, wenn Sie sie nicht gleich von Anfang verstehen und beachten. Ein gutes Beispiel dafür ist die Notwendigkeit, Anweisungen mit einem speziellen Zeichen, oft einem Semikolon, abzuschließen, wie es eine syntaktisch korrekte Anweisung in vielen Programmiersprachen erfordert. Solche Feinheiten erlernt man am besten anhand eines möglichst einfachen Programms, denn je simpler das Programm, desto weniger alternative Fehlerquellen gibt es und desto einfacher ist folglich die Fehlersuche und -behebung. Das ist wichtig, denn Sie wollen ja schließlich Ihre ersten Schritte in der neuen Programmiersprache nicht mit einer langen, frustrierenden Fehlersuche beginnen. Besser ist es da doch allemal, ein kleines, aber schnelles erstes Erfolgserlebnis zu haben! Dabei ist es an dieser Stelle noch gar nicht so wichtig, dass Sie tatsächlich alles, was Sie mit ihrem ersten Minimal-Programm tun, ganz und gar zu verstehen. Wichtig ist erst mal nur, dass Sie ein kleines, aber lauffähiges Programm schreiben und tatsächlich ausführen können.

„Ausführen“ ist das Stichwort, das uns sogleich zu einem weiteren Argument dafür bringt, direkt mit einem kleinen Programm loszulegen. Denn dabei lernen Sie auch, mit den wichtigsten Funktionen der neuen Werkzeuge umzugehen, insbesondere Quellcode zu kompilieren (sofern das nötig ist) und auszuführen. Wenn Sie mit einer integrierten Entwicklungsumgebung, einer IDE, arbeiten, dann werden die betreffenden Befehle vermutlich über Menüs oder Symbolleisten erreichbar sein. Vielleicht entscheiden Sie sich aber auch dazu, mit einer Kombination aus normalem (Code-)Editor und einem Kommandozeilen-Interpreter-/Compiler zu arbeiten. In diesem Fall müssen Sie verstehen, wie genau das Kommandozeilen-Programm, das Ihren Quellcode kompiliert oder direkt interpretiert, aufgerufen wird, insbesondere, welche Argumente ihm übergeben werden müssen. Das gilt auch dann, wenn Sie den Compiler bzw. Interpreter in Ihren Code-Editor einbinden können, denn das geschieht regelmäßig dadurch, dass Sie dem Code-Editor die betreffenden Kommandozeilen-Anweisung mitteilen. Informationen hierzu finden Sie in aller Regel in der offiziellen Dokumentation der Programmiersprache.

Bei compilierten Sprachen kommt zum Compilieren des Programms regelmäßig noch ein weiterer Schritt hinzu, das sogenannte *Linken* (dt. verbinden). Dabei produziert ein spezielles Tool, der *Linker*, die ausführbare Datei Ihres Programms. Diese besteht zum einen aus Ihrem eigentlichen Programm (dem in Maschinensprache übersetzten Quellcode, den Sie geschrieben haben), zum anderen aus compilierter Programmbibliotheken und anderen compilierten Programmbestandteilen, auf die Sie aus Ihrem Programm heraus zugreifen. Durch das Linken werden alle diese Komponenten zu einer einzigen ausführbaren Datei verbunden, die dann alleine für sich lauffähig ist. Gerade, wenn Sie mit einer IDE arbeiten, werden Sie häufig nicht nur separate Menübefehle für das Compilieren und Linken finden, sondern meist auch einen Befehl, der „Build“ heißt und das Compilieren und Linken – und damit die gesamte Erzeugung der ausführbaren Programmdatei aus Ihrem Quellcode – mit einem einzigen Klick nacheinander erledigt.

In manchen Fällen ist es aber nicht mit ein paar Klicks auf der Oberfläche Ihrer IDE oder dem Aufruf eines Kommandozeilen-Interpreters oder -Compilers getan. Denken Sie etwa an JavaScript, das Sie regelmäßig zunächst in eine Webseite, also ein HTML-Dokument einbetten müssen. Auch das will gelernt sein. Ein erster Versuch mit einem kleinen Beispielprogramm kann da nicht schaden.

Damit sind die Möglichkeiten, Quellcode auszuführen, aber noch nicht erschöpft. Manche Programmiersprachen unterstützen neben dem Ausführen von ganzen Programmen, also Aufstellungen mehrerer Anweisungen, auch einen *interaktiven Modus*. Im interaktiven Modus können Sie zunächst eine Programmanweisung eingeben. Durch Drücken der <ENTER>-Taste oder eines speziellen Buttons, wird diese *eine* Anweisung *sofort* ausgeführt und das Ergebnis unmittelbar angezeigt (sofern es ein darstellbares Ergebnis gibt). Danach können Sie eine weitere Anweisung eingeben und bekommen sogleich wiederum deren Resultat präsentiert. Dieses Vorgehen, das auch als *read-eval-print-loop* (REPL), also Einlesen-Auswerten-Anzeigen-Neubeginn, bezeichnet wird, ist zum Beispiel dann interessant, wenn man sich einen Datenbestand explorativ anschauen und statistische Analysen auf diesem Datenbestand vornehmen möchte. Kein Wunder, dass etwa R und Python, zwei Sprachen, die gerade auf diesem Gebiet erhebliche

Stärken haben, einen interaktiven Modus besitzen. Der interaktive Modus kann aber auch für die Fehlersuche verwendet werden, indem man den oft als *Konsole* bezeichneten interaktiven Interpreter nach und nach mit Anweisungen füttert und sich dann jeweils zunächst das Ergebnis anschaut, bevor man die nächste Anweisung ausführt. Auf diese Weise lässt sich leicht ermitteln, an welcher Stelle genau sich das eigentlich zusammenhängende Programmstück, das man dem Interpreter schrittweise zur Ausführung übergibt, anders verhält, als man es erwartet. Deshalb unterstützen bei weitem nicht nur Sprachen, die zur statistischen Analyse von Daten verwendet werden, einen interaktiven Modus.

Einen interaktiven Modus gibt es regelmäßig nur bei interpretierten Sprachen. Benötigt Ihre Sprache einen Compiler, müsste jede interaktive Anweisung, die Sie eingeben, vom Compiler zu einem eigenen, in sich abgeschlossenen Programm in Maschinensprache übersetzt werden. Dann könnten Sie aber mit dem nächsten Befehl nicht mehr auf Daten zurückgreifen, die Sie zuvor im Speicher bearbeitet haben, denn nach Beendigung des ersten Programms („also des ersten interaktiven Befehls“) wird dessen Speicherbereich vom Betriebssystem wieder freigegeben. Eine echte Interaktivität würden Sie so nicht erreichen. Compilierte Sprachen bieten daher normalerweise lediglich die Ausführung ganzer Programme an. Diese Art, Programme auszuführen, die es natürlich auch bei interpretierten Sprachen gibt, und die die ungleich häufiger anzutreffende Art der Ausführung ist, bezeichnet man in Abgrenzung zum interaktiven Modus auch als *Batch- oder Skript-Modus*.

Machen Sie sich also damit vertraut, wie genau man Programme in der Sprache, die Sie erlernen wollen, eigentlich ausführt.

## 9.2 Hallo, Welt!

Indem Sie ein Minimal-Programm schreiben, Erlernen Sie alles, was Sie brauchen, um die später erlernten Programmierprinzipien bzw. deren Umsetzung in Ihrer Programmiersprache sinnvoll auszuprobieren und Programme nach Belieben ausführen zu können.

Wie kann aber nun ein solches Minimal-Programm aussehen?

Das bekannteste Minimal-Programm, das wohl wirklich jeder Programmierer kennt, ist das „Hallo Welt“-Programm (engl. „Hello, world!“). Dabei handelt es sich um ein Programm, das nichts anderes tut, als den einfachen Satz „Hallo Welt“ auf dem Bildschirm auszugeben. Die Ursprünge dieses wahren Klassikers reichen wohl bis in die frühen Siebziger Jahre des 20. Jahrhunderts zurück.

Im Folgenden einige Beispiele, wie ein Hallo-Welt-Programm in verschiedenen Programmiersprachen aussieht. Zunächst in C:

```
#include <stdio.h>
main ()
{
    printf("Hallo Welt!");
}
```

Wie Sie sehen, steht die eigentliche Ausgabe-Anweisung, die Funktion `printf()`, umgeben von geschweiften Klammern, denen ein `main()` vorangeht. Wenn Sie sich näher mit C beschäftigen, werden Sie lernen, dass ein C-Programm letzten Endes selbst eine Funktion ist, nämlich die Funktion `main()`, die automatisch aufgerufen wird, wenn das Programm gestartet wird. In unserem Beispiel bewerkstelligt diese Funktion nicht anderes, als eine andere Funktion aufzurufen, nämlich `printf()`, die dann den Satz „Hallo Welt!“ auf dem Bildschirm anzeigt. Ohne die Funktion `main()` wird Ihr C-Programm nicht laufen. Ein Problem werden Sie auch bekommen, wenn Sie den abschließenden Strichpunkt hinter der `printf()`-Anweisung vergessen. Und damit `printf()` überhaupt aufrufbar ist, muss mit `#include <stdio.h>` eine Standardbibliothek mit Funktionen zur Ein- und Ausgabe eingebunden werden. Es ist also gar kein ganz triviales Vergnügen, ein C-Programm zum Laufen zu bringen. Andererseits lernen Sie anhand des Hallo-Welt-Programms bereits einige Grundregeln anzuwenden, selbst, wenn Sie deren Bedeutung an dieser Stelle noch nicht voll verstehen mögen. Dennoch wären Sie nun ohne weiteres in der Lage, Ihr Hallo-Welt-Programm zu erweitern, wenn Sie tiefer in die Sprache C einsteigen würden.

Erheblich einfacher strukturiert ist das Hallo-Welt-Programm in Ruby on Rails:

```
print 'Hallo Welt!'
```

In Delphi, wo Sie eine grafische Benutzeroberfläche entwickeln, lernen Sie direkt im Zuge des Hallo-Welt-Programms, eine Message-Box mit Informationsicon und Okay-Button auf den Bildschirm zu zaubern:

```
MessageDlg('Hallo Welt!', mtInformation, [mbOK], 0);
```

Tatsächlich würden Sie in der Praxis das Anzeigen der Message-Box natürlich von irgendeiner Benutzeraktion abhängig machen müssen, zum Beispiel davon, dass der Benutzer Ihres Programms auf einen Button klickt. Das heißt, Ihr Hallo-Welt-Programm würde auch eine minimale grafische Benutzeroberfläche umfassen, bestehend aus einem Fenster mit einem Button. Auf diese Weise würden Sie sogleich mit den Werkzeugen zur Erstellung grafischer Benutzeroberflächen in den Grundzügen umzugehen lernen.

In PHP schließlich macht es Sinn, sich gleich damit zu beschäftigen, wie man ein PHP-Skript mit dem Tag `<?php ... ?>` in eine Website einbindet. Hier könnte ein Hallo-Welt-Programm dann so aussehen:

```
<html>
  <body>
    <?php echo 'Hallo Welt!'; ?>
  </body>
</html>
```

### 9.3 Ihr Fahrplan zum Erlernen einer neuen Programmiersprache

---

- Wenn Sie eine neue Programmiersprache lernen...
  - starten Sie damit, ein ganz einfaches Programm zu schreiben, auch wenn Sie noch nicht in die Tiefen der neuen Programmiersprache eingetaucht sind. Auf diese Weise lernen Sie bereits einige wichtige Spezifika der Sprache kennen. Eine gute Möglichkeit dafür ist die Umsetzung des klassischen „Hallo Welt“-Beispiels, das man für praktisch alle Sprachen auch im Internet findet,
  - beschäftigen Sie sich mit der Frage, welche Möglichkeiten es gibt, Ihre Programme auszuführen (Interpreter im Skript-/Batch-Modus, interaktiver Modus) und wie genau man das bewerkstelltigt. Programme schreiben zu können, ist schön, nützlich ist es erst, wenn man sie auch ausführen kann!



# Wie stelle ich sicher, dass ich (und andere) mein Programm später noch verstehe?

## Inhaltsverzeichnis

- 10.1 Verständlicher Programm-Code – 76**
- 10.2 Gestaltung des Programmcodes und Benamung von Programmelementen – 77**
- 10.3 Kommentare – 80**
  - 10.3.1 Den eigenen Programmcode erläutern – 80
  - 10.3.2 Wozu Kommentare sonst noch nützlich sind – 82
  - 10.3.3 Dokumentation außerhalb des Programmcodes – 83
- 10.4 Ihr Fahrplan zum Erlernen einer neuen Programmiersprache – 85**

» „When I wrote this, only God and I understood what I was doing. Now, only God knows.“

(User johnc auf *StackOverflow* „What is the best comment in source code you have ever encountered?“)

## Übersicht

Dass man auf eine Art und Weise programmieren sollte, die es einem erlaubt, später noch zu verstehen, wie man genau vorgegangen ist, und wie die Lösungen, die man entwickelt hat, funktionieren, ist ein triviale Forderung, die unmittelbar einleuchtet. Doch tatsächlich tun sehr viele Programmierer, durchaus auch berufsmäßige, viel zu wenig dafür, diese Forderung auch in die Tat umzusetzen.

Deshalb beschäftigen wir uns als nächstes damit, wie Sie ihren Programmcode so gestalten können, dass er Ihnen und anderen, die damit arbeiten müssen, später noch zugänglich ist.

In diesem Kapitel werden Sie lernen:

- wie man Programm-Code lesbar gestaltet
- was Kommentare sind und warum sie nützlich sind
- wie man Kommentare geschickt einsetzt, um seinen Programm-Code zu erläutern
- welche Rolle eine Dokumentation des Programm-Codes spielt, wenn andere Programmierer ihn für eigene Entwicklungen benutzen sollen.

10

### 10.1 Verständlicher Programm-Code

Verständlichen Programm-Code zu entwickeln ist mühsam und anstrengend, und das, was dazu notwendig ist, gehört zu den unbeliebtesten Aufgaben, vor die Sie einem Programmierer stellen können.

Denn zum einen ist es mehr Aufwand auf eine verständliche Weise zu programmieren, weil man sich an Konventionen halten und sich Notizen, im Programmierer-Jargon *Kommentare*, schreiben muss, damit man später schnell versteht, wie das Programm, das man selbst geschrieben hat, eigentlich funktioniert. Zum anderen neigen die meisten Menschen – den Autor dieser Zeilen miteingeschlossen – dazu, in dem Moment, da sie gerade gedanklich tief in den Details einer Problemlösung stecken, ihre Fähigkeit massiv zu überschätzen, Ihr eigenes Tun später noch zu verstehen.

Man sollte aber bedenken, und das mag eine zusätzliche Motivation sein, die notwendige Arbeit doch zu investieren, dass dieses „später“ nicht unbedingt nächstes Jahr sein muss. Es kann auch schon übermorgen sein, dass Ihnen die mangelnde Lesbarkeit Ihres Programmcodes zum Verhängnis wird und unnötigen Aufwand verursacht, der mit etwas mehr Mühe beim Schreiben des Codes einige Tage vorher einfach zu vermeiden gewesen wäre.

Deshalb macht es Sinn, sich sehr früh mit der Frage zu beschäftigen, was man tun kann, um dem entgegenzuwirken und sicherzustellen, dass man von Anfang an

so arbeitet, dass man später stets ohne größere Schwierigkeiten wieder in seinen Programmcode einsteigen kann. Drei Faktoren können dazu maßgeblich beitragen:

- Die *optische Gestaltung des Programmcodes* hat Einfluss darauf, wie schnell man sich darin orientieren und Zusammenhänge zwischen unterschiedlichen Teilen erkennen kann.
- Die Art und Weise der *Benennung* verschiedener Elemente eines Programms (wie Variablen und Funktionen, mit denen wir uns später detailliert auseinandersetzen werden), entscheidet maßgeblich darüber, wie schnell man inhaltlich erfassen kann, was bestimmte Teile des Codes tun.
- Kommentare, also vom Programmierer verfaßte Erläuterungen, die direkt im Programmcode untergebracht werden, können, wenn sie gut geschrieben sind, vieles erklären, was man nicht gleich aus dem Code selbst herauslesen kann.

Mit den ersten beiden Faktoren beschäftigen wir uns gleich im folgenden, mit der wichtigen Frage der Kommentierung im übernächsten Abschnitt.

## 10.2 Gestaltung des Programmcodes und Benamung von Programmelementen

---

Ein wichtiger Faktor, der dazu beiträgt, dass Sie Ihren Programmcode später noch leicht verstehen, ist die Art, wie sie ihn formatieren, insbesondere, wie Sie mit Zeileneintrückungen arbeiten. Ein Beispiel:

```
for (i=1; i<=100; i++) {  
    if (i mod 2 == 0) {  
        for (f=1; f<=100; f++) {  
            printf("Feld (", i, ", ", f, ")");  
        }  
        printf("\n");  
    }  
}
```

Auch ohne, dass Sie verstehen, was genau dieser in der Programmiersprache C verfasste Code tut (für Interessierte: er schreibt für eine Matrix mit 100 Zeilen und 100 Spalten alle Feldkoordinaten in der Form (Zeile, Spalte) auf den Bildschirm, lässt dabei aber die ungeraden Zeilen aus) sehen Sie eine ganze Reihe von geschweiften Klammern, die hinter bestimmten Anweisungen aufgehen und irgendwann später wieder geschlossen werden. Alles was dazwischen liegt, ist ein Code-Block, der zu der vorangehenden Anweisung gehört. Die Code-Blöcke sind hier offensichtlich ineinander verschachtelt, das heißt, es gibt Code-Blöcke, die wiederum andere Code-Blöcke enthalten. In der obigen Schreibweise ist diese verschachtelte Struktur aber schwer zu erkennen. Deutlich wird das vor allem am Ende, wo kaskadenartig eine ganze Reihe von Blöcken geschlossen wird. Welche Klammer jetzt zu welchem Block gehört, und damit, welche Anweisung sich in welchem Block befindet, ist nicht einfach auszumachen. Diese Frage ist aber essentiell, wenn man verstehen will, was das Programm tut, und wie es genau arbeitet.

Übersichtlicher wird es, wenn man Einrückungen einfügt, zum Beispiel mit der Tabulator-Taste:

```
for (i=1; i<=100; i++) {
    if(i mod 2 == 0) {
        for(f=1; f<=100; f++) {
            printf("Feld (", i, ", ", f, ")");
        }
    }
    printf("\n");
}
```

Jetzt ist es erheblich leichter, zu erkennen, wo ein Code-Block beginnt und wo er aufhört. Der Code wird *lesbarer*.

Die meisten Programmiersprachen erlauben es, den Code beliebig durch Einrückungen zu formatieren. Es gibt allerdings einige Ausnahmen wie etwa Python, wo Einrückungen eine inhaltliche Bedeutung haben und deshalb nicht nach Beliebigen gesetzt werden können. Allerdings kann auch für Python Entwarnung gegeben werden: Die Art, wie Python die Einrückungen fordert, zwingt den Programmierer geradezu dazu, seinen Code lesbar zu gestalten. Oder anders herum ausgedrückt: Eine korrekte Syntax (also ausführbarer Programmcode) in Python bringt automatisch ein Mindestmaß an Lesbarkeit mit sich. Das werden wir uns im zweiten Teil des Buches genauer ansehen, wenn wir uns intensiver mit Python befassen.

Aber nicht nur die Formatierung des Quellcodes spielt für die Lesbarkeit und Verständlichkeit des Codes eine große Rolle. Beim Programmieren müssen sie vielen Dingen einen Namen geben, zum Beispiel Variablen oder Funktionen. Diese Namen (im Programmierer-Jargon auch *Bezeichner* genannt) sieht der Benutzer Ihres Programms zwar später nicht, das sollte allerdings noch keine ausreichende Rechtfertigung dafür sein, dass Sie bei der Wahl der Bezeichner ihren wildesten Phantasien freien Lauf lassen.

Betrachten Sie dazu als Beispiel folgendes Programm, das in Python geschrieben ist und den Body-Mass-Index (BMI) berechnet:

```
a = input("Ihr Gewicht [in kg]: ")
b = input("Ihre Größe [in m]: ")

c = float(a)/float(b)**2

print("Ihr Body-Mass-Index ist: ", c, "\n")
print("Ihre Eingaben waren:")
print("Größe:", a, "m")
print("Gewicht:", b, "kg")
```

Ohne Python zu können, fällt Ihnen etwas auf?

Das Programm fragt Sie zunächst nach Gewicht und Größe. Wenn Sie etwa eine Größe von 1,78 m und ein Gewicht von 80 kg angeben, wird der Body-Mass-Index als 25,25 ausgegeben (ein Wert, der ganz leichtes Übergewicht anzeigt). Danach werden Ihnen nochmal die Ausgangswerte, mit denen Sie das Programm gefüttert haben, angezeigt; das sieht dann so aus:

```
Ihre Eingaben waren:  
Größe: 80 m  
Gewicht: 1.78 kg
```

Mit diesen Werten wären Sie deutlich untergewichtig, was bei einer Größe von 80 Metern aber mit Sicherheit ein noch vergleichsweise unbedeutendes Problem wäre!

Was ist passiert? Im Programm wurde bei der Ausgabe der Parameter Gewicht und Größe die Variablen-Namen vertauscht. Dadurch werden die falschen Werte ausgegeben. Da die Variablen hier **a** und **b** heißen, fallen solche Verwechslungen im Programmcode erst einmal gar nicht so schnell auf.

Viel einfacher ist der Fehler zu entdecken, wenn man *sprechende* Variablen-Namen wählt:

```
gewicht = input("Ihr Gewicht [in kg]: ")  
groesse = input("Ihre Größe [in m]: ")  
  
bmi = float(gewicht) /float(groesse)**2  
  
print("Ihr Body-Mass-Index ist: ", bmi, "\n")  
print("Ihre Eingaben waren:")  
print("Größe:", gewicht, "m")  
print("Gewicht:", groesse, "kg")
```

Verwenden Sie also möglichst gut verständliche, sprechende Bezeichner, die eine Idee davon vermitteln, was das Ding, das Sie benennen, enthält (im Fall einer Variablen) oder tut (im Fall einer Funktion). Gerade, wenn man schnell programmiert, ist die Versuchung groß, ohne langes Nachdenken, kurze, oft einbuchstabige Variablen-Namen zu verwenden (zum Beispiel **x**, **y**, **i**, **f**), weil sich die einfach schnell tippen lassen. Dieser Versuchung sollten Sie zu widerstehen versuchen. Ihr „späteres Ich“, das Ihren Programmcode zu lesen versucht, wird es Ihnen danken!

Manchmal ist es bei den Bezeichnern aber nicht mit einem Wort, wie hier **groesse** oder **gewicht**, getan. Gerade bei zusammengesetzten Bezeichnern sieht man Unterschiede in den Namenskonventionen von Programmiersprachen. Angenommen, wir wollten Größe und Gewicht danach unterscheiden, ob es sich beim Benutzer des Programms um einen Mann oder um eine Frau handelt, und führten dazu unterschiedliche Variablen für beide Geschlechter ein. Manche Programmiersprachen präferieren für die Größe des Mannes die Schreibweise **groesseMann**, andere **groesse\_mann**, wieder andere **groesse.mann**.

Natürlich ist es letztlich vollkommen egal, ob Sie der Konvention folgen, die die meisten Programmierer in Ihrer Programmiersprache beherzigen, oder nicht. Für die syntaktische Korrektheit und damit die Ausführbarkeit ihres Programms spielt das natürlich keine Rolle, zumindest, so lange Sie nur die in der Sprache zulässigen Zeichen in Variablen-Namen verwenden (in manchen Sprachen ist beispielsweise \$ ein reserviertes Zeichen, **wechselkurs** \$ wäre dann ein ungültiger Variablen-Name und würde zu einem Fehler führen). Es empfiehlt sich aber, sich an *irgendeinen* Standard zu halten. Das spart Denkarbeit, denn Sie müssen nicht jedes Mal neu überlegen, und macht Ihr Programm später lesbarer.

Über die Frage der Formatierung des Codes, der Wahl geeigneter Bezeichner und einiger anderer Themen in diesem Kontext haben viele Leute viele kluge Überlegungen angestellt. Google hat in seinen *Google Style Guides* diese Gedanken für eine ganze Reihe von Programmiersprachen zusammengetragen, und auch an vielen anderen Stellen im Internet werden Sie (mitunter auch von den Google-Richtlinien leicht abweichende) Style Guides finden. Schauen Sie sich ruhig einen solchen Style Guide für Ihre Programmiersprache an. Sklavisch daran halten müssen Sie sich natürlich nicht. Wichtiger als einem bestimmten Style zu folgen, ist es, überhaupt einen Style zu haben, den man einigermaßen konsequent umsetzt, um sich gedankliche Arbeit zu sparen und seinen Code lesbar und mithin verständlich zu halten.

## 10

### 10.3 Kommentare

#### 10.3.1 Den eigenen Programmcode erläutern

Nicht nur eine gute Formatierung und eine sinnvolle Wahl von Bezeichnern helfen dabei, Code zu schreiben, den Sie später noch verstehen. Ein weiteres wichtiges Hilfsmittel auf diesem Weg sind *Kommentare*.

Kommentare sind Texte im Programmcode, die vom Compiler bzw. Interpreter *ignoriert* werden. Anders als im Rest des Quelltextes, wo sie sich peinlich genau an die Syntax-Regeln halten müssen, damit der Computer versteht, was Sie wollen, können Sie in Kommentaren nach Herzenslust drauflosschreiben. Der Trick besteht darin, dem Compiler bzw. Interpreter klar zu machen, dass Ihr Kommentar nicht Bestandteil des eigentlichen Programmcodes ist, und er ihn daher getrost vernachlässigen kann.

Damit der Compiler oder Interpreter weiß, was Programmcode und was Kommentar ist, werden Kommentare immer mit einem speziellen Symbol eingeleitet. Je nach Sprache reicht der Kommentar dann entweder bis zum Zeilenende oder bis zu der Stelle, an der er wiederum mit einem speziellen Symbol abgeschlossen wird.

Manche Sprachen unterstützen nur den ersten Modus, was dazu führt, dass es ausschließlich einzeilige Kommentare gibt. Soll sich ein Kommentar über mehrere Zeilen erstrecken, muss der Kommentar in jeder Zeile wieder mit dem Kommentar-Symbol beginnen.

Schauen wir uns einige Beispiele an. Zunächst ein Beispiel in der Programmiersprache C:

```
// Zählerstand ist jetzt vollständigen hochgelaufen.  
// Zählerstand ausgeben  
printf("Zählerstand: ", zaehler, "\n");  
  
zaehler = 0; // Zähler neu initialisieren
```

Das Symbol, das Kommentare hier einleitet, ist **//**. Alles, was zwischen **//** und dem Zeilenende steht, ist Kommentar und wird vom C-Compiler ignoriert. Deshalb können Kommentar und Code auch in derselben Zeile stehen, wobei sich der Code natürlich links vom Kommentarsymbol **//** befinden muss. Interessanterweise kennt die Programmiersprache C und viele an sie angelehnte Sprachen nicht nur den einzeiligen Kommentar mit **//**, sondern auch (potentiell) mehrzeilige Kommentare. Diese werden zwischen die Symbole **/\*** und **\*/** eingeschlossen (die Sternchen sind also immer dem Kommentartext zugewandt). Damit ließe sich das Beispiel oben wie folgt schreiben:

```
/* Zählerstand ist jetzt vollständigen hochgelaufen.  
 * Zählerstand ausgeben */  
printf("Zählerstand: ", zaehler, "\n");  
  
zaehler = 0; /* Zähler neu initialisieren */
```

Wie Sie sehen, benötigt die zweite Zeile des ersten Kommentars kein separates Symbol, um den Kommentar zu beginnen, denn wir befinden uns immer noch in dem Kommentarbereich, der in der vorangegangenen Zeile durch **/\*** geöffnet worden ist. Erst mit **\*/** wird der Kommentar wieder geschlossen.

Da dieser Kommentar einen klar definierten Beginn und ein klar definiertes Ende hat, kann er auch mitten im Code stehen, was allerdings nicht zu empfehlen ist, da es der Lesbarkeit des Codes eher abträglich ist:

```
zaehler /* der Schleifenzähler */ = 0;
```

Jede Programmiersprache hat natürlich grundsätzlich ihre eigenen Symbole für Kommentare. Häufig anzutreffen sind aber das eben bereits genannte **//**, die Kombination aus öffnendem **/\*** und schließendem **\*/** sowie das Symbol **#** (meist für einzeilige Kommentare).

Die Verwendung von Kommentaren kann Ihnen helfen, die Funktionsweise Ihres Programms direkt im Code zu dokumentieren. Jeder, der den Code hat, sieht damit zugleich die Dokumentation. Das ist natürlich insbesondere dann interes-

sant, wenn Sie nicht alleine, sondern mit jemand anderem zusammenarbeiten. Ihr Mitstreiter wird den ein oder anderen Hinweis an besonders komplizierten Stellen mit Sicherheit sehr zu schätzen wissen.

Aber auch Sie selbst, zumindest Ihr „späteres Ich“, werden von den Kommentaren immens profitieren. Nichts ist schöner, als sich ein altes Stück Code anzuschauen und freudig überrascht festzustellen, dass der damalige Entwickler (Sie selbst) einen kleinen Hinweis hinterlassen hat, der es jetzt erheblich einfacher macht, die schwierige Passage zu verstehen.

Weil jedoch das Kommentieren Ihrer Lösung Zeit kostet, sollten Sie Kommentare nur dort einsetzen, wo es wirklich erforderlich ist, das heißt, an Stellen des Programms, von denen Sie erwarten können, dass Sie sie ohne Hilfestellung später nicht mehr ohne weiteres verstehen werden. Kommentieren Sie also nicht jede Trivialität, sondern setzen Sie Kommentare ökonomisch ein. Das Kommentieren ist wahrscheinlich der am wenigsten beliebte Teil der Programmierarbeit. Schreiben Sie Ihre Kommentare daher zeitnah, idealweise direkt, nachdem Sie den betreffenden Code geschrieben haben. Nicht nur haben Sie auf diese Weise die Funktionsweise Ihrer Lösung noch genau in Erinnerung, auch vermeiden Sie, sich später noch einmal in die Lösung hineindenken zu müssen, um den Kommentar zu schreiben, ein mentaler Aufwand, der nicht nur eingefleischte Anhänger des gepflegten Prokrastinierens möglicherweise davon abhält, die Kommentierung noch zu ergänzen.

## 10

### 10.3.2 Wozu Kommentare sonst noch nützlich sind

Mit Kommentaren können Sie natürlich nicht nur die Vorgehensweise, die Sie angewandt haben, dokumentieren, sondern auch unerledigte Aufgaben direkt an Ort und Stelle notieren, zum Beispiel:

```
# TODO Nachfolgenden Teil komplett in separate
# Funktion auslagern
```

Ein klares **TODO** markiert den Kommentar für Sie als offene Aufgabe. Genauso können Sie etwa mit **REVIEW** Code-Segmente hervorheben, die Sie sich noch einmal genauer anschauen wollen, oder aber auch beliebige andere aussagekräftige **Tags** vergeben, die spezielle Arten von Kommentaren markieren.

Kommentare können Sie aber auch viel profanter einsetzen, nämlich um Ihren Quellcode besser zu strukturieren und für Sie übersichtlicher zu machen. In diesem Sinne können Sie Kommentare nutzen, um Teile des Quellcode voneinander abzutrennen, zum Beispiel so (in C++):

```
/* ----- BEGINN EINLESEN DER DATEN AUS DATEI ----- */
```

Wenn Sie anfangen zu programmieren, werden Sie schnell sehen, wie hilfreich es manchmal sein kann, eine Programm-Anweisung *auszkommentieren*. Das bedeutet, diese Anweisung in einen Kommentar zu setzen und auf diese Weise „abschalten“. Denn der Compiler bzw. Interpreter führt ja das, was in einem Kommentar steht, nicht aus. Indem Sie also eine Anweisung von einem Kommentar umschließen, können Sie diese Anweisung vorübergehend unwirksam machen. Ein kleines Beispiel in C:

```
printf("Dieser Text wird auf dem Bildschirm ausgegeben!");  
  
/* printf("Dieser Text erscheint nicht, denn die printf-Anweisung ist auskommentiert"); */
```

Kommentare sind natürlich nicht zuletzt hilfreich, wenn man gerade damit beginnt, sich mit einer neuen Programmiersprache auseinanderzusetzen und die Erkenntnisse, die man beim Ausprobieren erlangt, gleich im Code der Testprogramme festhalten möchte. Bekanntlich lernt man am besten, wenn man sich Erlerntes notiert (die hierfür anerkanntermaßen effektivste Methode, das handschriftliche Notieren, scheidet bei Programmcode naturgemäß aus).

### 10.3.3 Dokumentation außerhalb des Programmcodes

Kommentare sind eine Art Dokumentation *im* Programmcode. Das ist vorteilhaft, stehen doch die Erläuterungen und das, was erläutert wird, direkt beieinander. Allerdings spricht natürlich grundsätzlich nichts dagegen, Dokumentation auch außerhalb des Quelltextes abzulegen. Das bietet sich zum Beispiel an, wenn Sie etwas umfassendere Erläuterungen schreiben wollen, oder andere Informationen als Fließtext (etwa eine Tabelle oder die Fotografie einer handschriftlichen Notiz oder Skizze) einbinden wollen. Ebenso, wenn Sie übergreifende Zusammenhänge dokumentieren wollen, etwa die Gesamtstruktur Ihres Programms oder das Ineinander greifen verschiedener Programmteile. In allen diesen Fällen ist es naheliegend, zumindest einen Teil der Dokumentation vom Code zu trennen und in ein anderes Dokument auszulagern. Ob Sie diese externe Dokumentation nun mit einer Textverarbeitung verfassen oder in Ihrem elektronischen Notizbuch festhalten, ist letztlich vollkommen unerheblich. Es gilt der Grundsatz: Gelobt sei, was dem Verständnis hilft!

Eine besondere Art Dokumentation außerhalb des Quellcodes zu generieren, sind spezielle *Dokumentationsgeneratoren*, die für viele Programmiersprachen verfügbar sind. In ihrer Grundfunktion tun diese Werkzeuge nicht viel mehr als bestimmte, im Programmcode besonders markierte Kommentare automatisch auszulesen und in einer optischen ansprechenden Form aufzubereiten. Das Ergebnis wird dann meist als HTML-Dokument oder im PDF-Format ausgegeben. Beispiele für solche Tools sind etwa *Javadoc* für Java, *phpDocumentor* für PHP, *roxygen* für R oder *Doxygen*, das eine ganze Reihe unterschiedlicher Sprachen unter-

stützt, darunter Python und C++. Dokumentation auf diese Weise zu generieren ist insbesondere dann interessant, wenn man Programmcode schreibt, der später von anderen aufgerufen werden soll (wir werden weiter unten sehen, wie man Programmteile in modulare Pakete verpacken kann, die dann von anderer Stelle aus aufgerufen werden können). Diejenigen, die den Programmcode aufrufen sollen, müssen zwar nicht notwendigerweise im Detail verstehen, wie der Code funktioniert, aber sie müssen wissen, wie genau sie den Programmteil aufrufen können und wie sie dessen Arbeitsweise für ihre Zwecke steuern können. Da sich diese Entwickler ja nicht für den Programmcode als solchen interessieren, sondern nur für dessen *Schnittstelle*, also die Möglichkeit, ihn von außen aufzurufen, wäre es unpraktisch, wenn sie den Code durchstöbern müssten, um die entsprechenden Hinweise zu finden. Deshalb wird die Dokumentation ausgelagert. Auf diese Weise bleibt sie „sauber“ und übersichtlich und ist für den Anwender leicht zu benutzen. Für den Entwickler des Codes hingegen ist sie leicht zu schreiben, weil sie wie herkömmliche Kommentare auch direkt in den Programmcode eingebettet ist. Das könnte dann zum Beispiel in Java so aussehen:

```
10
/*
 * Liest den Namen des Benutzers ein
 *
 * @author Pogra Miehrer
 * @version 1.3
 *
 * @param prompt Eingabeaufforderung, die der Benutzer
 *               präsentiert bekommt
 * @return Eingelesener Name des Benutzers
 */

public String getUserName(prompt) {
    /* Hier steht der eigentliche Programmcode */
}
```

Hier wird ein spezieller Programmteil, eine sogenannte Funktion, mit *Javadoc* dokumentiert. Die Funktion kann mit einer Eingabeaufforderung aufgerufen werden, zum Beispiel `getUserName(„Gib Deinen Namen ein.“)`, liest dann eine Benutzereingabe ein und gibt demjenigen, der die Funktion aufgerufen hat, die Eingabe des Benutzers als sogenannten Rückgabewert zurück. Mit Funktionen beschäftigen wir uns an späterer Stelle eingehender. Wichtig hier ist lediglich, dass diese Funktion ein Programmteil ist, der von anderen Programmen aufgerufen werden kann. Damit man versteht, was die Funktion braucht, um aufgerufen zu werden (nämlich den Text der Eingabeaufforderung, die dem Benutzer angezeigt werden soll) und was sie zurückliefert (nämlich den eingelesenen Namen des Benutzers), wird die Funktion hier so dokumentiert, dass das Dokumentationswerkzeug Javadoc automatisch eine ansprechende HTML-Dokumentation erzeugen kann. Dazu verwendet Javadoc besondere Kommentare.

Normale Kommentare in Java stehen zwischen den Symbolen `/*` und `*/`. Wird ein Kommentar dagegen mit `/**` eingeleitet, weiß Javadoc, dass es sich hierbei um

einen Kommentar handelt, der Bestandteil der Dokumentation werden soll. Mit speziellen, vordefinierten Tags, die mit @ eingeleitet werden, können bestimmte Felder in der Dokumentation angesprochen werden. So wird zum Beispiel mit **@ param prompt** erläutert, was der Parameter **prompt** bedeutet, mit dem man die Funktion **getUserName** aufruft. Diese vordefinierten Felder können dann in der Dokumentation besonders dargestellt werden, etwa in einer speziellen Formatierung oder an einer besonderen Position innerhalb der Dokumentation.

Oft wird zwischen Dokumentation und Kommentierung strenger unterschieden, als wir es hier tun, und zwar anhand der Zielgruppe: Kommentare richten sich an denjenigen, der Ihren Code verstehen und bearbeiten will (regelmäßig Sie selbst, gegebenenfalls aber auch andere). Ein anderer Entwickler, der Ihren Code hingegen einfach nur in eigenen Programmen *einsetzen* möchte, sich aber nicht für seine innere Struktur und Funktionsweise interessiert, wird Ihre Kommentare im Code gar nicht lesen, sondern nur nach Informationen darüber suchen, wie genau er Ihren Code richtig verwendet; ähnlich wie auch der Benutzer einer Textverarbeitung sich nicht dafür interessiert, wie sie genau im Inneren arbeitet, er will wissen, wie für einen markierten Text die Schriftfarbe ändert! Die Bereitstellung genau dieser Informationen zur *Verwendung* des Codes leistet die Dokumentation im engeren Sinne, die natürlich via Dokumentationsgeneratoren aus speziellen Kommentaren im Code erzeugt sein mag.

## 10.4 Ihr Fahrplan zum Erlernen einer neuen Programmiersprache

---

- Wenn Sie eine neue Programmiersprache lernen ...
  - stellen Sie fest, ob Sie in der Programmiersprache Ihren Code beliebig formatieren können (zum Beispiel mit Einrückungen), oder ob die Formatierung eine inhaltliche Bedeutung hat,
  - schauen Sie sich einen gängigen Style-Guide an, der Auskunft darüber gibt, wie man seinen Code in der Programmiersprache formatieren und wie man die Bezeichner/Namen von Variablen, Funktionen und anderen Elementen wählen *sollte*,
  - entscheiden Sie sich für eine Art, Ihren Code zu formatieren und die Variablen, Funktionen und andere Elemente zu benennen; dabei müssen Sie nicht zwingend einem Style Guide folgen, aber Sie sollten für sich konsistent bleiben,
  - stellen Sie fest, wie Kommentare in den Code verbaut werden können, insbesondere, ob Kommentare mehrzeilig sein können, und ob sie vorne und hinten begrenzt werden, oder immer bis zum Ende der Zeile reichen,
  - gewöhnen Sie sich an, Ihren Code zu kommentieren, um ihn später noch zu verstehen, gehen dabei aber ökonomisch vor und konzentrieren Sie sich auf die schwierigen Stellen, bei denen tatsächlich die Gefahr besteht, dass Sie sie später nicht mehr oder nur noch unter großen Mühen verstehen werden,
  - stellen Sie fest, ob es Dokumentationsgeneratoren gibt, die es erlauben, aus der in den Programmcode integrierten Kommentierung eine ansprechende HTML- oder PDF-Dokumentation zu erzeugen.

# Wie speichere ich Daten, um mit ihnen zu arbeiten?

## Inhaltsverzeichnis

- 11.1 Variablen als Platzhalter für Daten – 89**
- 11.2 Datentypen von Variablen – 90**
  - 11.2.1 Unterschiedliche Arten von Daten erfordern unterschiedliche Arten von Variablen – 90
  - 11.2.2 Wichtige Datentypen – 91
  - 11.2.3 Den Datentyp ändern: Konvertierung von Variablen – 96
- 11.3 Variablen anlegen und initialisieren – 98**
- 11.4 Gar nicht so variabel: Konstanten – 101**
- 11.5 Geordnete Felder von Variablen/Arrays – 101**
- 11.6 Assoziative Felder von Variablen/Hashes – 105**
- 11.7 Objekte – 106**

- 11.7.1 Eine Welt aus lauter Objekten – 107
  - 11.7.2 Klassen – 108
  - 11.7.3 Vererbung – 110
  - 11.7.4 Methoden – 113
  - 11.7.5 Polymorphismus – 116
  - 11.7.6 Zugriffsrechte – 118
- 11.8 Ihr Fahrplan zum  
Erlernen einer neuen  
Programmiersprache – 120**
- 11.9 Lösungen zu den Aufga-  
ben – 120**

## Übersicht

Programme wären langweilig und wenig nützlich, wenn sie nicht in der Lage wären, Eingaben vom Benutzer entgegenzunehmen. Manche Daten gibt er direkt ein, andere werden aus Dateien ausgelesen oder aus einer Datenbank abgefragt. Sogar der Klick auf einen Button ist eine Form der Dateneingabe.

Wo auch immer die Daten herkommen, die ein Programm verarbeiten soll, sie müssen im Speicher des Computers auf eine Weise abgelegt werden, die es dem Programm erlaubt, jederzeit schnell darauf zuzugreifen. Dies geschieht aus Sicht des Programmierers in Form von Variablen. Auch wenn diese Variablen durchaus einiges mit ihren Namensvettern aus der Mathematik zu tun haben, brauchen Sie keine Angst zu haben: Um dieses Kapitel verstehen zu können, müssen Sie sich nicht an die Kettenregeln beim Ableiten von Funktionen aus dem Mathematikunterricht erinnern!

In diesem Kapitel werden Sie folgendes lernen:

- was Variablen genau sind und wie man sie anlegt
- welche Arten von Daten Variablen aufnehmen können
- wie Sie viele gleichartige Variablen geschickt zu Variablen-Feldern zusammenfassen (Arrays und Hashes)
- wie Sie unterschiedliche Variablen, die die Eigenschaften eines bestimmten realen Gegenstands (z.B. eines Autos mit den Eigenschaften Marke, maximale Geschwindigkeit und Listenpreis) wiedergeben, zu einem Objekt zusammenfassen und wie Sie diese Eigenschaften dann bearbeiten (objektorientiertes Programmier-Paradigma).

## 11.1 Variablen als Platzhalter für Daten

Variablen sind Hilfsmittel, mit denen wir in Programmen Daten aufnehmen. Wie in der Mathematik auch, fungieren Variablen als *Platzhalter*, die wir über ihren Namen, im Programmierer-Jargon auch gerne *Bezeichner* genannt, ansprechen können. Ihr Inhalt ist, wie der Name schon suggeriert, variabel, wir können also (nacheinander) durchaus unterschiedliche Daten in der Variablen ablegen, aber ganz gleich, was der aktuelle Inhalt der Variable auch gerade sein mag, er ist auf jeden Fall über den Namen der Variable ansprechbar.

Eine Variable ist in etwa vergleichbar mit einer Box, die wir mit einem Etikett beschriften. Die Beschriftung, also der Name oder Bezeichner, bleibt immer gleich, auch wenn wir den Inhalt der Box austauschen. Natürlich sollte der Inhalt der Box dabei einigermaßen zur Beschriftung passen, ansonsten entsteht Verwirrung. Genauso ist das mit Variablen auch.

Bei der Wahl des Namens der Variable sind wir im Allgemeinen recht frei. Wir Sie bereits im vorangegangenen Kapitel gesehen haben, gibt es aber einige sprachspezifische Grundregeln, die zu beachten sind. Diese Regeln schreiben meist vor, welche Zeichen im Namen einer Variablen vorkommen dürfen (bestimmte Sonderzeichen wie % oder # sind meist unzulässig). Darüber hinaus ist in vielen Programmiersprachen auch festgelegt welche Zeichen am Anfang eines Variab-

len-Namens stehen dürfen oder sogar müssen. In R beispielsweise dürfen Variablen-Namen nicht mit einer Ziffer beginnen, in PHP müssen Variablen-Namen mit dem Dollarzeichen (\$) beginnen. Wichtig ist aber, wie Sie sich ebenfalls aus dem letzten Kapitel erinnern werden, nicht nur die Einhaltung dieser „hartten“ Regeln, deren Verletzung unweigerlich dazu führen wird, dass Ihnen der Compiler oder Interpreter die Gefolgschaft verweigert, sondern auch, dass Sie Variablen *sinnvoll* und *konsistent* benennen. Sinnvoll bedeutet in diesem Zusammenhang, dass Sie aus dem Namen den Inhalt oder Zweck der Variable erkennen können. Konsistent bedeutet, dass Sie idealerweise immer Variablen-Namen immer in der gleichen Weise bilden, also zum Beispiel die Teile von zusammengesetzten Variablen-Name immer auf dieselbe Weise gross und klein schreiben (sofern Ihre Sprache zwischen Groß- und Kleinschreibung unterscheidet) und auf dieselbe Weise voneinander abgrenzen. Nehmen wir an, Sie entwickeln einen Webshop und wollen die Anzahl der Bestellungen eines Kunden in den letzten 12 Monaten in einer Variablen ablegen. Diese Variable könnte zum Beispiel **AnzahlBestellung12Monate** heißen, aber auch **AnzBest12M** oder **anzBest12M** oder **Anzahl\_Bestellungen\_12\_Monate** oder ... Die Möglichkeiten sind vielfältig, der Kreativität nur wenige Grenzen gesetzt. Für die Lesbarkeit und Verständlichkeit des Programmcodes ist es wichtig, dass Sie sich eine Systematik überlegen, wie Sie Variablen-Namen bilden wollen, und diese Systematik dann möglichst konsistent durchziehen.

## 11.2 Datentypen von Variablen

### 11

#### 11.2.1 Unterschiedliche Arten von Daten erfordern unterschiedliche Arten von Variablen

Bisher haben wir gar nicht darüber gesprochen, welche *Art von Informationen* eine Variable überhaupt aufnehmen kann, also etwa eine Zahl oder einen Text. Für den Compiler oder Interpreter Ihrer Programmiersprache macht das aber durchaus einen Unterschied, und zwar aus mindestens zwei Gründen.

Zum einen nämlich muss der Compiler oder Interpreter Ihrer Sprache Speicher für die Variable reservieren. Es ist einleuchtend, dass ein langer Text (zum Beispiel ein Straßename) mehr Speicher benötigt als eine Zahl (zum Beispiel eine Hausnummer). Wenn Sie einer Variablen zunächst eine Zahl, sagen wir die Hausnummer 58, zuweisen und Ihr Compiler oder Interpreter daraufhin dafür sorgt, dass so viel Speicherplatz reserviert wird, wie eben das Speichern einer Zahl benötigt, und Sie dann derselben Variablen plötzlich aber doch einen längeren Text, etwa die Anschrift „Ernst-Reuter-Platz“ zuweisen, reicht der zunächst reservierte Platz nicht mehr aus. Es muss neuer Platz im Speicher, gesucht werden, möglicherweise an einer ganz anderen Stelle.

Zum anderen können mit Zahlen und Texten sehr unterschiedliche Operationen durchgeführt werden. Eine Zahl zum Beispiel können Sie mit einer anderen Zahl multiplizieren. Bei einem Text macht diese Rechenoperationen keinen Sinn. Schlimmstenfalls stürzt das Programm sogar ab, wenn Sie eine Operation

## 11.2 · Datentypen von Variablen

durchführen, die für die Art von Information, die Ihre Variable enthält, gar nicht zulässig ist. Deswegen macht es Sinn, bei Zeiten zu prüfen, ob die Information in einer Variablen von der „richtigen“ Art ist, den richtigen *Datentyp* hat, wie man im Programmierjargon sagt.

Der Datentyp einer Variablen beschreibt, welche Art von Informationen in ihr gespeichert werden können. Insofern gibt ein Datentyp zu Beispiel an, ob eine Variable ganze Zahlen, gebrochene Zahlen oder Texte aufnehmen soll. Datentypen unterscheiden sich in der Praxis aber nicht nur darin, welche *Art* von Informationen sie abdecken. Sie unterscheiden sich auch in Hinblick auf den *Wertebereich* bzw. *die Länge der Information*, die sie aufnehmen können: Eine Variable, die für einen Text (also eine Zeichenkette) gedacht ist, und dafür 10 Zeichen zur Verfügung stellt, wird eben „Peter Müller“ als „Peter Müll“ speichern (das Leerzeichen zählt hier natürlich auch als Zeichen!). In einer Ganzzahl-Variable, deren Wertebereich von 0 bis 65.535 reicht (das ist ein Wertebereich, den man mit zwei Byte abdecken kann), werden Sie einen negativen Kontostand von –254 EUR nicht speichern können. Genauso wird eine Ganzzahlvariable mit dem (ebenfalls zwei Byte Speicher benötigenden) Wertebereich von –32.768 bis 32.767 mit einem Kontostand von 50.000 EUR nicht umgehen können. Datentypen haben also unterschiedliche Wertebereiche, die – genauso wie die grundsätzliche Art von Information, für die sie vorgesehen sind – die Daten beschränkt, die sie aufnehmen können. Bei Datentypen für Fließkommazahlen, das heißt, gebrochenen Dezimalzahlen wie etwa 3,1415926, tritt mit der *Genauigkeit*, also der Zahl von Nachkommastellen, ein weiteres Merkmal hinzu. Genauigkeit spielt eine Rolle. Ob man bei einer Ski- oder Eisschnelllauf-Weltmeisterschaft zusammen mit einem Konkurrenten die Goldmedaille in Empfang nimmt, oder es hinter diesem Konkurrenten doch „nur“ auf Platz zwei schafft, kann durchaus davon abhängen, ob das Ergebnis in Hundertstel-Sekunden gemessen wird (und beide Sportler dasselbe Ergebnisse aufweisen und daher beide zu Siegern erklärt werden würden) oder ob die Tausendstel-Sekunden mit herangezogen werden, und sich dann doch ein minimaler Unterschied zwischen den beiden Zeiten zeigt.

### 11.2.2 Wichtige Datentypen

Die grundlegenden Datentypen sind in den meisten Programmiersprachen recht ähnlich. Regelmäßig gibt es Datentypen für:

- **Ganzzahlen**

Diese Datentypen nehmen ganzen Zahlen wie –4, –3, –2, 0, 1, 2, 3, 4 auf. Wahlweise gibt es Datentypen, die nur positive ganze Zahlen (inklusive der Null), das heißt, natürliche Zahlen, aufnehmen. Ganzzahlige Datentypen haben in vielen Programmiersprachen einen Namen der das Wort *integer*, den englischen Begriff für Ganzzahl, oder eine Abkürzung davon beinhaltet. Klassische Namen solcher Datentypen sind daher etwa **integer** oder **int**, für Ganzzahl-Datentypen mit besonders großem Wertebereich, also der Möglichkeit, sehr große Zahlen speichern zu können, gerne auch **bignum**, **long int** oder einfach nur **long**.

## ■ Fließkommazahlen

Datentypen für Fließkommazahlen nehmen gebrochene Dezimalzahlen auf, wie zum Beispiel 1,7 oder 3,141459. Natürlich lassen sich auch die Ganzzahlen als Fließkommazahlen darstellen, so etwa die 4 als 4,0. Dementsprechend können alle Ganzzahlen auch in Fließkommazahl-Variablen gespeichert werden.

Wenn das so ist, stellt sich natürlich die Frage, warum man dann überhaupt Ganzzahlvariablen benötigt und nicht einfach durchgängig immer mit Fließkommavariablen arbeitet. Der Grund liegt vor allem in dem größeren Speicherbedarf der Fließkommazahlen, denn diese müssen ja neben den Stellen „vor dem Komma“ auch den gebrochenen Nachkommateil speichern, bzw. Kapazität für dessen Speicherung vorhalten, selbst wenn der Nachkommanteil bei einer ganzzahligen Größe immer Null ist. Das Problem ist eben nur, dass unser Compiler bzw. Interpreter ja im Vorhinein nicht weiß, dass wir nur ganzzahlige Werte zu speichern vorhaben. Er reserviert deshalb immer so viel Speicher, wie es für die Speicherung einer gebrochenen Dezimalzahl eben braucht. Wenn man sehr viele solcher Fließkomma-Variablen auf einmal benötigt, schlägt dadurch ein merklich höherer Speicherbedarf zu Buche.

Für Fließkomma-Datentypen gibt es in unterschiedlichen Programmiersprachen eine ganze Reihe von Namen, wie etwa **real** (für reelle Zahlen) oder **float** (für Fließkomma). In den meisten Sprachen gibt es – analog zu den **integers** und **long integers** – noch einen weiteren Fließkomma-Datentyp, der höhere Präzision (das heißt, mehr Nachkommastellen) und einen größeren Wertebereich bietet. Meist deutet hier bereits Namen wie **double** oder gar **long double** auf die höhere Genauigkeit hin.

## 11

## ■ Zeichen und Zeichenketten

Einzelne Zeichen, also etwa Buchstaben, sind Ganzzahlen nicht unähnlich, denn jedes Zeichen, dass der Computer versteht, das also Bestandteil seines *Zeichensatzes* ist, lässt sich auch als Zahl codieren. Bekannte Zeichensätze sind etwa ASCII oder Unicode. Obwohl es also einen engen Zusammenhang zwischen Zeichen und Zahlen gibt, kennen die meisten Programmiersprachen einen speziellen Datentyp für einzelne Zeichen, oft mit einem Namen der an den englischen Begriff für Zeichen, *character*, angelehnt ist. Dementsprechend gibt es in vielen Sprachen einen Zeichtyp **char** für einzelnen Zeichen.

Ganze Texte sind letztlich Aneinanderreihungen von Zeichen, also Zeichenketten (engl. *strings*). Manche Programmiersprachen kennen daher für Zeichenketten gar keinen eigenen Typ, sondern bauen Zeichenketten aus einer Aneinanderreihung einzelner Character-Variablen zusammen. Hier wird also aus einem einfachen Datentyp ein komplexerer Datentyp geschaffen in dem viele Variablen des einfachen Datentyps praktisch hintereinandergeschaltet werden. Diese Art von Hintereinanderschaltung wird auch als *Feld* (engl. *array*) bezeichnet, eine Konstruktion, mit der wir uns später noch etwas intensiver befassen werden.

Andere Sprachen besitzen für Zeichenketten einen eigenen Datentyp, der oft nach dem englischen Wort für Zeichenkette **string** heißt. Dadurch wird der Charakter der Zeichenkette, dass sie nämlich aus unterschiedlichen aneinandergereihten Zeichen besteht, vom Programmierer etwas abgeschirmt. Für ihn „fühlt“ es

## 11.2 · Datentypen von Variablen

sich dann an, als sei der Text eine einzige Variable und nicht zusammengesetzt aus Einzelvariablen, die in einem Feld angeordnet sind.

Zeichenketten können aber oftmals nicht nur Buchstaben, Ziffern und Sonderzeichen (wie etwa Satzzeichen, #, <, >, \*, ~) enthalten, sondern auch sogenannte Escape-Sequenzen. Dabei handelt es sich um spezielle Steueranweisungen. Die in der Praxis wohl wichtigste markiert einen Zeilenumbruch und wird als **\n** dargestellt. Durch den Backslash (\) weiß der Interpreter bzw. Compiler, dass das nachfolgende Zeichen nicht als Buchstabe, sondern als Steueranweisung zu verstehen ist; das **n** selbst steht für *new line*, also neue Zeile. Damit würde also der String "**Herr/Frau\nVorname Name\nStraße Hausnummer\nPLZ Stadt**" interpretiert werden als

```
Herr/Frau
Vorname Name
Straße Hausnummer
PLZ Stadt
```

Überall da, wo die Escape-Sequenz **\n** anzutreffen ist, wird ein Zeilenumbruch eingefügt. Während also innerhalb der Zeichenkette die Escape-Sequenz die Stelle des Zeilenumbruchs markiert, „verstehen“ bestimmte Programmfunctionen – etwa die zur Ausgabe von Strings auf dem Bildschirm – die Codierung und setzen sie entsprechend um. Neben **\n** existieren noch eine ganze Reihe weiterer Escape-Sequenzen, zum Beispiel **\t** für einen Tabulator-Sprung. Escape-Sequenzen lösen noch ein weiteres Problem das häufig auftritt: In den meisten Programmiersprachen werden Zeichenketten von einfachen ('') oder doppelten ("") Anführungszeichen eingeschlossen. Was aber ist, wenn eine Zeichenkette zum Beispiel ein Zitat beinhaltet soll, wie die Anführungszeichen also innerhalb der Zeichenkette als *echte Zeichen* benötigen? Betrachten Sie die folgende Zeichenkette:

```
"Er sagte: "Ich liebe Dich!""
```

Sie ist in doppelte Anführungszeichen eingefasst, enthält aber ein Zitat, das selbst doppelte Anführungszeichen verwendet. Normalerweise würde uns der Interpreter bzw. Compiler hier die Gefolgschaft verweigern. Er würde hier zunächst eine Zeichenkette "**Er sagte: "** erkennen. Darauf folgt aber (ohne dass dies nun als Zeichenkette verstanden würde) **Ich liebe Dich!**, gefolgt von einer leeren Zeichenkette (""). Insbesondere das **Ich liebe Dich!** Würde vermutlich Probleme bereiten, weil es keine gültige Anweisung in der jeweiligen Programmiersprache sein wird, aber außerhalb einer Zeichenkette als Anweisung zu interpretieren versucht würde. Was also tun? Abgesehen von der trivialen Lösung, schlicht einfache Anführungszeichen für das Zitat innerhalb der Zeichenkette zu verwenden, können die „inneren“ doppelten Anführungszeichen auch einfach maskiert (*escaped*) werden:

```
"Er sagte: \"Ich liebe Dich!\""
```

Durch die Backslashes wird dem Interpreter bzw. Compiler mitgeteilt, dass das folgende Anführungszeichen als *Teil des Strings*, und nicht als seine Begrenzung verstanden werden soll.

Was aber, wenn im String ein Backslash vorkommen soll? Zum Beispiel in diesem String:

```
"Auch \n ist eine Escape-Sequenz."
```

Die Escape-Sequenz `\n` würde bei der Ausgabe normalerweise zu einem hier unerwünschten Zeilenumbruch innerhalb des Strings führen. Auch hier hilft Maskieren durch den Backslash, nur, dass eben dieses Mal der vorhandene Backslash selbst maskiert wird, weil er als Teil des Texts verstanden werden soll und nicht als Beginn einer Steueranweisung. Also:

```
"Auch \\n ist eine Escape-Sequenz."
```

Diese Zeichenkette würde in der Ausgabe genau zu dem Text im Beispiel führen. Das Escapen des Backslashes selbst ist insbesondere wichtig, wenn Sie Pfadangaben auf Windows-Systemen codieren, zum Beispiel `C:\\Home\\Meine Quellcodes`. Eine häufige Fehlerursache ist, dass das Escapen hier vergessen wird.

Escape-Sequenzen kommen in vielen Programmiersprachen zum Einsatz, zum Beispiel in C, Python, Perl oder R.

## 11

### ■ Wahrheitswerte

Praktisch alle Programmiersprachen besitzen einen speziellen Datentyp für den Wahrheitsgehalt von Aussagen, also *wahr* oder *falsch*. Häufig wird in diesem Zusammenhang auch von *boole'schen* (nach dem englischen Mathematiker und Logiker des 19. Jahrhunderts George Boole) oder *logischen* Variablen gesprochen. Anders als die Datentypen, die wir bisher betrachtet haben, gestatten boole'sche Variablen die Speicherung von nur zwei unterschiedlichen Werten, wahr und falsch. Das ist gänzlich anders als etwa bei den Ganzzahl-Variablen, die ja *jede beliebige* ganze Zahl aufnehmen können. In den meisten Programmiersprachen haben die beiden Wahrheitswerte spezielle Bezeichner, meist **true** (wahr) und **false** (falsch), sodass man boole'schen Variablen auf einfache Art und Weise Werte zuweisen oder Vergleiche mit ihnen anstellen kann und diese Zuweisungen und Vergleiche im Programmcode gut lesbar sind. Denn es die gut verständlichen Bezeichner **true** und **false** gibt, speichern die Programmiersprachen in der Regel nur **1** und **0** als Werte der boole'schen Variablen. In diesem Sinne sind boole'sche Variablen meist nur Ganzzahl-Variablen, bei denen der Compiler bzw. Interpreter darauf achtet, dass sie nur einen von zwei möglichen Ausprägungen haben, auf die praktischerweise mit speziellen Bezeichnungen, meist eben **true** und **false**, zugegriffen werden kann.

In Programmiersprache heißen die boole'schen Datentypen oft **bool**, **boolean** oder **logical**.

Diese Datentypen finden sich, durchaus mit unterschiedlichen Namen, unterschiedlichen Wertebereichen und unterschiedlicher Genauigkeit in praktisch allen modernen Hochsprachen. Daneben kennen die meisten Programmiersprachen noch eine ganze Reihe weiterer, komplexerer Datentypen, die oft auf den einfachen Typen, die Sie soeben kennengelernt haben, basieren.

Manche Sprachen haben zum Beispiel (mindestens) einen speziellen Datentyp für das Datum bzw. die Uhrzeit. Wenn man das Datum und die Uhrzeit speichern will, führt, so scheint es, kein Weg daran vorbei, Tage, Monate, Jahr, Stunden, Minuten und Sekunden einfach als Ganzzahlen zu speichern und dann irgendwie zu einem komplexen Datentyp „zusammenzubauen“. Das ist in der Tat eine Lösungsmöglichkeit, die oft angewendet wird. Es gibt aber noch andere: Die Zeit auf Systemen, die mit dem aus den 1970er-Jahren stammenden Betriebssystem UNIX arbeiten, wurde nämlich mit nur einer einzigen Ganzzahl gemessen. Als die Zahl der Sekunden, die seit dem 01. Januar 1970 (in Greenwich Mean Time, GMT) vergangen sind. Dann entsprach beispielsweise der Jahrtausendwechsel in Deutschland, also der 01. Januar 2000, 0:00 Uhr der Unix-Zeit 946.681.200. Das lässt sich einfach nachrechnen (wundern Sie sich dabei nicht, dass 946.681.200 scheinbar 3.600 Sekunden, also eine Stunde, „zu wenig“ hat; das liegt daran, dass Länder mit GMT gegenüber Deutschland mit mitteleuropäischer Zeit um eine Stunde zurück sind, vom letzten Tag des Jahres dort also erst 23 Stunden vergangen waren, als in Deutschland bereits das Millennium gefeiert wurde). Das Datum kann also auch in einer einzigen Zahl dargestellt werden, die aber in vielen Programmiersprachen als eigener Datentyp „verkauft“ wird, obwohl es letztlich eine Ganzzahl ist.

An diesem Beispiel lässt sich sehr schön auch die Rolle des Wertebereichs bei Datentypen verdeutlichen. Systeme, die auf UNIX laufen, hatten kein Jahr-Zweitausend-Problem. Allerdings werden diese dafür am 19. Januar 2038 um 3:14:08 Uhr Schwierigkeiten bekommen. Denn dann wird der Datentyp, der bei UNIX (zumindest auf den älteren Systemen) für die Speicherung der Uhrzeit verwendet wird, den Wert 2.147.483.647 und damit die Grenze seines Wertebereichs erreichen. Größere Zahlen kann dieser Datentyp nicht speichern. Was passiert dann eine Sekunde später, um 3:14:09? Der in Sekunden gemessene Datumswert wird wieder zurückspringen und zwar auf seinen kleinsten möglichen Wert, -2.147.483.648. Das entspricht nach Unix-Zeitrechnung einem Zeitpunkt im Dezember 1913. Die Entwickler von Unix haben das seinerseits natürlich gewusst, aber in Kauf genommen, das Jahr 2038 war aus Perspektive der 1970er-Jahre schließlich noch weit entfernt und Datentypen mit größerem Wertebereich standen nicht zur Verfügung.

Das Beispiel zeigt aber sehr schön, dass man sich sehr wohl Gedanken darüber machen muss, ob der Datentyp, den man zu verwenden beabsichtigt, für den angeachten Zweck tatsächlich ausreicht, oder ob man nicht einen Datentyp mit größerem Wertebereich benötigt (wenn man denn, anders als die UNIX-Entwickler, einen zur Verfügung hat).

Das Datum ist also ein Beispiel für einen Datentyp, der in vielen Sprachen neben den oben besprochenen, einfachen Datentypen existiert.

Daneben gibt es oftmals auch weitere Datentypen, etwa für Aufzählungen. Damals lassen sich kategoriale Daten speichern, Daten also, die nur bestimmte Aus-

prägungen annehmen können, zum Beispiel das Geschlecht einer Person, ihr höchster Schulabschluss oder eine Autofarbe. In diesem Sinne ähneln sie den boole'schen Variablen, nur dass die Zahl der Ausprägungen durchaus mehr als zwei sein kann. Häufig sind solche Aufzählungen (*enumerations*, *sets* oder *factors*) besondere Datentypen in einer Programmiersprache auch wenn sie „unter der Haube“ im Wesentlichen als Ganzzahlen (jede Kategorie/Ausprägung ist durch eine bestimmte Zahl definiert) gespeichert werden.

Datumswerte, Zeichenketten und Aufzählungen sind allesamt Beispiele für Datentypen, die auf einfacheren Datentypen basieren. Mit dem Feld, also der Aneinanderreihung von Variablen *des gleichen Typs*, hatten Sie im Zusammenhang mit Zeichenketten bereits eine Möglichkeit kennengelernt, Variablen zu einem komplexeren Datentyp zusammenzusetzen. Neben dieser werden wir uns später noch eine andere Möglichkeit genauer ansehen, deren Kerngedanke es, Variablen *unterschiedlicher Typen* zu einem Objekt zusammenzusetzen. Ein solches Objekt bildet dann oft einen realen Gegenstand mit seinen zentralen Eigenschaften ab, zum Beispiel ein Auto, das sich u.a. beschreiben lässt durch sein Alter in Jahren (Ganzzahl), seine Typbezeichnung (Zeichenkette) und seine Farbe (Aufzählung).

### 11.2.3 Den Datentyp ändern: Konvertierung von Variablen

Manchmal muss man den Datentyp von Variablen ändern. Man spricht auch davon, die Variable zu *konvertieren*.

11

Nehmen Sie beispielsweise an, Sie hätten vom Benutzer Ihres Programms, der Software für einen Online-Shop für Bücher, die Information eingelesen, welche Stückzahl er von welchem Buch kaufen will (meistens nur eins, manchmal aber vielleicht auch zwei, eines für ihn selbst, eines als Geschenk). Auf der Check-out-Seite, also dem letzten Schritt im Bestellprozess wollen Sie dem Benutzer anzeigen, wie viele Bücher er insgesamt bestellt. Das ist einfach, wenn Sie die Stückzahlen in Ganzzahl-Variablen (Integer-Variablen) eingelesen haben. Problematisch aber wird es, wenn Sie Zeichenketten-Variablen (String-Variablen) verwendet haben. Damit wird Ihre Addition nicht gelingen. Warum ist der Unterschied zwischen den beiden Variablen so wichtig?

Datentypen sind, wie Sie in den vorangegangenen Unterabschnitten gesehen haben, definiert durch die Art der Informationen, die Variablen dieses Typs aufnehmen können; weiterhin durch den Wertebereich und ggf. auch durch die Genauigkeit, mit der (Fließkomma-)Werte gespeichert werden. Es gibt aber noch ein weiteres Merkmal, das Datentypen unterscheidet: Die Frage, welche Arten von Operationen mit Variablen eines Datentyps durchgeführt werden können. Das man Ganzzahlen addieren kann, liegt auf der Hand. Aber wie sieht das mit Zeichenketten aus? Kann man die beiden Zeichenketten „Äpfel“ und „Birnen“ im mathematischen Sinne addieren? Nein, natürlich nicht, das leuchtet auch sofort ein. Wie sieht es aber mit den Zeichenketten „2“ und „1“ aus? Das könnten die bestellten Stückzahlen zweier Bücher aus unserem Online-Shop-Beispiel sein. Lassen die diese beiden Zeichenketten sich addieren? Die Antwort lautet, möglicherweise zu Ihrer Überraschung: Nein.

Den Compiler bzw. Interpreter, der Ihren Programmcode verarbeitet, interessiert es schlichtweg nicht, was genau inhaltlich in den Variablen enthalten ist. Es ist einfach eine Aneinanderreihung von Zeichen, für den Computer vollkommen ohne Bedeutung. Ob in der Zeichenkette Buchstaben, Zahlen oder irgendwelche Sonderzeichen wie das Dollarzeichen oder der Unterstrich enthalten sind, spielt keine Rolle. Er interessiert sich für den Inhalt nicht. Deshalb ist die Addition keine Operation, die für Zeichenketten zulässig ist. Wenn Sie nun aber wissen, dass in den String-Variablen, die Sie über die Website ihres Online-Shops eingelesen haben, definitiv Zahlen enthalten sind, wollen Sie natürlich trotzdem mit diesen rechnen. Was tun?

Der Schlüssel zur Lösung besteht darin, den Typ der Variablen zu ändern. Viele Programmiersprachen stellen besondere Anweisungen bereit, mit denen Sie genau das tun und eine *explizite Typumwandlung* vornehmen können. Manche Sprachen kennen aber auf die implizite Typumwandlung. Sie sind in gewissen Sinne nicht so ignorant wie vorhin beschrieben, sondern schauen sich, wenn Sie beispielsweise "2" + 1 rechnen wollen (wobei "2" eben ein Zeichenkettenwert ist) die Variablen und ihren Inhalt durchaus genauer an und würden erkennen, dass in unserem Beispiel die Berechnung durchaus möglich wäre, wenn die Zeichenkette "2" in eine Zahl konvertiert würde. Die Konvertierung wird dann bei diesen Sprachen automatisch vorgenommen, ohne, dass Sie mit einer besonderen Anweisung extra eingreifen müssten.

Ein anderes Beispiel hierfür ist die folgende Rechnung: **TRUE – 1**: Hier wird von einem bool'schen Wert ein Ganzzahl-Wert, abgezogen. Sprachen, die ein sehr striktes Typkonzept verfolgen, würden diese Berechnung ablehnen und mit einer Fehlermeldung reagieren. Sprachen, die eine implizite Konvertierung unterstützen würden erkennen, dass **TRUE** letztlich durch den Wert **1** repräsentiert ist, weil es dieser ist, der intern als Wahrheitswert gespeichert wird, auch wenn der Programmierer stets mit dem Label-Konstanten **TRUE** und **FALSE** arbeitet. Demnach lässt sich also der Wert von **TRUE – 1** durchaus bestimmen. In diesem Sinne gilt sogar: **TRUE – 1 = 0 = FALSE**.

Sprachen, die sehr strikt auf die Einhaltung der Typen schauen, wenig implizit umwandeln und ggf. sogar wenig explizite Umwandlung erlauben, nennt man *stark typisiert*. Hier spielen also die Datentypen von Variablen eine große Rolle. Am anderen Ende des Spektrums stehen Sprachen, bei denen der Programmierer sich bzgl. des Typs der Variablen nicht festlegen muss, sondern der Datentyp sich stets durch implizite Konvertierung automatisch so anpasst, dass die gewünschten Operationen möglichst durchgeführt werden können. Solche Sprachen sind „schwächer typisiert“. Im Extremfall können dabei sogar Operationen, die überhaupt keinen Sinn machen, ohne Fehlermeldung durchgeführt werden: Die Addition **3 + "Mein Name"** (wobei 3 eine Zahl, "Mein Name" eine Zeichenkette ist) würde dann vielleicht einfach in 3 resultieren. Die Typumwandlung von "Mein Name" in eine Zahl scheitert natürlich, aber die Programmiersprache ist so schwach typisiert, dass sie einfach „weiterrechnet“ so gut es eben geht.

Was vielleicht verlockend klingen mag, ist aber zugleich auch gefährlich. Denn offensichtlich wäre es schlecht, wenn der Benutzer unseres Online-Shops bei einem Buch die Stückzahl 3 und beim anderen Buch in das Stückzahl-Feld "Mein Name"

einträgt. Mit diesen Informationen können wir nicht wirklich arbeiten. Schlimmstenfalls gerät unser Programm dadurch in Schieflage und produziert unplausible Ergebnisse oder stürzt sogar vollständig ab. Etwas mehr „Kontrolle“, ob die Variablen tatsächlich für die Operationen, die man durchführen will, taugen, sollte man also nicht unbedingt als Einschränkung der eigenen Programmierfreiheit wahrnehmen. Sie ist vor allem ein Hilfsmittel, sichereren, stabileren Programmcode zu schreiben und Fehler frühzeitig, idealerweise bereits während der Entwicklung und nicht erst zu Laufzeit zu erkennen.

### 11.3 Variablen anlegen und initialisieren

---

Im letzten Abschnitt haben Sie gesehen, dass Variablen nach ihrem Datentyp unterschieden werden können, also, danach, welche Art von Informationen sie speichern können, und welcher Wertebereich damit abgedeckt werden kann. Nun stellt sich natürlich die Frage, wie man so eine Variable selbst erzeugt, um damit arbeiten zu können. Die „Geburt“ einer Variablen geht in unterschiedlichen Programmiersprachen unterschiedlich vonstatten. Grob lassen sich aber zwei Arten von Sprachen unterscheiden: Solche, in denen man eine Variable vor erstmaliger Benutzung explizit erzeugen muss, und solche, die die Variable automatisch erzeugen, sobald man sie zum ersten Mal verwendet.

Sprachen von ersterer Art, also solche, bei denen man die Variable zunächst selbst anlegen muss, sind beispielsweise C, Visual Basic for Applications (VBA) und JavaScript. Angenommen, wir wollten in beiden Sprachen einer Ganzzahl-Variablen namens **stueckzahl** den Wert **10** zuweisen. Bevor das geschieht, muss die Variable allerdings angelegt werden. Die Programmierer sprechen in diesem Zusammenhang auch vom *Deklarieren*, also dem Vorgang, durch den dem Compiler oder Interpreter kundgetan wird, dass man fortan diese Variable verwenden möchte. Der Compiler oder Interpreter übernimmt dann den technischen Part der Variablenerzeugung.

Die Deklaration der Variablen samt Zuordnung des Wertes **10** sähe dann in C so aus:

```
int stueckzahl;  
stueckzahl = 10;
```

Durch **int stueckzahl** wird nicht nur eine neue Variable **stueckzahl** deklariert, sondern zugleich auch ihr Typ festgelegt, nämlich als **int**, was in C der Ganzzahl-Datentyp (integer) ist. Nach dieser Deklaration weiß der Compiler, dass es eine Ganzzahl-Variable namens **stueckzahl** gibt, und sie kann von nun an im Programm verwendet werden. Ohne die Deklaration würde die Zuweisung **stueckzahl = 10** zu einer Fehlermeldung führen, mit der der Compiler darauf hinweist, dass er die Variable **stueckzahl** nicht kennt und ihr folglich auch kein Wert zugewiesen werden kann.

### 11.3 · Variablen anlegen und initialisieren

Derselbe Code-Abschnitt in VBA würde folgendermaßen aussehen:

```
Dim stueckzahl As Integer  
stueckzahl = 10
```

Anders als in C (wo die Deklaration mit dem Typ der Variable eingeleitet wurde) wird hier zum Deklarieren der Variablen ein spezielles Schlüsselwort verwendet, nämlich **Dim**, vom engl. *to dimension*, also dimensionieren. Und das trifft es tatsächlich sehr gut, denn was beim Erzeugen der Variable letztlich passiert, ist, dass Speicherplatz reserviert wird, und zwar so viel, wie der Datentyp der Variable eben benötigt. In diesem Sinne wird der Speicher also tatsächlich bedarfsgerecht dimensioniert.

Manchen Sprachen wie JavaScript erfordern zwar eine Deklaration der Variablen, wobei aber kein Typ angegeben wird:

```
var stueckzahl;  
stueckzahl = 10;
```

Hier entscheidet der Compiler bzw. Interpreter später anhand der Verwendung der Variablen, welchen Datentyp er benötigt. Durch die Zuweisung **stueckzahl = 10** wird klar, dass es sich hier um eine Ganzzahl-Variable handeln muss. Wird die Variable später einem Wert zugewiesen, der einen anderen Datentyp erforderlich macht, zum Beispiel durch die Zuweisung **stueckzahl = „Keine Angabe“**, dann ändert sich der Datentyp der Variablen im Hintergrund entsprechend, ohne dass Sie als Programmierer etwas davon mitbekommen. Anders als etwa in C oder VBA erfolgt die Typisierung hier also *implizit*. Bei C und VBA muss dagegen der Typ *explizit* angegeben werden; man spricht daher hier auch von *explizit typisierten Programmiersprachen*.

Sie stellen sich nun vielleicht die Frage, ob eine Deklaration wie in JavaScript tatsächlich so viel bringt, schließlich wird ja der Typ der Variablen gar nicht angegeben. Muss der Typ mit angegeben werden, so kann man argumentieren, dass der Compiler bzw. Interpreter auf diese Weise prüfen kann, ob einer Variablen unabsichtlich Werte zugewiesen werden, die diese aufgrund ihres Datentyps gar nicht aufnehmen kann. Diese zusätzliche Prüfung schafft Sicherheit und verhindert ggf. bereits während der Entwicklung lästige Fehler. Der C-Compiler wird nach der obigen Deklaration bei einer Zuweisung wie **stueckzahl = „Keine Angabe“** den Dienst verweigern und mit einer Fehlermeldung abbrechen. So wird man als Programmierer darauf aufmerksam gemacht, dass man irgendwie unsauber im Code gearbeitet hat.

Hat aber die Notwendigkeit, eine Variable so zu deklarieren, wie es in JavaScript geschieht, also ohne Typangabe, irgendeinen Nutzen für den Programmierer? Oder ist es einfach nur eine Schikane, die sich der Erfinder der Sprache ausgedacht hat, um die Nutzer seiner Erfindung in den Wahnsinn zu treiben? Wie Sie sich leicht vorstellen können, ist letzteres nicht der Fall. Tatsächlich hat es einen

Sinn, den Programmierer zu zwingen, seine Variablen anzumelden. Denn auf diese Weise weiß der Compiler bzw. Interpreter, welche *Variablen-Bezeichner* zulässig sind. Wenn Sie sich dann vertippen, wie es dem Autor beim Schreiben dieser Zeilen gleich mehrfach passiert ist, und zum Beispiel die Zuweisung **steuckzahl = 10** (man beachte die vertauschten Buchstaben e-u) in Ihren Code schreiben, wird der JavaScript-Interpreter sofort erkennen, dass Sie hier versuchen, auf eine Variable zuzugreifen, die es überhaupt nicht gibt, denn deklariert ist ja eine Variable anderen Namens. So werden Sie schnell zur Ursache des Problems vorstoßen und dieses beheben können. Das wäre ungleich schwieriger, wenn die Programmiersprache keine Deklaration erforderte. Dann nämlich würde die Anweisung **steuckzahl = 10** einfach eine neue Variable namens **steuckzahl** erzeugen. In diesem Fall hätten Sie vermutlich einige Mühe, festzustellen, woran es liegt, dass sich Ihr Programm nicht so verhält, wie Sie es wollen. Die wahre Ursache aufzudecken, nämlich, dass Sie tatsächlich mit zwei verschiedenen Variablen arbeiten, ist dann erheblich aufwendiger als wenn Ihnen der Compiler bzw. Interpreter schon mal einen „Tipp“ gibt.

Manchmal kann es also durchaus hilfreich sein, sich einem etwas strengeren Regime zu unterwerfen. Diese Strenge macht es leichter, Probleme aufzudecken und zu lokalisieren. Bietet Ihre Programmiersprache die Möglichkeit, in einen strengeren Modus zu wechseln (was in VBA zum Beispiel durch eine spezielle Option, die zur Variablen-deklaration zwingt, erreicht werden kann), sollte man dieses Angebot durchaus annehmen, auch wenn es auf den ersten Blick nach mehr Kontrolle und weniger Freiheit aussieht.

Zur Strenge, die manche Programmiersprachen ihren Benutzern entgegenbringen gehört manchmal auch, dass die Deklaration von Variablen nur zu Beginn eines Programms/Programmteils erlaubt ist, was die Struktur und damit die Lesbarkeit des Codes verbessert

In unseren Beispielen oben weisen wir der Variablen nach ihrer Deklaration direkt einen Wert zu. Was aber, wenn wir darauf verzichten, die Variable aber später trotzdem benutzen, zum Beispiel ihren Inhalt auf dem Bildschirm ausgeben? Was würde dann angezeigt werden? Anders ausgedrückt: Was ist der Wert, mit dem Variable „geboren“ wird? Früher hatten Variablen nach ihrer Deklaration oftmals einen einigermaßen zufälligen Wert, nämlich jenen, der gerade in dem vom Betriebssystem freigegebenen Speicherbereich stand, der dann für die Variablen reserviert wurde. Dieser Wert war letztlich ein Rückstand desjenigen Programms, das denselben Speicherbereich zuvor genutzt hatte, hinter sich aber nicht „aufgeräumt“ hatte. Deshalb war eine sehr wichtige Empfehlung an jeden Programmierer, seine Variablen stets zu Beginn mit einem Wert zu beladen, sie zu *initialisieren*, wie man im Programmierer-Jargon auch sagt, um ihnen einen klar definierten, bekannten Inhalt zu geben. Das verhinderte, dass das Programm ggf. durch einen „merkwürdigen“ Variableninhalt abstürzte oder in sonst wie unerwünschter Art reagierte.

Heute werden in den meisten Programmiersprachen Variablen automatisch initialisiert, numerische Variablen oft mit 0, Zeichenketten mit einer leeren Zeichenkette. Einige Sprachen haben sogar einen speziellen Wert für nicht ausdrücklich vom Benutzer initialisierte Variablen, wie etwa **undefined** in JavaScript. Dieser Wert signalisiert, dass die Variable noch keinen echten Wert hat, weil sie noch nicht initialisiert worden ist. Viele Sprachen kennen darüber hinaus noch einen anderen Wert,

## 11.5 · Geordnete Felder von Variablen/Arrays

der anzeigt, dass der Benutzer den Wert der Variable bewusst offenlässt (denken Sie etwa an eine nicht beantwortete Frage in einer Meinungsumfrage). In JavaScript zum Beispiel lautet dieser Wert **null** (nicht zu verwechseln mit der Zahl), in Delphi/Object Pascal **nil**, in R **NA** (für *not available*). Hat die Variable einen solchen Wert, so zeigt das an, dass die Variable zwar sehr wohl verwendet wird, nur eben keinen expliziten Wert beinhaltet. Anders ausgedrückt: Kein Wert ist auch ein Wert!

Auch, wenn das Initialisieren heute in vielen Programmiersprachen strenggenommen nicht mehr notwendig ist, so ist es doch gute Praxis. Häufig kann man die Initialisierung direkt bei der Deklaration vornehmen, so zum Beispiel in C. Unser obiges Beispiel würde sich damit verkürzen zu:

```
int stueckzahl = 10;
```

## 11.4 Gar nicht so variabel: Konstanten

Eine andere Art von Sprachelementen, die meist direkt bei der Deklaration initialisiert werden, sind *Konstanten*. Konstanten verhalten sich in dem Sinne ähnlich zu Variablen, als dass es Werte sind, die unter einem bestimmten Namen, ihrem Bezeichner, im Programm angesprochen werden können. Anders als bei Variablen lässt sich aber ihr Wert im weiteren Programmablauf nicht mehr verändern, nachdem er einmal gesetzt worden ist. Das schützt den Wert der Konstante vor versehentlichem Überschreiben. In der Regel müssen Konstanten auch bereits bei der Deklaration mit ihrem (forthin konstanten) Wert initialisiert werden. Ein Beispiel aus Pascal:

```
const pi = 3.14159;
```

In C sieht die Konstantendeklaration genauso aus wie die Deklaration einer Variablen, lediglich ergänzt um das Schlüsselwort **const**:

```
const int pi = 3.14159;
```

## 11.5 Geordnete Felder von Variablen/Arrays

Bislang haben wir immer einzelne Variablen angelegt. Die meisten Programmiersprachen unterstützen aber auch sogenannte Felder (engl. *arrays*) von Variablen. Ein Feld ist eine geordnete Zusammenstellung von Variablen des gleichen Typs, die unter demselben Namen angesprochen werden können. Der Zugriff auf die einzelnen Werte des Feldes geschieht über einen Index.

Im Rahmen unseres hypothetischen Online-Shop-Beispiels könnten wir zum Beispiel die Klick-Historie, also die Folge der Produkte, die sich der Kunde ange-

schaut hat, in einem Array speichern. Das ist eine wichtige Information, wenn es darum geht, das Kundenverhalten genauer zu analysieren und dem Kunden, wie es heute bei vielen Online-Shops ja üblich ist, passgenaue Vorschläge zu machen, welche Produkte ihn gegebenenfalls auch interessieren könnten.

In diesem Beispiel könnte unser Feld **historie** heißen. In diesem Feld würden wir nacheinander die IDs der Produkte speichern, die sich der Kunde angesehen hat. Wir wollen hier annehmen, dass diese IDs Ganzzahlwerte sind. Dann haben wir also ein Feld von Ganzzahlvariablen. Mit einem Index können wir jetzt auf die einzelnen Elemente des Felds zugreifen. Den fünften Eintrag in der Klickhistorie könnten wir so durch **historie[5]** abgreifen. In den eckigen Klammern steht mit dem Index also die Nummer des Elements, auf das zugegriffen werden soll, hier also das fünfte Produkt, das sich unser aktueller Kunde angesehen hat.

Natürlich hätten wir das Ganze auch mit einzelnen Variablen realisieren können: So hätten wir Variablen **historie1**, **historie2**, **historie3**, **historie4**, **historie5**, **historie6** etc. anlegen und die Folge der betrachteten Produkte in diesen Variablen speichern können. Allerdings hat das einige Nachteile: Zum einen nämlich müssen, wie wir im vorangegangenen Abschnitt gesehen haben, in vielen Sprachen die Variablen ja explizit deklariert werden. Wenn Sie eine Klick-Historie mit, sagen wir, 30 Produkten vorsehen, hätten Sie eine Menge Schreibarbeit, nur, um die Variablen überhaupt einmal anzumelden. Zum anderen gibt es in praktisch allen Sprachen, die Felder unterstützen, sehr effiziente Mechanismen, um diese Felder zu durchlaufen, in dem nämlich der Index, der ein Feld-Element identifiziert nach und nach immer um eine Position weitergeschoben wird. So können Sie das ganze Feld Schritt für Schritt durchgehen. Das Ganze lässt sich zudem programmiertechnisch sehr elegant lösen, sodass Sie wenig Code schreiben müssen. Ungleich komplexer und pflegeaufwendiger (denken Sie an den Fall, dass Sie einfach mal eben schnell die Länge der Historie von 30 auf 100 Produkte hochsetzen wollen) wäre es, wenn Sie mit einzelnen, unabhängigen Variablen arbeiten würden.

Wenn auch praktisch alle modernen Hochsprachen Felder anbieten, so unterscheiden sich die Sprachen aber in einem wichtigen Punkt; nämlich in der Frage, welches der Index-Wert des *ersten Feldelements* ist. In vielen Sprachen starten Feld-Indizes bei 0. Dann wäre **historie[0]** die Produkt-ID des ersten Produkts, das der Benutzer sich angesehen hat.

Felder/Arrays können auch mehrdimensional sein. In unserem Beispiel könnten wir die Klickhistorie aller unserer Besucher in einem Feld speichern; wir würden ein zweidimensionales Feld verwenden, das wir uns als Tabelle oder Matrix vorstellen können. In den Zeilen stehen die Benutzer, in den Spalten die IDs der Produkte, die sie sich angesehen haben. Dann wäre **historie[3][1]** die ID, die sich Besucher Nummer 3 als erstes angesehen hat (vorausgesetzt, unsere Feldindizierung beginnt bei 1). Um auf Elemente eines Feldes zuzugreifen, brauchen wir jetzt zwei Indizes als Koordinaten, die genau beschreiben, wohin wir in unserem zweidimensionalen Tableau greifen wollen.

Die Dimensionalität von Feldern ist natürlich keineswegs auf zwei Dimensionen beschränkt. Wir könnten ohne weiteres eine dritte, vierte, fünfte Dimension hinzufügen. Alles kein Problem, solange wir noch im Blick haben, welcher Index

(und damit welche Dimension) für welche „Koordinate“ steht, mit der wir die Informationen in unserem Feld ablegen. So könnten wir zum Beispiel neben dem Benutzer auch den Tag (von 1 = Montag, bis 7 = Sonntag) speichern und hätten damit eine dritte Dimension. Unser Feld hätte damit den Aufbau **historie[tag][benutzer][klicknummer]** und wir würden mit **historie[2][3][1]** die ID des ersten Produkts erhalten, dass sich Benutzer 3 am Dienstag angeschaut hat.

Ohne besonders darauf einzugehen, haben wir in den vergangenen Absätzen eine Notation eingeführt, um mit Indizes auf die einzelnen Elemente von Feldern zuzugreifen: Wir haben die Indexnummer in eckige Klammern geschrieben. So machen das tatsächlich viele Programmiersprachen, aber nicht alle: Einige setzen die Indizes in *runde* Klammern. Die Möglichkeiten der Indizierung erschöpfen sich allerdings darin, einfach eine Indexnummer anzugeben. Viele Programmiersprachen erlauben weitere Methoden der Indizierung, zum Beispiel die Indizierung durch Ausschluss: Dabei wird nicht der Index oder die Indizes jener Elemente des Feldes angegeben, die man selektieren möchte, sondern gerade umgekehrt, diejenigen, die man nicht selektieren möchte. Oft geschieht das, indem dem Index ein Minuszeichen vorangestellt wird. **historie[-5]** wäre dann das gesamte Feld mit Ausnahme des fünften Elements. In manchen Programmiersprachen bedeutet ein negativer Wert allerdings auch, dass vom Ende des Feldes her indiziert wird. Dann entspräche **historie[-5]** dem fünften Element von *hinten*. Einige Sprachen bieten auch die Angabe eines ganzen Bereichs von Indizes an: so würde man mit **historie[5:9]** das fünfte, sechste, siebte, achte und neunte Element des Felds greifen. Eine Schreibweise in der Form **historie[von:bis]** spart nicht nur Tipparbeit, sondern macht es vor allem dann einfacher, wenn die Selektionsgrenzen von und bis a priori noch gar nicht bekannt sind, sondern an ihrer Stelle Variablen eingesetzt werden, deren Werte erst durch das Programm bestimmt werden (zum Beispiel durch eine Benutzereingabe).

Die Welt der Felder ist quer über die Programmiersprachen betrachtet verhältnismäßig bunt. Den meisten sind aber folgende Grundsätze gemeinsam:

- Felder sind Zusammenstellungen mehrerer einzelner Variablen (zu dieser Fundamentalen Festlegung gibt es allerdings durchaus Ausnahmen: In der Statistiksprache R sind eindimensionale Felder, sogenannte Vektoren, der Standard; eine einzelne Variable ist dann nur ein Spezialfall eines solchen Vektors, nämlich ein Vektor der Länge eins).
- Die Variablen in einem Feld haben alle denselben Typ (sind also zum Beispiel alle Zeichenketten, oder alle Zahlen).
- Felder können ein- oder mehrdimensional sein.
- Auf einzelne Elemente der Felder kann über einen numerischen Index durch Angabe des zu selektierenden Elements zugegriffen werden.
- Bei der Erzeugung eines Feldes (sofern Variablen überhaupt deklariert werden müssen; wir haben gesehen, dass das nicht auf alle Programmiersprachen zutrifft, müssen) dessen Dimensionen sowie der Typ der Feldvariablen angegeben werden.

Darüber hinaus kann sich die Arbeit mit Feldern in verschiedenen Programmiersprachen aber stark unterscheiden. Wir hatten bereits einige dieser Unterschiede

kennengelernt. Zusammenfassend können unter anderem folgende Frage in verschiedenen Programmiersprachen durchaus sehr unterschiedlich gehandhabt werden:

- Ob die numerischen Indizes bei 0 oder bei 1 beginnen.
- Ob runde oder eckige Klammern bei der Indizierung verwendet werden.
- Welche Indizierungsmöglichkeiten über die einfache Angabe der Elementnummer hinaus es gibt.
- Ob, und wenn ja, welche Funktionen zur Verfügung stehen, um mit Feldern zu arbeiten (also beispielsweise die Länge eines Feldes zu ermitteln oder ein Element aus einem Feld zu löschen).
- Welche Datentypen überhaupt für Felder erlaubt sind.
- Wie groß Felder maximal sein dürfen.

Schauen wir uns als nächstes an, wie in einigen Programmiersprachen Felder tatsächlich deklariert und benutzt werden. In allen Fällen wollen wir ein Feld aus sechs Variablen anlegen, dass die Namen einer Personengruppe aufnehmen können soll. Den zweiten Namen wollen wir dann auf „Ulrike“ setzen.

Das ganze zunächst in Visual Basic for Application (VBA):

```
Dim Namen(6) as String
Namen(1) = "Ulrike"
```

Das gleiche in JavaScript:

```
var Namen = [];
Namen[1] = "Ulrike"
```

Wie Sie sehen, muss in JavaScript die Größe des Arrays nicht im Vorhinein festgelegt werden. Die Indizierung beginnt auch hier bei 0.

Und schließlich noch in Delphi:

```
var
Namen: array[1..6] of string;
Namen[2] := "Ulrike";
```

Hier kann der Wertebereich der Indizes explizit festgelegt werden. Wir stellen ihn so ein, dass er von 1 bis 6 läuft. Das zweite Element hat damit tatsächlich den Index 2.

Zum Abschluss sei noch erwähnt, dass in einigen Programmiersprachen Zeichenketten-Variablen als Felder einzelner Zeichen aufgefasst werden. Auf die einzelnen Zeichen innerhalb der Zeichenkette kann dann durch normale Indizierung zugegriffen werden. Betrachten Sie dazu die folgenden Beispiele aus C und Python; zunächst die C-Version:

```
char mein_name[] = "Thomas";
printf("Drittes Zeichen: %c", mein_name[2]);
```

Dann in Python:

```
mein_name = "Thomas"
print("Dritten Zeichen: ", mein_name[2])
```

In beiden Fällen greifen wir jeweils das Zeichen mit dem Index 2, also das dritte Zeichen (weil beide Programmiersprachen die Feld-Indizierung bei 0 beginnen) und zeigen es an.

Beide Sprachen begreifen Strings (zumindest auch) als Felder, wobei C hier noch viel strikter als Python ist.

## 11.6 Assoziative Felder von Variablen/Hashes

---

Manche Programmiersprachen kennen neben den klassischen, geordneten Feldern/Arrays noch einen zweiten Typ von Feldern, die assoziativen Felder, gelegentlich auch als Hashes, Dictionaries (Wörterbücher) oder Maps (Karten) bezeichnet.

Assoziative Felder bestehen aus einer *ungeordneten* Menge von *Schlüssel-Wert*-Pärchen. Über den Schlüssel kann auf den jeweiligen Wert zugegriffen werden. Solche Schlüssel-Wert-Pärchen könnten zum Beispiel aus dem Namen eines Kunden (Schlüssel) und dem Bestellwert seines letzten Auftrags (Wert) bestehen.

Die Einträge in einem assoziativen Feld haben, anders als die in einem geordneten Feld/Array, keine natürliche Reihenfolge; die brauchen sie auch nicht, greift man doch über einen klar inhaltlich definierten Schlüssel auf die einzelnen Elemente zu.

Schauen wir uns das am Beispiel von zwei Sprachen an, die assoziative Felder unterstützen: Perl – assoziative Felder heißen hier *Hashes* – und Python, wo dieser Datenstrukturen als *Dictionaries* bezeichnet werden.

Zunächst die Perl-Variante:

```
my %bestellwerte = (
    "Thomas_Schulz" => 43.99,
    "Max_Meier" => 19.49,
    "Susanne_Mueller" => 68.99,
);
$bestellwerte{"Susanne_Mueller"} = 8.99;
print('Bestellwert Meier: $bestellwerte{"Max_Meuer"}');
```

Wie Sie sehen, wird im oberen Teil des Codes zunächst eine neues Hash-Feld namens bestellwerte angelegt (den Bezeichnern von Hashes wird in Perl immer das

Prozentzeichen vorangestellt, wenn vom Feld als „ganzem“ gesprochen wird). Das Hash-Feld wird sogleich mit drei Schlüssel-Wert-Pärchen initialisiert. Links des Operators `=>` steht dabei jeweils der Schlüssel, der Name des Kunden, rechts vom Operator der letzte Bestellwert. Weiter unten im Code wird dann auf jeweils ein spezielles Element des Hash-Felds zugegriffen, einmal, um einen Wert zu ändern, ein anderes Mal, um einen Wert auf dem Bildschirm auszugeben. Der Zugriff erfolgt erwartungsgemäß nicht mit einem numerischen Index (schließlich sind die Elemente im Hash-Feld ja nicht sortiert), sondern über den Schlüssel, hier also den Kundennamen.

Dasselbe jetzt nochmal in Python:

```
bestellwerte = {
    "Thomas_Schulz" : 43.99,
    "Max_Meier" : 19.49,
    "Sophie_Mueller" : 68.99,
}
bestellwerte['Sophie_Mueller'] = 8.99
print("Bestellwert von Max Meier: ",
      bestellwerte['Max_Meier'])
```

Wenn auch die exakte Syntax in beiden Sprachen leicht voneinander abweicht, die Parallelen im Umgang mit Hashes bzw. Dictionaries, wie die assoziativen Felder in Python heißen, ist unübersehbar. Es wird deutlich, wie leicht Werte in einem assoziativen Feld – in der Tat gleichsam einem Wörterbuch, einem Dictionary – nachgeschlagen werden können.

## 11

Die Sprachen, die assoziative Felder unterstützen, bringen in der Regel eine ganze Reihe von Werkzeugen zur Analyse und Bearbeitung solcher Felder mit. So stehen dem Programmierer zum Beispiel regelmäßig Funktionen oder Operatoren zur Verfügung, um auf einen Schlag alle Schlüssel oder alle Werte aus dem Feld zu extrahieren, oder um die Größe des Feldes, das heißt, die Anzahl der enthaltenen Schlüssel-Wert-Pärchen, zu bestimmen.

### 11.1 [3 min]

Was versteht man unter der Deklaration von Variablen?

### 11.2 [3 min]

Nennen Sie zwei Vorteile, die der Zwang, Variablen zu deklarieren, mit sich bringt.

## 11.7 Objekte

Im letzten und im vorletzten Abschnitt haben wir uns mit Feldern beschäftigt. Felder erlauben es, viele gleichartige Informationen geordnet abzulegen und wieder darauf zuzugreifen. Das ist in vielen Fällen sehr nützlich, oftmals aber nicht der einfachste und natürlichste Weg, mit Daten umzugehen.

In diesem Abschnitt lernen Sie deshalb einen Ansatz kennen, mit zusammengehörenden Daten zu arbeiten, der so grundlegend ist, dass er ein eigenes Programmier-Paradigma darstellt. Viele Programmiersprachen haben sich diesem Ansatz ganz oder in Teilen verschrieben. Weil er vielen populären Sprachen, wie etwa C++, Java und JavaScript, Python und Kotlin prägt, hat er in der Praxis eine außerordentlich große Bedeutung.

Die Rede ist von der sogenannten *objektorientierten Programmierung* (kurz OOP). Mit ihr beschäftigen wir uns in diesem Abschnitt recht ausgiebig und das nicht ganz ohne Hintergedanken, gehören doch die beiden Programmiersprachen, in die der vierte und der fünfte Teil dieses Buches einführen, ebenfalls in die breite Klasse der objektorientierten Sprachen.

Der Objektorientierung haften in den Augen nicht weniger Zeitgenossen Attribute wie „schwer verständlich“ und „komplex“ an. Sie werden allerdings nach der Lektüre dieses Abschnitts feststellen, dass derlei Befürchtungen glücklicherweise überhaupt nicht berechtigt sind.

### 11.7.1 Eine Welt aus lauter Objekten

---

Gehen wir davon aus, wir wollten für einen Online-Shop eine katalogartige Anzeige der Produkte programmieren. Jedes Produkt hat eine Reihe von Eigenschaften, die wir anzeigen wollen, eine Bezeichnung, eine Beschreibung, eine Artikelnummer, einen Hersteller, einen Preis und wahrscheinlich noch eine ganze Reihe anderer Attribute. Mit dem Wissen aus den vorangegangenen Abschnitten könnten wir diese Eigenschaften jeweils als eigenes Feld/Array abbilden. Dann gäbe es zum Beispiel ein Array **artikelnummern** und **artikelnummern[187]** wäre die Artikelnummer des 187. Artikels. Würde man zu demselben Artikel die Artikelbeschreibung abrufen wollen, würde man auf das Array-Element **beschreibung[187]** zurückgreifen.

Das führende Kriterium ist also immer die jeweilige Eigenschaft. Das Produkt, *für das* wir diese Eigenschaft abfragen, wird durch den Index angegeben, in unserem Beispiel also die 187. Diese Herangehensweise ist aber ein wenig unnatürlich, geradezu gekünstelt. Denn in der Realität gehen wir normalerweise nicht von der Eigenschaft aus, sondern von ihrem Träger.

So stehen wir bei der Programmierung unserer Katalogansicht vor dem Problem, dass wir irgendein *Produkt* vor uns haben und es darstellen müssen, und zwar mit *allen* relevanten Eigenschaften. Wir kommen also vom Produkt her und fragen uns, welche Attribute dieses Produkt nun besitzt. Und so nehmen wir uns dann die Bezeichnung, die Beschreibung, die Artikelnummer und alle weiteren Eigenschaften, die wir in unserer Katalogliste darstellen wollen und zeigen sie für das jeweilige Produkt an. Wir betrachten dabei also stets ein- und dasselbe *Objekt*, nämlich das Produkt, unter allen möglichen Gesichtspunkten (seinen *Attributen*).

Genauso ist es, wenn Sie zum Gebrauchtwagenhändler gehen und die auf dessen Hof die zum Verkauf angebotenen Autos begutachten. Sie schauen sich ein Auto an, prüfen Modell, Farbe, Alter, Erhaltungszustand, Preis und weitere Parameter, die für Ihre Einschätzung wichtig sind. Dann schauen Sie sich das nächste

Auto an und dann das darauffolgende. Immer gehen Sie aber von einem Objekt, dem Auto aus, und betrachten dessen jeweilige Eigenschaften.

Sie haben sicher schon gemerkt, worauf diese Überlegung hinausläuft: Die Welt ist einfach nicht nach Eigenschaften organisiert, sondern nach Objekten, ob das nun Produkte, Autos, Häuser, Unternehmen, Buttons, E-Mails oder Studenten sind. Alle diese physischen und nicht-physischen Objekte sind letztlich Zusammenfassungen von Eigenschaften. In diesem Sinne können auch Personen oder Rollen, die Personen ausüben (etwa die Rolle „Student“), Objekte darstellen; wir sollten es hier mit der Sprachwahl nicht so genau nehmen, auch wenn es am Anfang dem ein oder anderen vielleicht etwas merkwürdig vorkommen mag, einen Kunden oder einen Kollegen als „Objekt“ zu betrachten. Wenn wir aber ein Objekt als eine Zusammenstellung von Eigenschaften definieren, wird klar, dass in diesem weiten Sinne auch Menschen, Tiere, Pflanzen und Götter „Objekte“ sind.

Wenn aber die Welt nach Objekten und nicht nach Eigenschaften organisiert ist, warum spiegelt sich das eigentlich nicht in der Programmierung wider? Warum arbeiten wir mit Feldern, die klar eine einzelne Eigenschaft in den Vordergrund stellen und nicht ein Objekt als eine ganze Sammlung verschiedener Eigenschaften?

Diese Frage haben sich in den 1960er-Jahren der amerikanische Informatiker Alan Kay und andere Vordenker der objektorientierten Programmierung auch gestellt und als Antwort im wahrsten Sinne des Wortes einen Paradigmenwechsel herbeigeführt. Eine der ersten Programmiersprachen, die diesem neuen Paradigma folgte, war dann auch Kays *Smalltalk*.

Im Konzept der objektorientierten Programmierung steht das Objekt im Vordergrund, es verkörpert das organisierende Prinzip dieses Ansatzes. Nicht länger die Eigenschaften sind es, die die Art, wie wir mit Daten umgehen, bestimmen, es sind die Träger der Eigenschaften, die Objekte.

## 11

### 11.7.2 Klassen

Folgen wir also dem objektorientierten Ansatz und definieren ein Objekt **produkt** mit folgenden Eigenschaften:

```
produkt.bezeichnung
produkt.beschreibung
produkt.artikelnummer
produkt.hersteller
produkt.preis
```

Alle Produkte, die wir über unserem Shop vertreiben, besitzen diese Eigenschaften. Damit man erkennt, dass die Eigenschaften alle das Objekt „Produkt“ beschreiben, fassen wir sie zusammen, in dem wir Ihnen **produkt** vorstellen.

Lassen Sie uns jetzt sprachlich noch ein klein wenig präziser werden. Was wir da definiert haben, ist unsere Sicht auf ein *beliebiges* Produkt, es ist gewissermaßen die *Schablone* oder *Vorlage für ein Produkt*: So sehen Produkte für uns aus, das sind aus unserer Sicht ihre wesentlichen Eigenschaften.

Eine solche abstrakte Vorlage, die beschreibt, welche Eigenschaften ein Objekt hat, bezeichnet man in der objektorientierten Programmierung als *Klasse*. Jedes reale Produkt, das wir anbieten, hat für jede dieser Eigenschaften einen jeweils individuellen Wert, zum Beispiel die Bezeichnung „Gartenschaufel, Edelstahl“ und einen Preis von 10,99 EUR. Die eigentlichen Objekte, deren Eigenschaften nach dem Vorbild unserer Klasse aufgebaut sind, nennt man in der objektorientierten Programmierung *Instanzen* der Klasse. Eine Instanz ist also gewissermaßen die Konkretisierung der abstrakten Idee, die in einer Klasse zum Ausdruck kommt. Alle unsere Produkte werden unterschiedliche Werte für jede der Eigenschaften haben, dementsprechend existieren so viele Instanzen wie es Produkte gibt. Alle Produkte aber gehören zu gleichen Klasse, sie sind eben Produkte. Dementsprechend haben die alle Eigenschaften, die Produkte nun einmal haben, wenn auch jeweils unterschiedlich ausgeprägt.

Im nächsten Schritt werden wir etwas formaler und definieren unsere Klasse so, wie wir es in einer Programmiersprache tun würden:

```
Klasse Produkt
Beginn
    bezeichnung : Zeichenkette
    beschreibung : Zeichenkette
    artikelnummer : Ganzzahl
    hersteller : Zeichenkette
    preis : Fliesskommazahl
Ende
```

Die Eigenschaften der Klasse, man spricht in diesem Zusammenhang auch von *Attribut*en – stehen zwischen den begrenzenden Schlüsselwörtern **Beginn** und **Ende**. Natürlich ist dieser Code-Ausschnitt hier nicht in einer tatsächlich existierenden Programmiersprache verfaßt, sondern als „Pseudo-Code“ formuliert, wie wir es im weiteren Verlaufe dieses Teils des Buches noch öfters tun werden, um Grundprinzipien anschaulich zu machen. Es geht hier lediglich darum, in formalisierter, aber gut verständlicher Weise zu beschreiben, wie eine Klassendefinition aussehen kann. Später werden Sie sehen, wie Klassendefinitionen in einigen Programmiersprachen aufgebaut sind und werden damit sofort zurechtkommen, wenn Sie das Grundkonzept anhand dieses Pseudo-Codes verstanden haben.

Nachdem wir nun definiert haben, aus welchen Attributen unsere Klasse bestehen soll und welchen Datentyp diese Attribute haben, können wir eine Instanz der Klasse erzeugen, also eine Variable, die – aufgebaut nach dem Vorbild der Klassendefinition – ein konkretes Produkt darstellt. Sobald die neue Variable vom Typ **Produkt** angelegt ist, können wir damit beginnen, ihre Attribute anzupassen:

```
Gartenschaufel : Produkt

Gartenschaufel.bezeichnung = "Gartenschaufel, Edelstahl"
Gartenschaufel.preis = 10.99
```

Um auf die Attribute der Instanz **Gartenschaufel** unserer Klasse **Produkt** zuzugreifen, benutzen wir den Punkt-Operator in der Form **Instanz.Klassen-Attribut**. Diese Notation ist tatsächlich in vielen Programmiersprachen gängig.

Wir haben uns nun also mit **Produkt** einen eigenen Datentyp geschaffen, der komplexer ist als die Datentypen, die wir in den vorangegangenen Abschnitten kennengelernt haben, denn er speichert unterschiedliche Werte. Arbeiten lässt sich mit ihm aber ebenso wie mit einem der „fest eingebauten“ Datentypen (wie etwa Ganzzahlen oder Wahrheitswerten) auch: So können wir zum Beispiel Variablen dieses Typs anlegen und Werte zuweisen (nur diesmal eben nicht der Variable als ganzen, sondern den einzelnen Attributen, die natürlich wiederum elementare Variablen sind).

### 11.7.3 Vererbung

Manchmal haben wir Objekte, die Spezialfälle von anderen Objekten sind. Ein Buch zum Beispiel ist ein spezielles Produkt. Es besitzt alle Eigenschaften, die unsere Produkte nun einmal so haben, es hat eine Bezeichnung (den Buchtitel), einen Hersteller (den Verlag), und natürlich besitzt es auch einen Preis. Darüber hinaus hat es noch einige weitere Eigenschaften, die wir in unserem Webshop auch zeigen sollten, zum Beispiel den Autor und die Seitenzahl. Beides sind immerhin Informationen, die die Kaufentscheidung unserer Kunden beeinflussen könnten.

**11** Um nun unser spezielles Produkt „Buch“ als Klasse abzubilden, gibt es in der objektorientierten Programmierung einen Trick, die sogenannte *Vererbung*. Anders als der etwas martialisch anmutende Begriff vielleicht suggeriert, muss hier niemand sterben, um dieses elegante Konzept anwenden zu können. Die Grundidee ist allerdings wirklich totsimpel: Unser Buch als Spezialfall des Produkts „erbt“ einfach alle Attribute, die ein Produkt hat und bekommt mit dem Autor und der Seitenzahl noch zwei weitere Attribute dazu. Diese beiden Eigenschaften machen das Spezielle am Buch aus. Unser Buch ist also ein Produkt, aber nicht jedes Produkt ist auch ein Buch. Es gibt Produkte, die nur die Standardeigenschaften von Produkten besitzen, nicht aber die speziellen Eigenschaften Autor und Seitenzahl; diese sind nur Büchern zu eigen.

Natürlich könnten wir diese zusätzlichen Eigenschaften, die Bücher über die allgemeine Definition eines Produkts hinaus noch haben, auch direkt in die Klasse **Produkt** mit aufnehmen. Aber welche Werte sollten wir diesen Attributen dann bei konkreten Instanzen unserer Klasse zuweisen, die kein Buch sind? Und was passiert, wenn wir neben Büchern noch weitere Spezialfälle von Produkten besonders behandeln, zum Beispiel Bekleidung, oder Gartenmöbel? Die Zahl der Attribute, die dann nicht mehr auf alle Produkte anwendbar sind, sondern nur noch auf eine einzelne Produktkategorie, würde erheblich steigen, die Klasse **Produkt** dadurch sehr unübersichtlich werden.

Einfacher und eleganter geht es mit Vererbung. Wir erschaffen dabei eine neue Klasse **Buch**, die alle Eigenschaften der oben definierten Klasse **Produkt** übernimmt und zusätzlich noch die Attribute **seitenzahl** und **autor** mitbringt. Die Klassendefinition für die Klasse **Buch** könnte dann so aussehen:

```
Klasse Buch Erbt Produkt
Beginn
    autor : Zeichenkette
    seitenzahl : Ganzahl
Ende
```

Wenn Sie diese Klassendefinition mit der Definition der Klasse **Produkt** vergleichen, stellen Sie fest, dass hier das Schlüsselwort **Erbt** hinzutritt, gefolgt von der Klasse, deren Attribute unsere Klasse **Buch** übernehmen (also „erben“ soll).

Natürlich können wir jetzt wiederum Instanzen unserer Klasse anlegen, also konkrete Variablen, und deren Attribute verändern.

```
Grisham1992 : Buch

Grisham1992.bezeichnung = "Die Akte"
Grisham1992.preis = 8.99
Grisham1992.autor = "John Grisham"
Grisham1992.seitenzahl = 478
```

Wie Sie hier sehen, bearbeiten wir in diesem Code-Ausschnitt unsere Variable **Grisham1992** nicht nur in Hinblick auf die speziellen Eigenschaften von Büchern, nämlich Autor und Seitenzahl, sondern auch bezüglich der Standard-Attribute von Produkten, nämlich Bezeichnung und Preis. Diese kommen zwar in der Definition der Klasse **Buch** gar nicht explizit vor. Bücher erben diese Eigenschaften aber von der allgemeineren Klasse **Produkt**, von der die Klasse **Buch** abgeleitet ist.

Letztlich bauen wir also eine *Klassenhierarchie* auf, mit **Produkt** als Ober- und **Buch** als Unterkategorie. Klassenhierarchien können natürlich sehr viel mehr als zwei Ebenen haben; so könnten wir als weitere Unterklassen von Büchern zum Beispiel Romane und Fachbücher hinzufügen und diese mit speziellen Attributen versehen, die in der Klasse **Buch** nicht enthalten sind. Genauso könnten wir die Hierarchie natürlich auch in die Breite ziehen, in dem wir neben Büchern weitere Produktkategorien (etwa die bereits erwähnten Kategorien Bekleidung und Gartenmöbel) als separate Klassen modellieren, die von der Klasse **Produkt** direkt abgeleitet sind.

Schauen wir uns als nächstes einmal an, wie unsere beiden Klassen **Produkt** und **Buch** in zwei echten Programmiersprachen aussehen würden; wir beginnen mit C++:

```
class Produkt
{
    char bezeichnung[30];
    char beschreibung[200];
    long artikelnummer;
    char hersteller[30];
    float preis;
}
```

```

class Buch : public Produkt
{
    char autor[50];
    int seitenzahl;
}

// Später im Hauptprogramm ...

Buch Grisham1992;
Produkt Gartenschaufel;

Gartenschaufel.bezeichnung = "Gartenschaufel, Edelstahl";
Gartenschaufel.preis = 10.99;

Grisham1992.bezeichnung = "Die Akte";
Grisham1992.preis = 8.99;
Gisham1992.autor = "John Grisham";
Gisham1992.seitenzahl = 478;

```

Auch, wenn Sie natürlich mit der besonderen Syntax von C++ nicht vertraut sind, können Sie sich mit Ihrem Verständnis der Konzepte objektorientierter Programmierung und unseres Pseudo-Codes durchaus erschließen, was in diesem Programmausschnitt passiert.

Schauen wir uns dasselbe nun in Delphi/Object Pascal an:

## 11

```

type
TProdukt = Class(TOObject)
    property bezeichnung : String;
    property beschreibung : String;
    property artikelnummer: Longint;
    property hersteller : String;
    property preis : Single;
end;

TBuch = Class(TProdukt)
    property autor : String;
    property seitenzahl : Integer;
end;

// Später im Hauptprogramm ...

var
Gartenschaufel : TProdukt;
Grisham1992 : TBuch;

// ...

Gartenschaufel.bezeichnung := "Gartenschaufel, Edelstahl";
Gartenschaufel.preis := 10.99;

```

```
Grisham1992.bezeichnung := "Die Akte";
Grisham1992.preis := 8.99;
Gisham1992.autor := "John Grisham";
Gisham1992.seitenzahl := 478;
```

Wir sind hier der Delphi-typischen Notation gefolgt, dass Klassen (und überhaupt alle über die Basisdatentypen hinaus definierten Datentypen) mit „T“ beginnen, unsere beiden Klassen heißen dann hier entsprechend **TProdukt** und **TBuch**. Hinter dem Schlüsselwort **Class** steht in Klammern, die Klasse, von der geerbt wird, die also eine Ebene höher in der Klassenhierarchie steht. Das Interessante ist hier, dass auch die Klasse **TProdukt** Attribute von einer höheren Klasse erbt, nämlich von der Klasse **TObject**. Diese Klasse ist die höchste in der Klassenhierarchie, alle anderen Klassen sind letztlich von ihr abgeleitet.

Sie sehen an diesen Beispielen, dass Klassendefinitionen in Programmiersprachen zwar ihre Eigenheiten haben mögen, allerdings dennoch große Gemeinsamkeiten aufweisen. Mit den wenigen Grundgedanken objektorientierter Programmierung, die wir uns bis hierher angeschaut haben, können Sie sich bereits ohne Detailverständnis der unterschiedlichen Programmiersprachen erschließen, was die Klassendefinitionen in der jeweiligen Sprache bedeuten.

#### 11.7.4 Methoden

In den Beispielen des vorangegangenen Abschnitts haben wir die Attribute, also Eigenschaften unserer Klassen, direkt verändert, indem wir ihnen Werte zugewiesen haben. Für die „reine Lehre“ der objektorientierten Programmierung ist das ein Sakrileg. Der „reinen Lehre“ zufolge dürfen nämlich die Attribute nicht direkt bearbeitet werden, sondern nur mit Hilfe sogenannter *Methoden*.

Methoden sind aufrufbare Teilprogramme, denen man bestimmte Werte, die sogenannten *Argumente*, übergeben kann und die dann diese Werte in irgendeiner Weise verarbeiten, zum Beispiel eben, indem sie den übergebenen Wert einer Klassen-Eigenschaft zuweisen.

Um das ganze etwas konkreter zu machen, nehmen wir an, unsere Klasse **Produkt** hätten eine Methode **setzePreis()**, mit der der Preis bearbeitet werden kann. Der Methode wird der Preis als Argument übergeben, und die Methode wiederum sorgt dann dafür, dass die Klassen-Eigenschaft **preis** entsprechend geändert wird. Die Klassendefinition sähe dann so aus:

```
Klasse Produkt
BeginnKlasse
    bezeichnung : Zeichenkette
    beschreibung : Zeichenkette
    artikelnummer : Ganzzahl
    hersteller : Zeichenkette
    preis : Fliesskommazahl
    setzePreis(neuerPreis: Fliesskommazahl)
EndeKlasse
```

Unsere ursprüngliche Klasse ist also um die Methode `setzePreis()` ergänzt worden. Diese Methode übernimmt als Argument `neuerPreis` eine Fließkommazahl, nämlich den Preis, den wir für unser Produkt setzen wollen.

Wir könnten dann später im Programm eine neue Instanz der Klasse **Produkt** erzeugen und den Preis mit Hilfe der Methode `setzePreis()` initialisieren, hier im Beispiel auf den Preis 10,99:

```
Gartenschaufel : Produkt
Gartenschaufel.setzePreis(10.99)
```

Hier sehen wir nun scheinbar einen erheblichen Unterschied zu den fundamentalen Datentypen, die wir bislang kennengelernt haben, wie etwa Ganzzahl- oder Zeichenketten-Variablen: Die Klassen der objektorientierten Programmierung beinhalten nämlich nicht nur Datenwerte, sondern mit den Methoden auch gleich die Werkzeuge, um diese Daten zu bearbeiten. Das ist aber nur ein scheinbarer Unterschied; tatsächlich sind auch die fundamentalen Datentypen in vielen objektorientierten Sprachen selbst Klassen, die nach außen eine Reihe von Methoden anbieten. So könnte zum Beispiel die Klasse **Fliesskommazahl** eine Methode `runden()` zur Verfügung stellen; wäre `preis` ein **Fliesskommazahl**-Objekt, also eine Instanz der Klasse **Fliesskommazahl**, dann würde zum Beispiel `preis.runden(2)` den Wert der Variablen `preis` auf zwei Nachkommastellen runden.

Warum aber so kompliziert? Warum bleiben wir nicht einfach dabei, den Attributen unserer Klassen-Instanzen direkt Werte zuzuweisen? Warum ist eine spezielle Methode nötig, die selbst ja auch entwickelt werden muss? In unserem Beispiel oben hatten wir der Einfachheit halber darauf verzichtet und sind davon ausgegangen, dass die Methode `setzePreis()` schon irgendwo programmiert ist und deshalb von uns verwendet werden kann; deshalb hat in der Klassendefinition ein Hinweis (ein sogenannter *Prototyp*) darauf genügt, dass die Methode Bestandteil der Klasse sein soll. Tatsächlich aber muss der Code, der hinter dieser Methode steht, der also ausgeführt wird, wenn die Methode aufgerufen wird, natürlich auch entwickelt werden. Warum also der ganze Aufwand, nur um einen Wert zu ändern, was wir auch mit einer einfachen Zuweisung hätte erledigen können?

Die Befürworter der objektorientierten Programmierung würden argumentieren, dass durch die Verwendung von Methoden die interne Datenstruktur der Klasse nach außen, also gegenüber dem Programmierer, der die Klasse verwendet, abgeschirmt wird. Der Programmierer muss sich gar nicht darum kümmern, wie genau die verschiedenen Sachverhalte in der Klasse genau abgebildet werden; er bearbeitet die Klassen-Attribute ja nicht direkt, sondern über die Methoden. Der Entwickler der Klasse könnte also etwas an den Klassen-Attributen ändern; solange sich die Methoden nicht ändern, die dem Anwender der Klasse, also dem mit ihr arbeitenden Programmierer, zur Verfügung stehen, merkt dieser von den Änderungen nichts. Aus seiner Sicht bleibt alles beim Alten. Er muss seine Software nicht umschreiben, sondern kann mit dem vorhandenen Code ohne Änderungen weiterarbeiten.

Der Vorteil der Verwendung von Methoden liegt also darin, dass die Modularisierung von Code und damit die Arbeitsteilung zwischen Programmierern (dem Entwickler der Klasse und demjenigen, der die Klasse in seinen Programmen einsetzt) vereinfacht wird. Der Entwickler der Klasse ist dann zuständig für die Funktionalität, die seine Klasse über die Methoden bereitstellt, der Programmierer als „Konsument“ dieser Klasse muss nur die immer gleich zu verwendenden Methoden aufrufen und braucht sich nicht um deren genaue Funktionsweise zu kümmern. Diese Art der Programmierung, bei der nach außen in Gestalt der Methoden eine *Programmierschnittstelle* (engl. *programming interface*) zur Verfügung gestellt wird, macht die Programme damit *robuster*, das heißt, weniger anfällig für Änderungen.

Ein zweiter Faktor, der zur Robustheit der objektorientierten Programmierung beiträgt, ist, dass die Methoden natürlich sicherstellen können, dass *nur zulässige Operationen* ausgeführt werden. Angenommen, unserer Programmierer würde dem Attribut **Gartenstuhl.preis** den Wert -10.99 zuweisen wollen. Wenn er dem Attribut **preis** Werte einfach zuweisen kann, könnte er ihm auch einen negativen Preis zuweisen. Das könnte aber an späterer Stelle im Programm ungünstige Auswirkungen haben, spätestens dann, wenn der Kunde einen negativen Rechnungsbetrag „begleichen“ soll und damit letztlich gar eine Erstattung bekäme. Hier können nun die Methoden ihre Stärken ausspielen: Unsere Methode **setzePreis()** könnte überprüfen, ob der Preis, der ihr als Argument übergeben wird, größer als 0 ist. Ist er das, würde das Attribut **preis** auf diesen Wert gesetzt. Andernfalls, also bei einem negativen Preis, würde das Attribut auf den Wert 0 gesetzt. So würde die Methode verhindern, dass versehentlich unzulässige, also negative Preise gesetzt werden. Indem die Methode den Preis validiert, trägt sie zur Stabilität des Programms bei; anders ausgedrückt: Es ist nicht mehr so leicht, das Programm „aus der Fassung“ zu bringen, es wird robuster gegenüber fehlerhaften Dateneingaben.

Eine besondere Methode, die in praktisch allen objektorientierten Sprachen existiert, ist der sogenannte *Konstruktor*. Der Konstruktor wird automatisch aufgerufen, wenn eine neue Instanz der Klasse erzeugt wird. Er kann zum Beispiel dazu verwendet werden, bestimmte wichtige Attribute der Klasse zu initialisieren, entweder mit Standardwerten oder mit Werten, die der Konstruktor-Methode als Argument übergeben werden. Würden wir einen solchen Konstruktor in unsere Klasse **Produkt** mit einbauen, könnte unsere Klassendefinition etwa so aussehen:

```
Klasse Produkt
Beginn
    bezeichnung : Zeichenkette
    beschreibung : Zeichenkette
    artikelnummer : Ganzzahl
    hersteller : Zeichenkette
    preis : Fliesskommazahl
    setzePreis(neuerPreis: Fliesskommazahl)
    Produkt(startpreis : Fliesskommazahl, name : Zeichenkette)
Ende
```

Neu ist, dass die Klasse eine Konstruktor-Methode **Produkt()** besitzt. Sie heißt genauso wie die Klasse selbst und übernimmt zwei Argumente, einen Preis und einen Produktnamen. Mit diesen beiden Daten könnte der Konstruktor jetzt die Attribute **preis** und **bezeichnung** initialisieren, wenn eine neue Instanz dieser Klasse erzeugt wird. Dafür muss der Konstruktor bei erzeugen der Instanz natürlich mit den beiden Argumenten aufgerufen werden; das könnte zum Beispiel so aussehen:

```
Gartenschaufel : Produkt(10.99, "Gartenschaufel, Edelstahl")
```

Wie Sie sehen, deklarieren wir hier – wie bereits in den vorangegangenen Beispielen – eine Variable vom Typ **Produkt**, dieses Mal allerdings wird dazu der Konstruktor aufgerufen und zwar mit den notwendigen Parametern, dem Preis und der Bezeichnung.

### 11.7.5 Polymorphismus

Der Begriff mag wie eine Krankheit klingen, aber *Polymorphismus* ist keineswegs ein negatives Phänomen, sondern im Gegenteil eine sehr praktische Möglichkeit, die objektorientierte Programmierung bietet.

Polymorphismus hängt eng mit dem Konzept der Vererbung zusammen. Sie erinnern sich, dass Klassen ihre Methoden und Attribute an abgeleitete Klassen „vererben“, also weitergeben können. In den vorangegangenen Abschnitten hatten wir eine Klasse **Buch** definiert, die alle Eigenschaften und Methoden der allgemeineren Klasse **Produkt** erbt und zusätzliche noch eigene Eigenschaften und Methoden besitzen kann, die in der „Elternklasse“ **Produkt** nicht verfügbar sind.

Nun könnten wir eine Methode definieren, die die Eigenschaften des Produkts anzeigt, also eine Art Produktsteckbrief erzeugt. Diese Methode könnten wir in die allgemeine Klasse **Produkt** stecken. Sie wäre dank der Vererbung auch für die von **Produkt** abgeleitete Klasse **Buch** verfügbar. Allerdings würden bei der Anzeige die speziellen Eigenschaften von Büchern, etwa der Autor oder die Seitenzahl, die beide Attribute der Klasse **Buch** sind, unberücksichtigt bleiben. Diese Eigenschaften sind nur Bestandteil der Klasse **Buch**, nicht aber der Klasse **Produkt**, weshalb eine Anzeige-Methode, die wir in der Klasse **Produkt** unterbringen, auf diese Eigenschaften natürlich nicht zurückgreifen kann. Die Gartenschaufel aus den vorangegangenen Beispielen, eine Instanz der allgemeinen Klasse **Produkt**, hat eben keine Seitenzahl!

Praktisch wäre es aber doch, wenn wir eine Anzeige-Methode hätten, die einfach für jedes Produkt immer die richtige Anzeige liefert, ganz gleich, mit welcher Art von Produkt wir es zu tun haben. Idealweise würden wir einfach die Anzeige-Methode, nennen wir sie **produktAnzeigen()**, aufrufen und sie würde sich darum kümmern, für jede Art von Produkt die richtige Anzeige auf dem Bildschirm auszugeben.

Genau das erlaubt Polymorphismus. Polymorphismus bedeutet, dass Klassen, die von einander abgeleitet sind, Methoden gleichen Namens haben können, die

## 11.7 · Objekte

aber alle etwas anderes tun. Wird die Methode dann für ein konkretes Objekt, also eine Instanz einer Klasse, aufgerufen, wird automatisch die zu der *jeweiligen Klasse* gehörende Methode ausgeführt. In unserem Beispiel würden dann eben auch die Eigenschaften Autor und Seitenzahl angezeigt werden.

Eine solche polymorphe Ausgestaltung der Methode **produktAnzeigen()** könnte so aussehen:

```
Klasse Produkt
BeginnKlasse
    bezeichnung : Zeichenkette
    beschreibung : Zeichenkette
    artikelnummer : Ganzzahl
    hersteller : Zeichenkette
    preis : Fliesskommazahl
    produktAnzeigen()
EndeKlasse

Klasse Buch Erbt Produkt
BeginnKlasse
    autor : Zeichenkette
    seitenzahl : Ganzzahl
    produktAnzeigen()
EndeKlasse
```

Wie Sie sehen, haben beide Klassen, die „Elternklasse“ **Produkt** und die abgeleitete „Kindklasse“ **Buch**, jeweils eine Funktion **produktAnzeigen()**. Welche von beiden ausgeführt wird, wenn wir die Methode aufrufen, hängt davon ab, ob das Objekt, für das wir die Methode aufrufen, eine Instanz von **Produkt** oder eine Instanz der davon abgeleiteten Klasse **Buch** ist.

Wenn wir also zwei Objekte deklarieren

```
Gartenschaufel : Produkt
Grisham1992 : Buch
```

und dann für beide Objekte jeweils die Methode **produktAnzeigen()** aufrufen,

```
Gartenschaufel.produktAnzeigen()
Grisham1992.produktAnzeigen()
```

werden letztlich zwei unterschiedliche Methoden aufgerufen; für das Objekt **Gartenschaufel** die Methode der Klasse **Produkt**, von der Gartenschaufel eine Instanz ist, für das Objekt **Grisham1992** die Methode der abgeleiteten Klasse **Buch**, sodass **produktAnzeigen()** dann auch Seitenzahl und Autor korrekt darstellen kann.

Das Praktische an den polymorphen Methoden ist, dass wir uns nicht damit beschäftigen müssen, welche Art von Objekt **Gartenschaufel** und **Grisham1992** ei-

gentlich sind. Wir rufen einfach stur die Methode **produktAnzeigen()** auf, und es geschieht stets das, was für die jeweilige Objektklasse das Beste ist; die gleichen Namen der Methoden machen es möglich.

Ein Begriff, der im Zusammenhang mit Polymorphismus immer wieder auftaucht, ist das *Überladen*. Man spricht davon, dass die Methode **produktAnzeigen()** der Klasse **Produkt** überladen wird, indem von **Produkt** abgeleitete Klassen ihre jeweils eigene Methode **produktAnzeigen()** mitbringen, um ihre Spezifika optimal zu berücksichtigen. Beide Begriffe treffen den Sachverhalt eigentlich gut: Während „Polymorphismus“ darauf abstellt, dass (scheinbar) ein und dieselbe Methode viele (*poly*) Gestalten (*morphía*) haben kann, beschreibt „Überladen“ den Vorgang, durch den dieselbe Funktion mehrfach mit anderer Bedeutung versehen wird.

### 11.7.6 Zugriffsrechte

Zum Abschluss unserer Betrachtung der objektorientierten Programmierung schauen wir uns eine letzte Eigenschaft an, die nochmal die Motivation objektorientierter Programmierung betont, die Entwicklung und die Verwendung der Klassen strikt voneinander zu trennen.

Es gibt nämlich eine Möglichkeit, den Zugriff auf Attribute und Methoden von Klassen zu beschränken. Die konkrete Ausgestaltung kann von Programmiersprache zu Programmiersprache variieren, aber in der Regel gibt es zumindest die folgende Abstufung von Zugriffsrechten:

- **Privat** (engl. *private*): Methoden und Attribute, die unter dieser Zugriffsbeschränkung stehen, können nur von Methoden derselben Klasse verwendet werden. Sie sind nach außen nicht „sichtbar“; als Benutzer der Klasse, der die Klasse in seinen eigenen Programmen verwendet, könnten Sie auf diese Methoden und Attribute nicht zugreifen. Sehr wohl könnte aber eine Methode, die Sie aufrufen (die also selbst nicht privat ist), mit diesen Methoden und Attributen arbeiten. Sie können das aber direkt nicht tun. Die privaten Methoden und Attribute werden also nach außen abgeschirmt. Der Zugriffsschutz privat eignet sich also gut, um zum Beispiel Hilfsvariablen oder Hilfsmethoden zu definieren, die nicht von außen aus aufgerufen werden, sondern lediglich von anderen Methoden der Klasse verwendet werden sollen.
- **Geschützt**(engl. *protected*): Methoden und Attribute, die als *protected* deklariert worden sind, können von der Klasse selbst, zu der sie gehören, verwendet werden, und auch von abgeleiteten Klassen, nicht aber vom Programmierer, der mit diesen Klassen in seinem Programm arbeitet.
- **Öffentlich/offen** (engl. *public*): Auf Methoden und Attribute mit der Zugriffsbeschränkung *public* kann von überall her zugegriffen werden, aus der eigenen Klasse heraus, aus abgeleiteten Klassen heraus und auch durch den Anwender der Klasse.
- Betrachten Sie zur Verdeutlichung die folgende Erweiterung unseres Beispiels:

```
Klasse Produkt
Beginn
Offen
    bezeichnung : Zeichenkette
    beschreibung : Zeichenkette
    artikelnummer : Ganzzahl
    hersteller : Zeichenkette
    setzePreis()

Privat
    preis : Fliesskommazahl
Ende
```

Hier haben wir das Attribut **preis** als eine private Eigenschaft deklariert. Die Methode **setzePreis()** ist hingegen eine öffentliche Methode. Wir wollen als Entwickler der Klasse also nicht, dass jemand unser Attribut **preis** direkt bearbeitet. Deshalb schützen wir es als privat. Eine Methode aus derselben Klasse kann jedoch darauf zugreifen und seinen Wert verändern. **setzePreis()** ist so eine Methode. Sie ist eine öffentliche Methode, die auch von außerhalb der Klasse aus aufgerufen werden kann. Ein Programmierer, der unsere Klasse verwendet, könnte nun also über die Schnittstellen-Methode **setzePreis()** das Attribut **preis** bearbeiten, nicht aber direkt, zum Beispiel durch eine Wertzuweisung.

Auf diese Weise lässt sich also sehr einfach steuern, welche Teile von Klassen nach außen sichtbar sein und als Schnittstelle, als Interface zu den Funktionalitäten der Klasse dienen sollen, und welche nicht.

### ?

#### 11.3 [5 min]

Sind die folgenden Aussagen richtig oder falsch?

- Objektorientierte Programmierung ist der Versuch, eine möglichst „natürliche“ Abbildung von Dingen in der realen Welt zu erreichen.
- Eine Methode ist eine Funktion, die zu einer Klasse gehört und die Attribute dieser Klasseninstanz ändern kann.
- Alle Attribute einer Klasseninstanz sind aus dem Programm heraus direkt durch Zuweisung veränderbar.
- Die Verwendung objektorientierter Programmierung macht das Programm zwar übersichtlicher, erschwert aber Anpassungen am Programm.
- Vererbung bedeutet, dass man die Definition einer Klasse in unterschiedlichen Programmen wiederverwenden kann.

### ?

#### 11.4 [3 min]

Beschreiben Sie den Unterschied zwischen einer Klasse und einer Instanz.

### ?

#### 11.5 [3 min]

Was sind die wesentlichen Elemente einer Klassendefinition?

### ?

#### 11.6 [3 min]

Warum ist Polymorphismus ein nützlicher Ansatz in der objektorientierten Programmierung?

## 11.8 Ihr Fahrplan zum Erlernen einer neuen Programmiersprache

---

### ■ Wenn Sie eine neue Programmiersprache lernen ...

finden Sie heraus,

- ob die Sprache bei den Bezeichnern, also den Namen von Variablen zwischen Groß- und Kleinschreibung unterscheidet (das heißt, case-sensitive ist),
- welche grundlegenden Datentypen die Sprache anbietet (insbesondere Zahlen, Zeichenketten, Wahrheitswerte),
- ob man Variablen deklarieren muss, und wenn ja, wie man das macht
- wie man Variablen Werte zuweist,
- ob, und wenn ja, wie man Variablen zwischen Datentypen (explizit) konvertieren kann und welche Konversionen die Programmiersprache ggf. bereits selbst (implizit) vornimmt,
- ob die Sprache Felder von gleichartigen Variablen (Arrays) unterstützt, und wenn ja, wie man Felder anlegt und auf ihre Elemente zugreift (insbesondere: ob die Indizierung der Feldelemente bei 0 oder bei 1 beginnt),
- ob die Sprache assoziative Felder kennt, auf deren Werte mit Schlüsseln zugegriffen werden kann, und wenn ja, wie man diese Felder anlegt und ihre Elemente adressiert,
- welche weiteren komplexen Datentypen in der Programmiersprache gebräuchlich sind,
- ob die Sprache objektorientierte Programmierung unterstützt und falls ja, wie man auf Klassen-Attribute und -Methoden zugreift, und wie man selbst Klassen definiert, insbesondere, wie man Klassen von bestehenden Klassen ableitet.

## 11.9 Lösungen zu den Aufgaben

---

### ■ Aufgabe 11.1

In den Sprachen, die eine Variablen Deklaration notwendig machen, wird die Variablen durch die Deklaration beim Interpreter/Compiler angemeldet; er reserviert den notwendigen Speicher und weist der Variable gegebenenfalls einen initialen Wert zu. Danach kann die Variable im Programm verwendet werden. Beim Deklarieren wird der Bezeichner (Name) der Variable und in manchen Sprachen auch ihr Datentyp festgelegt.

### ■ Aufgabe 11.2

Die Notwendigkeit, eine Variable zu deklarieren, erlaubt es dem Interpreter/Compiler, auf die Verwendung nicht deklarerter Variablen hinzuweisen. Weil nicht deklarierte Variablen häufig durch Tippfehler im Programmcode entstehen, wird so verhindert, dass versehentlich eine neue Variable erzeugt und mit dieser gearbeitet wird, wohingegen jene Variable, auf die eigentlich hätte zugegriffen werden sollen, gänzlich unverändert bleibt. Der Programmcode wird also durch den Zwang, Va-

riablen zu deklarieren, robuster. Ähnliches gilt auch, wenn beim Deklarieren bereits der Typ festgelegt wird, und dieser Typ im Nachhinein nicht mehr geändert werden kann. In diesem Fall kann der Interpreter/Compiler einen Fehler melden, wenn der Variable versehentlich ein Wert von einem „unpassendem“ Typ zugewiesen wird. Auch dies vermeidet Fehler und macht den Programmcode robuster.

#### ■ Aufgabe 11.3

- a. Richtig.
- b. Richtig.
- c. Falsch. In vielen Programmiersprachen können Attribute von Klasseninstanzen gegen den Zugriff von außen abgeschirmt werden, indem sie als „privat“ definiert werden. Diese Attribute können dann nur von Methoden der Klasse bearbeitet werden, sind aber nach außen praktisch unsichtbar und können daher vom Programmierer nicht direkt angesprochen werden.
- d. Falsch. Objektorientierte Programmierung trägt dazu bei, dass Programmlemente voneinander unabhängiger werden. Weil der Programmierer die Klasseninstanz nur über die definierten Methoden (und ggf. durch direkten Zugriff auf die Attribute) anspricht, muss ihn die innere Funktionsweise der Klasse nicht weiter interessieren. Solange also die *Schnittstelle* der Klasse nach außen unverändert bleibt, kann der Entwickler der Klasse selbst diese intern nach Belieben verändern und die auf der Klasse basierenden Programme bleiben syntaktisch korrekt. Anpassungen am Code werden durch diese stärkere Modularisierung erleichtert.
- e. Falsch. Vererbung bedeutet, dass sich von einer Klasse weitere Klassen ableiten lassen, die deren Methoden und Attribute „erben“. Auf diese Weise lässt sich eine Klasse elegant erweitern, insbesondere für speziellere Verwendungen ausdetaillieren.

#### ■ Aufgabe 11.4

Die Klasse ist die abstrakte Definition eines Objekts (oder besser: Objektstyps) mit den zu Objekten dieser Art gehörenden Attributen und Methoden und fungiert wie eine Schablone. Nach dieser Schablone werden die konkreten Objekte, die Klasseninstanzen, geformt und besitzen daher als Abbilder der Klasse deren Methoden und Attribute.

#### ■ Aufgabe 11.5

Im Wesentlichen besteht eine Klassendefinition aus dem Bezeichner (Namen) der Klasse und den zur Klasse gehörenden Attributen und Methoden. Diese können durch entsprechende Schlüsselwörter mit Einschränkungen der Zugriffsrechte versehen werden (vgl. ► Abschn. 11.7.6). Ist die Klasse von einer anderen abgeleitet, ist der Verweis auf die „Elternklasse“ ebenfalls Bestandteil der Klassendefinition (vgl. ► Abschn. 11.7.3).

#### ■ Aufgabe 11.6

Polymorphismus erlaubt es, dass eine bestimmte Methode von Objekten unterschiedlicher Typen (Klassen) angeboten wird. Dadurch ist es möglich, die Methode

auf die Spezifika der jeweiligen Klasse anzupassen. Insbesondere ist dies interessant, wenn durch Vererbung eine Klassenhierarchie entsteht. Wird nun für die Instanz einer Klasse aus dieser Klassenhierarchie die Methode aufgerufen, kommt dabei die spezielle Implementierung der Methode für die Klasse der Objektinstanz zum Zuge, und nur, wenn diese Klasse keine besondere Implementierung der Methode besitzt, die gleichnamige Methode der nächst höheren Klasse. Auf diese Weise wird sichergestellt, dass stets die möglichst gut an die Besonderheiten der jeweiligen Klasse angepasste Methode verwendet wird, nötigenfalls aber Methoden zum Tragen kommen, die zu Klassen gehören, die weiter oben in der Klassenhierarchie angesiedelt sind. Unterschiedliche Objekt-Typen können damit unterschiedlich behandelt werden, aber nach außen dieselbe Schnittstelle (nämlich die jeweils stets gleichnamige Methode) anbieten. Der Programmierer muss sich nicht mit der Frage beschäftigen, die Methode welcher Klasse er nun eigentlich aufrufen soll; er ruft einfach die Methode für seine Objektinstanz auf und der Interpreter/Compiler klärt für, welche Methode nun genau zum Zuge kommen soll.



# Wie lasse ich Daten ein- und ausgeben?

## Inhaltsverzeichnis

- 12.1 Formen der Datenein- und -ausgabe – 124**
- 12.2 Grafisch oder nicht grafisch – das ist hier die Frage – 125**
  - 12.2.1 Grafische Benutzeroberflächen – 127
  - 12.2.2 Konsolenanwendungen – 135
- 12.3 Arbeiten mit Dateien – 139**
- 12.4 Arbeiten mit Datenbanken – 145**
- 12.5 Ihr Fahrplan zum Erlernen einer neuen Programmiersprache – 148**
- 12.6 Lösungen zu den Aufgaben – 149**

## Übersicht

Nachdem wir im letzten Kapitel der Frage nachgegangen sind, wie Daten mit Hilfe von Variablen so vorgehalten werden können, dass wir im Programm mit ihnen arbeiten können, wenden wir uns nun der Frage zu, wie wir überhaupt Daten von außen in das Programm hineinbringen (wir sprechen in diesem Zusammenhang in einem weiteren Sinne von „Dateneingabe“) und auch wieder aus dem Programm herausbekommen können (im weiteren Sinne „Datenausgabe“). Dabei geht es sowohl um Dateneingabe und Datenausgabe direkt vom/an den Benutzer, als auch um Eingabe und Ausgabe im Kontext von Dateien. In diesem Kapitel beschäftigen wir uns also damit, wie ein Programm mit seiner „Außenwelt“ kommuniziert.

In diesem Kapitel werden Sie folgendes lernen:

- was die beiden zentralen Grundmodi der direkten Datenein- und -ausgabe durch bzw. an den Benutzer, grafische Benutzeroberflächen und Konsolenanwendungen, voneinander unterscheidet, und wann welche Form vorzuziehen ist
- welches die wichtigsten Bedienelemente auf grafischen Benutzeroberflächen sind
- wie Sie grafische Benutzeroberflächen gestalten
- wie Sie Daten aus Dateien auslesen und in Dateien schreiben
- wie die Arbeit mit Datenbanken in den Grundzügen funktioniert.

### 12.1 Formen der Datenein- und -ausgabe

12

Die meisten Programme kommunizieren auf die eine oder andere Weise mit auf ihrer „Umwelt“. Zur „Umwelt“ gehört natürlich zuvorderst der Benutzer des Programms, der Informationen zur Verfügung stellt und Entscheidungen trifft. Teil der Umwelt sind aber auch andere Gegenstände und Phänomene, deren Eigenschaften und Zustände Einfluss auf den Ablauf des Programms haben. Sprechen wir beispielsweise über eine Software, die proaktiv Ihre Heizung hochfahren soll, wenn es beginnt, kalt zu werden, dann sind aktuelle (und ggf. auch prognostizierte) Temperaturen Teil der relevanten Umwelt des Programms. Die Informationen über diese Umwelt kann das Programm natürlich nur verarbeiten, wenn sie ihm in Form von Daten bekannt gemacht werden. Dieses „Bekanntmachen“ wollen wir in diesem Kapitel etwas genauer beleuchten. Es geht also um die Frage, wie Informationen in Form von Daten in das Programm „eingegeben“ werden können.

Beim Begriff „eingeben“ denkt man wohl als erstes an die direkte Eingabe durch den Benutzer, und dabei vor allem an die Eingabe mit Hilfe einer Tastatur. Aber natürlich ist das bei weitem nicht die einzige Form, in der der Benutzer dem Programm Daten zur Verfügung stellen kann. Andere Eingabegeräte, für die Mikrofon, Webcam, Maus, Joystick oder Touchscreen nur einige Beispiele sind, erlauben es, ganz unterschiedliche Arten von Daten – zum Beispiel Sound-, Video-, Positions-, Richtungs- und Geschwindigkeitsdaten – einzugeben.

## 12.2 · Grafisch oder nicht grafisch – das ist hier die Frage

Doch nicht alle Eingaben gehen ganz unmittelbar vom Benutzer aus. Als Quelle von Dateneingaben (und wir wollen die Begriffe „Eingabe“ und „Ausgabe“ hier in eben diesem weiteren Sinne verstehen), mit denen ein Programm arbeitet, kommen auch Dateien und Datenbanken in Frage. Das Programm zur Heizungssteuerung zum Beispiel wird vielleicht über einen Webservice Daten aus der Datenbank eines meteorologischen Dienstes abfragen, um dann zu ermitteln, ob die Heizung angefahren werden muss, und, wenn ja, wann und wie stark.

Umgekehrt stehen vielfältige Möglichkeiten zur Verfügung, Daten wieder *auszugeben*. Beispiele dafür sind vor allem die Ausgabe auf dem Bildschirm und das Schreiben von Daten in Dateien und in Datenbanken.

In diesem Kapitel werden wir uns mit drei Arten von Datenein- und -ausgaben in Programmen beschäftigen – der Eingabe bzw. Ausgabe durch bzw. an den Benutzer über eine wie auch immer geartete *Benutzeroberfläche*, sowie die Arbeit mit *Dateien* und *Datenbanken*. Da letztere keine ganz triviale Angelegenheit ist und in der Regel die Kenntnis einer eigenen Abfrage-(Programmier-)Sprache, die speziell zu diesem Zweck entwickelt worden ist, voraussetzt, werden wir diesen Themenbereich hier zwar nur überblickartig behandeln, aber doch wenigstens so, dass Sie ein solides Grundverständnis für dieses in der Praxis der modernen Softwareentwicklung enorm wichtige Feld entwickeln.

### 12.2 Grafisch oder nicht grafisch – das ist hier die Frage

---

Spätestens seit dem Aufkommen der Betriebssysteme mit grafischen Benutzeroberflächen und dem Siegeszug des World Wide Webs sind wir es gewohnt, Daten auf ansprechend gestalteten grafischen Oberflächen (engl. *graphical user interfaces*, *GUIs*, oder, weil graphisch heute ohnehin der Standard ist, einfach *UIs*) einzugeben und zu betrachten. Was „ansprechend“ dabei bedeutet, ist dabei sicherlich abhängig vom Geschmack und den technischen Möglichkeiten der jeweiligen Zeit.

Grafischen Benutzeroberflächen aller Zeiten und aller technischen Spielarten gemein ist aber, dass sie versuchen, dem Benutzer die Eingabe von Daten durch geeignete *Bedienelemente* möglichst bequem zu machen; zum Beispiel, indem Zahlen aus einem vorgegebenen Wertebereich (etwa Lautstärken oder Farbintensitäten) nicht einfach über die Tastatur eingegeben werden müssen, sondern bequem durch Schieberegler festgelegt werden können.

Insbesondere im Zuge der zunehmenden Bedeutung von Mobil- und Webanwendungen haben sich ganz neue Berufsbilder entwickelt, die des *User-Interface-Designers* (*UI-Designers*) und des *User-Experience-Designers* (*UX-Designers*). Während der UI-Designer die Oberfläche technisch gestaltet und mit dem dahinterliegenden Programmcode „verdrahtet“, beschäftigt sich der UX-Designer intensiv mit den Endanwendern und deren Verhaltensweisen. Er ist gewissermaßen der Marktforscher des UI-Designers und versucht, die für die Endanwender optimale Art der Interaktion mit der Anwendung zu finden. Diese kann der UI-Designer dann in der Benutzeroberfläche umsetzen. Ist der UI-Designer also eher nach „in-

nen“, in Richtung der technischen Gestaltung, tätig, so wendet sich die Arbeit des UX-Designers nach „außen“ und besteht maßgeblich daraus, die Endanwender und ihre Arbeitsweise zu verstehen und dieses Wissen für die Oberflächengestaltung nutzbar zu machen.

Wenn Sie nicht beruflich mit dem Programmieren beschäftigt sind, werden Sie in der Regel Entwickler, UI-Designer und UX-Designer in einer Person sein. Das ist zwar mehr Arbeit, bedeutet aber auch, dass Sie beim Gestalten Ihrer Oberflächen Ihrer Kreativität freien Lauf lassen können; dennoch werden Sie natürlich die Bedürfnisse und Wünsche Ihrer Nutzer berücksichtigen müssen, es sei denn, Sie entwickeln die Software ausschließlich für sich selbst.

Mit dem Triumphzug der grafischen Benutzeroberflächen sind im Endanwenderbereich die *Konsolenanwendungen* beinahe vollkommen aus der Mode gekommen, an die sich derjenige, vielleicht noch mit wohliger Schauer erinnern mag, der in seiner Computer-Anfangszeit mit Betriebssystemen wie MS-DOS gearbeitet hat oder heute ein Linux-System betreibt, das er nicht ausschließlich über eine der grafischen Frontends für Linux bedient. Konsolenwendungen bieten nur eine Form der direkten Eingabe durch den Benutzer an, nämlich die über die Tastatur.

Ein zentraler Unterschied besteht aber nicht nur darin, wie die Oberfläche optisch und in Bezug auf die Bedienfreundlichkeit daherkommt. Auch der Ablauf des Programms ist bei den Konsolenanwendungen meist ein ganz anderer als bei Anwendungen mit grafischen Benutzeroberflächen. Angelehnt an Karl Marx' berühmtes Diktum, das Sein bestimme das Bewusstsein, könnte man sogar sagen: „Die Oberfläche bestimmt das Programmieren“.

Denn Konsolenanwendungen sind normalerweise *lineare Programme*, sie laufen Schritt für Schritt ab: Zum Beispiel wird zunächst etwas angezeigt (etwa: „Bitte geben Sie Ihren Benutzernamen ein“); dann wartet das Programm auf eine Benutzereingabe; sobald der Benutzer seine Eingabe gemacht und mit <ENTER> oder <RETURN> bestätigt hat, folgt die nächste Ausgabe des Programms („Bitte geben Sie Ihr Passwort ein“); das Programm wartet wieder, bis der Benutzer seine Eingabe gemacht und bestätigt hat; dann verarbeitet das Programm die Eingabe (prüft zum Beispiel Benutzername und Passwort auf Gültigkeit) und reagiert wieder mit einer Ausgabe („Zugang gewährt.“), und so weiter.

Anders dagegen bei einer grafischen Benutzeroberfläche: Hier gibt das Programm im Regelfall keine exakte Reihenfolge der Benutzeraktionen vor. Der Benutzer könnte in unserem Beispiel auch zuerst das Passwort und erst dann den Benutzernamen eingeben. Erst ein Klick auf den Login-Button löst die Prüfung der Benutzereingaben durch das Programm aus. Noch deutlicher wird der Unterschied, wenn Sie eine grafische Benutzeroberfläche wie die eines Textverarbeitungsprogramms betrachten, bei dem der Benutzer über Schaltflächen eine Vielzahl von Funktionen ansteuern oder sich stattdessen auch erst einmal der Arbeit am Text des aktuell geöffneten Dokuments widmen kann. Diese Art, mit dem Programm zu arbeiten, ist *nicht linear*. Stattdessen beobachtet das Programm, was der Benutzer tut und reagiert auf *Ereignisse*, etwa den Klick auf eine Schaltfläche oder die Auswahl einer Funktion aus einem Menü. Programme mit grafischen Benutzerober-

flächen sind also typischerweise *ereignisgesteuert*. Löst der Benutzer ein bestimmtes Ereignis aus, springt das Programm an diejenige Stelle im Programmcode, wo beschrieben ist, was zu tun ist, wenn dieses Ereignis eintritt. Löst der Benutzer danach ein ganzes anderes Ereignis aus, springt das Programm wiederum an die richtige Stelle, ganz egal, wo genau im Quelltext des Programms dieser Code-Abschnitt auch zu finden sein mag. Anders als die Konsolenanwendung, die strikt eine Zeile Code nach der anderen ausführt und aus einer linear ablaufenden Folge von Anweisungen besteht, „springt“ die Verarbeitung bei der ereignisgesteuerten Programmierung von einem Block von Anweisungen zu einem irgendwo anders liegenden Block, je nachdem, was der Benutzer gerade tut.

In ► Kap. 14, wenn es darum geht, wie wir Programme auf die Eingaben des Benutzers reagieren lassen, werden wir uns mit ereignisgesteuerten Programmen noch etwas genauer befassen. In diesem Kapitel beschäftigen wir uns erst mal mit den Möglichkeiten der Dateneingabe, also der Oberfläche als solcher.

## 12.2.1 Grafische Benutzeroberflächen

### 12.2.1.1 Wichtige Bedienelemente

Die Bedienelemente, die auf grafischen Benutzeroberflächen verwendet werden, mögen unterschiedlich aussehen, je nach Betriebssystem (zum Beispiel *Windows*, *MacOS*, *Android*, *iOS*) und Plattform (Computer, Tablet, Smartphone); viele Elemente gibt es aber in praktisch allen Betriebssystemen und auf praktisch allen Plattformen.

Im Folgenden wollen wir uns die wichtigsten Bedienelemente und ihre zentralen Eigenschaften kurz ansehen. Neben den hier angesprochenen Eigenschaften haben die Bedienelemente natürlich noch eine Vielzahl anderer Charakteristika. Einige davon sind spezifisch für das jeweilige Bedienelement, andere sind allen oder doch zumindest den allermeisten Bedienelementen gemein. Zu den letzteren zählen unter anderem die Position auf der Oberfläche, die Ausmaße (Höhe und Breite), die Sichtbarkeit (ist das Bedienelement aktuell sichtbar oder ist es ausgeblendet?), die Benutzbarkeit (kann der Benutzer das Bedienelement aktuell verwenden, ist es also aktiv, oder ist es inaktiv und damit gesperrt und „ausgegraut“?), sein Name (damit es im Programm angesprochen werden kann) und seine Farbe.

#### ■ Edit-Felder

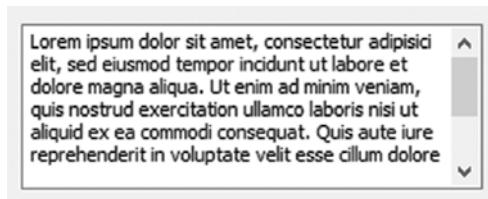
Zur klassischen Eingabe über die Tastatur stellen grafische Benutzeroberflächen die Edit-Felder bereit. Diese erlauben, je nach Typ bzw. Einstellung, die ein- oder mehrzeilige Eingaben von Informationen (► Abb. 12.1, 12.2, und 12.3).



► Abb. 12.1 Beispiel für ein Eingabefeld



■ Abb. 12.2 Beispiel für ein Eingabefeld



■ Abb. 12.3 Beispiel für ein Eingabefeld

Edit-Felder können typischerweise umfassend konfiguriert werden oder es stehen unterschiedliche Edit-Felder mit unterschiedlichen Eigenschaften zur Verfügung. So kann zum Beispiel die Passwort-Eingabe des Benutzers verdeckt werden, indem statt der eingegebenen Zeichen einfach gar nichts oder ein bestimmtes anderes Zeichen dargestellt wird. Auch können die eingegebenen Informationen oftmals bereits während der Eingabe validiert werden. Auf diese Weise kann man den Benutzer zum Beispiel zwingen, eine numerische Eingabe zu machen; würde er stattdessen Buchstaben eingeben, würde dieser Text überhaupt nicht als Eingabe vom Edit-Feld entgegengenommen werden. Mitunter ist es wichtig, dass unterschiedliche Teile der Eingabe unterschiedlich formatiert sind (zum Beispiel bei der Eingabe von Programmcode, der mit Syntax Highlighting dargestellt werden soll); dann benötigt man ein Eingabefeld, das mit verschiedenen Textformatierungen umgehen kann.

12

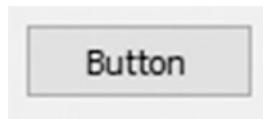
### ■ Buttons

Buttons sind Schaltflächen, durch die der Benutzer üblicherweise eine Aktion auslöst, zum Beispiel das Speichern eines Dokuments oder das Absenden einer Nachricht. Wichtigste Eigenschaft des Buttons ist sicherlich seine Beschriftung und die Verknüpfung zu dem Programmteil, der ausgeführt wird, wenn der Button geklickt wird. Die optische Erscheinung wird zudem oft durch ein Symbolbild dominiert, das die durch den Button ausgelöste Aktion stilisiert darstellt (■ Abb. 12.4).

### ■ Menüs

Ähnlich wie Buttons dienen Menüs dazu, den Benutzer Aktionen auslösen zu lassen. Dabei steht die Auswahl unter verschiedenen Optionen in Form der Menüeinträge im Vordergrund. Das erklärt auch den Namen des Bedienelements, das der Benutzer ähnlich verwendet wie die Speisekarte in einem Restaurant. Wichtige Eigenschaften sind – wiederum analog zum Button – neben den Bezeichnungen des

## 12.2 · Grafisch oder nicht grafisch – das ist hier die Frage



■ Abb. 12.4 Beispiel für ein Button



■ Abb. 12.5 Beispiel für ein Menü

Menüs und seiner Einträge natürlich die Aktionen, die der Benutzer durch Klick auf die Einträge auslösen kann (■ Abb. 12.5).

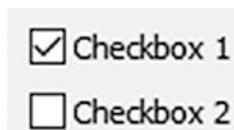
### ■ Checkboxen und Radiobuttons

Checkboxen und Radiobuttons sind Bedienelemente, die es dem Benutzer der Oberfläche erlauben, eine Auswahl zwischen mehreren Optionen zu treffen. Dabei löst – anders als bei Buttons und Menüs – der Klick auf eines dieser Elemente normalerweise nicht unmittelbar eine Aktion aus. Stattdessen werden Checkboxen und Radiobuttons meist dazu verwendet, Einstellungen festzulegen, die das genaue Verhalten von Aktionen steuern, die der Benutzer später anderweitig auslöst, zum Beispiel eben durch Klick auf einen Button oder einen Menüeintrag. So könnte der Benutzer etwa mit einem Radiobutton auswählen, ob er eine Datei schreibgeschützt öffnen möchte oder nicht. Die eigentliche Aktion, das Öffnen der Datei, wird erst später durch den Klick auf einen Button „Datei öffnen“ ausgelöst. Was ein Klick auf diesen Button aber genau bewirkt (ob die Datei nämlich im Nur-Lese-Modus geöffnet wird oder so, dass sie auch verändert werden kann), wird durch die zuvor mit Hilfe des Radiobuttons getroffene Einstellung bestimmt (■ Abb. 12.6 und 12.7).

Checkboxen treten oft zu mehreren auf, Radiobuttons eigentlich immer. Der Unterschied zwischen den üblicherweise eckigen Checkboxen und den runden Radiobuttons besteht darin, dass aus einer Gruppe solcher Bedienelemente im Fall von Radiobuttons *nur eine einzige Option* ausgewählt werden kann, bei Checkboxen aber *mehrere* anklickbar sind.

### ■ Toggle Buttons

Ähnlich wie Checkboxen erlauben es auch Toggle Buttons (engl. *to toggle*: umschalten) eine Option an- oder abzuschalten. Anders als bei den Checkboxen sind es allerdings keine Häkchen oder Farbfüllungen, die anzeigen, ob die Option derzeit ausgewählt ist oder nicht. Stattdessen ist die Darstellung einem Schiebeschalter-

**Abb. 12.6** Beispiele für Checkboxen**Abb. 12.7** Beispiele für Radiobuttons**Abb. 12.8** Beispiel für einen Toggle-Button

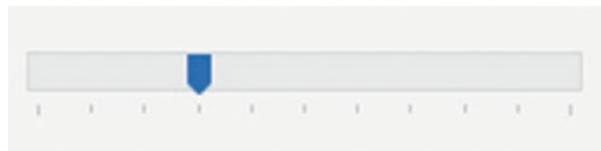
## 12

ter nachempfunden. Toggle Buttons sind mit dem Siegeszug der mobilen Betriebssysteme iOS und Android populär geworden, mittlerweile aber auch auf anderen Plattformen verfügbar (**Abb. 12.8**).

### ■ Sliders

Slider sind Schieberegler, die eine Auswahl entlang einer Skala erlauben, also zwischen Optionen, die sich anhand irgendeines Kriteriums in einer Reihenfolge anordnen lassen. Ebenso wie Radiobuttons und Checkboxen/Toggle-Buttons, löst eine Änderung der aktuellen Auswahl durch den Benutzer hier meist keine unmittelbare Aktion aus; vielmehr dienen sie in der Regel dazu, eine Einstellung vorzunehmen, die sich später auswirkt, wenn der Benutzer eine Aktion auslöst, indem er etwa auf einen Button oder einen Menüpunkt klickt (manchmal allerdings wirken sich Änderungen, die der Benutzer vornimmt, auch direkt aus, zum Beispiel, wenn der Slider zur

■ Abb. 12.9 Beispiel für einen Slider



■ Abb. 12.10 Beispiel für einen Slider

Skalierung einer Grafik verwendet wird und sich die Grafik bei einer Bewegung des Slider-Reglers automatisch aktualisiert). Ihre wichtigste Einstellung ist die Skala, das heißt, die Abstufung, in der der Benutzer den Slider-Regler einstellen kann; hier insbesondere die Zahl und Bezeichnung der Ausprägungen (■ Abb. 12.9 und 12.10).

#### ■ List Views

List Views sind „flache“, nicht-hierarchische Listen von Elementen, die es dem Benutzer erlauben, eines oder mehrere dieser Elemente auszuwählen. Ein Beispiel für die Anwendung solche List Views sind Dateimanager, die die in einem Ordner enthaltenen Dateien auflisten. Dabei können die Elemente, die im List View gezeigt werden, mit Icons versehen werden. Manchmal werden zusätzlich zur Bezeichnung des Elements (im Fall des Dateimanagers also der einzelnen Dateien) weitere Eigenschaften der Elemente in Extra-Spalten des List Views dargestellt, im Beispiel der Dateien etwa ihre Größe oder das Datum ihrer letzten Änderung. Neben der Art der Darstellung (Icons, zusätzliche Spalten für ergänzende Element-Eigenschaften) ist eine wichtige Eigenschaft, die bei der Verwendung von List Views festgelegt werden kann, ob der Benutzer immer nur ein einzelnes, oder auch mehrere Elemente auf einmal auswählen können soll (■ Abb. 12.11).

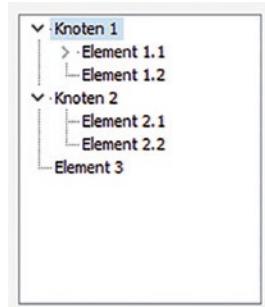
#### ■ Tree Views

Tree Views sind in dem Sinne ähnlich zu Listen, als dass sie es erlauben, mehrere Elemente darzustellen. Anders als Listen haben sie aber – wie der Name bereits suggeriert – eine baumähnliche Struktur, erlauben es also, hierarchische Beziehungen zwischen den Elementen darzustellen. Ein klassisches Beispiel für die Verwendung von Tree Views sind die Ordneransichten in Dateimanagern, bei denen die Hierarchie der Ordner typischerweise als Baumstruktur angezeigt wird. Auf diese Weise lassen sehr gut hierarchischen Zusammenhänge jeglicher Art abbilden, etwa die Struktur einer Organisation oder eine Hierarchie von Produkten, angefangen

■ Abb. 12.11 Beispiel für einen List View

Element	Eigenschaft 1	Eigenschaft 2
Eintrag 1	Eigenschaft 1.1	Eigenschaft 1.2
Eintrag 2	Eigenschaft 2.1	Eigenschaft 2.2
Eintrag 3	Eigenschaft 3.1	Eigenschaft 3.2
Eintrag 4	Eigenschaft 4.1	Eigenschaft 4.2

■ Abb. 12.12 Beispiel für einen Tree View



von breiten Produktkategorien bis hinunter zu Einzelprodukten. Anders als List Views haben Tree Views typischerweise keine zusätzlichen Spalten zur Darstellung weiterer Eigenschaften der hierarchisch angeordneten Elemente (■ Abb. 12.12).

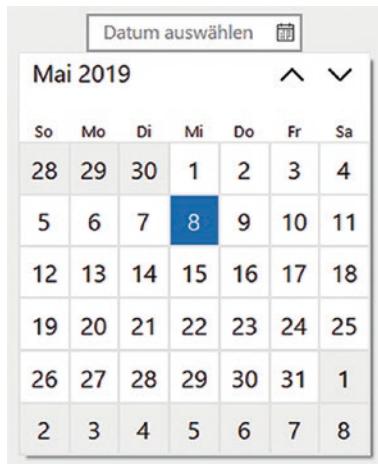
## 12

### ■ Pickers

Pickers (engl. *to pick*: auswählen) sind Bedienelemente, die es erlauben, eine Auswahl aus mehreren vorgegebenen Möglichkeiten vorzunehmen, in der Regel, ohne dass dadurch gleich irgendeine Aktion ausgelöst wird. Insofern ähneln sie zunächst den Radiobuttons, die ebenfalls eine Einstellungsauswahl aus mehreren vorgegebenen Alternativen gestatten. Die Arten und Formen, in denen Picker daherkommen, sind aber sehr unterschiedlich. Mitunter erlauben sie die Auswahl aus einem zwar vorgegebenen, aber großen und komplex aufgebauten Set von Möglichkeiten. Ein gutes Beispiel für eine solche Auswahl ist die Datumspicker, wie sie mittlerweile auf mobilen und nicht mobilen Plattformen gang und gäbe sind. Zwei ganz unterschiedliche Beispiele für solche Datumspicker sind in den Abbildungen ■ Abb. 12.13 und 12.14 dargestellt.

Natürlich muss hinter Pickern nicht zwingend eine so komplexe Auswahlssituation wie beim Datum stehen. Nicht selten werden Picker auch zur Auswahl aus ei-

■ Abb. 12.13 Beispiele für Datumspicker



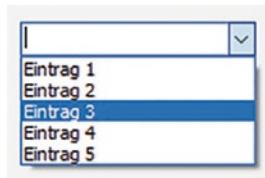
ner überschaubaren Liste von simplen Text-Optionen eingesetzt. In diesem Sinne gehört zu den Pickern auch die gute alte Combobox, wie sie ■ Abb. 12.15 zeigt, die eine nach unten aufklappende Liste von Auswahloptionen präsentiert.

### 12.2.1.2 Grafische Oberflächen entwickeln

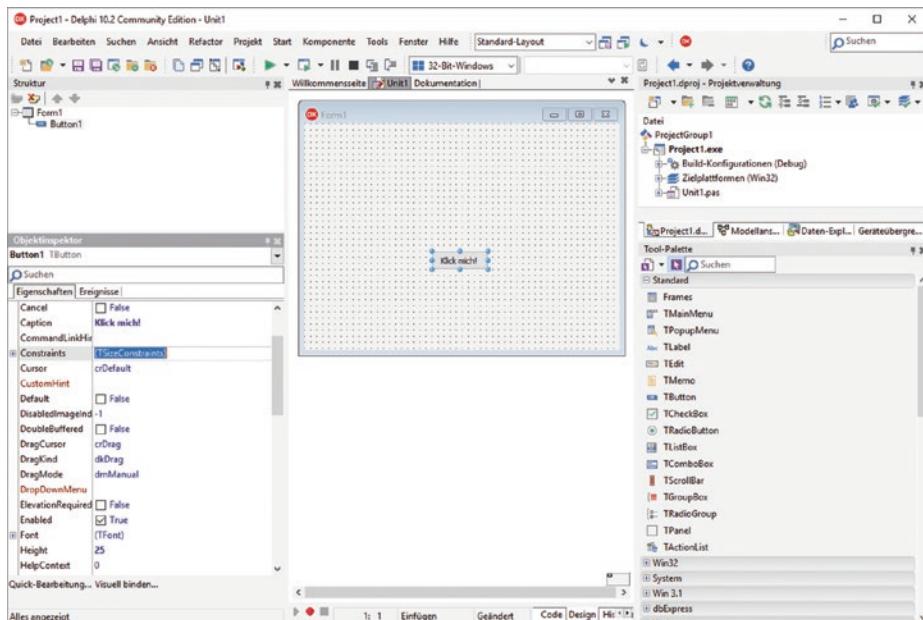
Nachdem wir uns nun einige populäre Steuerelemente grafische Benutzeroberflächen angesehen haben, stellt sich natürlich die Frage, wie genau man denn nun eine Oberfläche technisch entwickelt. Irgendwie müssen ja die Anordnung der Steuerelemente und ihre Eigenschaften definiert werden. Das beste Oberflächendesign, das Sie erdacht und vielleicht auf Papier oder mit Hilfe eines Grafik- oder Präsentationsprogramms skizziert haben, hilft Ihnen nicht, wenn Sie es nicht in eine wirklich benutzbare Programmoberfläche, eine echte GUI eben, umsetzen können.

Häufig geschieht dies mit Hilfe der Integrierten Entwicklungsumgebungen, die wir bereits in ▶ Abschn. 8.1.3 kennengelernt haben. Diese erlauben es vielfach, grafische Benutzeroberflächen ganz ohne Programmierung einfach mit der Maus „zusammenzuklicken“, in dem man die unterschiedliche Steuerelemente anwählt und dort platziert, wo man sie haben möchte.

■ Abb. 12.16 zeigt die integrierte Entwicklungsumgebung von Delphi. Hier sehen Sie, wie gerade eine grafische Benutzeroberfläche entsteht. Rechts (Bereich

**Abb. 12.14** Beispiele für Datumspicker**Abb. 12.15** Beispiel für eine Combobox

12

**Abb. 12.16** Gestaltung einer grafischen Benutzeroberfläche mit Delphi

## 12.2 · Grafisch oder nicht grafisch – das ist hier die Frage

„Tool-Palette“) lassen sich unterschiedliche Steuerelemente auswählen, die dann per Drag & Drop im Programmester platziert und ebenfalls mit der Maus in ihre Größe angepasst werden können. Im sogenannten „Objektinspektor“ links sehen die Eigenschaften des gerade ausgewählten Bedienelements, in diesem Fall des ausgewählten Buttons. Als zweite Eigenschaft können Sie hier den „Caption“, also die Beschriftung des Buttons sehen. Weitere Eigenschaften sind zum Beispiel die Form, die der Cursor annehmen soll, wenn man mit der Maus über den Button fährt („Cursor“), die Schriftart der Buttonbeschriftung („Font“) oder die Höhe des Steuerelements („Height“). Offensichtlich kann man selbst bei einem einfachen Button eine Unmenge an Einstellungen vornehmen, um Aussehen und Verhalten bis ins Detail festzulegen.

Auf dem Reiter „Ereignisse“ des Objektexplorers würden Sie eine Übersicht über die *Ereignisse* bekommen, die dieses Steuerelement auslösen kann. Das sind neben dem Klick zum Beispiel auch das Überfahren des Buttons mit der Maus. Dazu aber mehr in ► Abschn. 14.7, wo wir uns mit eingehender mit Ereignissen beschäftigen.

Übrigens: Intern speichert Delphi die grafische Oberfläche (dort auch „Formular“ genannt) als Codedatei ab. Das sehen Sie in □ Abb. 12.17. Sie könnten also die Oberfläche auch bearbeiten, indem Sie in dieser Textdatei Elemente hinzufügen, löschen oder die Ausprägungen ihrer Eigenschaften verändern. Bequemer ist natürlich die Bearbeitung per Drag & Drop im *WYSIWYG*-Modus (*what you see is what you get*), wie in □ Abb. 12.16 gesehen.

Manche Programmiersprachen sind darauf ausgelegt, dass grafische Oberflächen direkt im Programmcode definiert werden, ähnlich wie es Delphi automatisch im Hintergrund tut. Die einzelnen Elemente der Oberfläche sind dann meist Objekte im Sinne der objektorientierten Programmierung. Sie werden durch Programmanweisungen erzeugt, platziert, bzgl. ihrer Eigenschaften angepasst und mit dem restlichen Programmcode „verdrahtet“. Dieses Vorgehen werden Sie am Beispiel von Python in ► Abschn. 22.2 genauer kennenlernen. Im ► Abschn. 32.4 werden wir sehen, wie im Fall von JavaScript die Oberfläche zwar mit Programmcode gestaltet wird, aber in einer anderen Sprache (nämlich HTML) als der, in der das eigentliche Programm geschrieben wird.

### 12.2.2 Konsolenanwendungen

#### ■ Funktionsweise von Konsolenanwendungen

Konsolenwendungen haben keine grafische Benutzeroberfläche, sondern lediglich eine Textoberfläche.

Sie laufen entweder in der Konsole oder Terminal des Betriebssystems (zum Beispiel Linux Bash, Mac Terminal oder MS-DOS-Eingabeaufforderung) oder in einer integrierten Entwicklungsumgebung. Im ersten Fall handelt es sich um Pro-

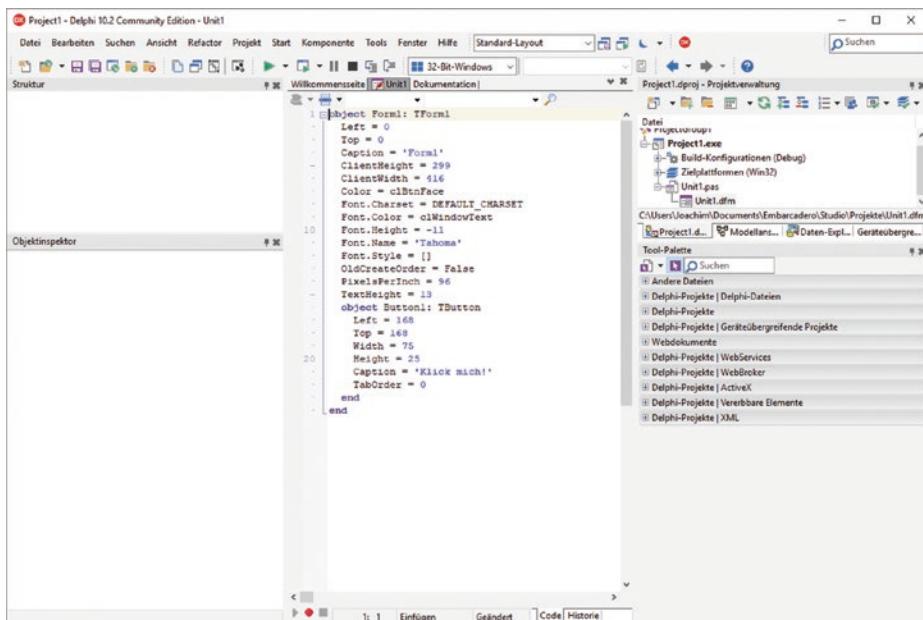


Abb. 12.17 Grafische Benutzeroberfläche, wie Delphi sie intern speichert

12

gramme, die das Betriebssystem direkt ausführen kann (also bereits in Maschinencode vorliegende Programme) oder um Programme in einer interpretierten Sprache, die dadurch ausgeführt werden, dass man in der Konsole des Betriebssystems den Interpreter aufruft und ihn den Programmcode ausführen lässt.

Beim Ausführen in einer Integrierten Entwicklungsumgebung, wird das Programm aus der Entwicklungsumgebung, also einer grafischen Oberfläche heraus aufgerufen, es hat aber selbst nur eine Textoberfläche, läuft also praktisch wie eine Konsolenanwendung. Auch hier wird der Interpreter der Programmiersprache aufgerufen und führt das Programm aus, nur ist die Konsole, in der es läuft, fest in die IDE integriert. In Teil 3 des Buches werden wir Konsolenanwendungen mit Python entwickeln und aus der grafischen IDE heraus aufrufen.

Das Besondere an Konsolenanwendungen ist, dass sie in der Regel strikt linear ablaufen. Wo der Benutzer über die ereignisgesteuerte grafische Benutzeroberfläche selbst entscheiden kann, welche Funktionen er in welcher Reihenfolge aufruft, folgt er in der Konsolenanwendung dem im Programm fest eingebauten Ablauf.

## 12.2 · Grafisch oder nicht grafisch – das ist hier die Frage

Hier ein einfaches Beispiel (dabei signalisiert **>**, dass hier etwas durch den Benutzer eingegeben wird):

```
Bitte geben Sie die Temperatur in Grad Celsius ein:  
> 23  
Umrechnung in Kelvin oder Grad Fahrenheit (K/F)?  
> K  
23 Grad Celsius in Grad Fahrenheit sind: 296.15.
```

In einer grafischen Benutzeroberfläche hätten Sie vielleicht mit Hilfe eines Radio-buttons auswählen können, in welches Temperatursystem Sie Ihre Eingabe umrechnen wollen. Und vor allem: Sie hätten diese Auswahl wahrscheinlich auch treffen können, *bevor* Sie die umzurechnende Temperatur in Grad Celsius eingegeben haben. Nicht so bei der Konsolenanwendung mit ihrer Textoberfläche: Sie gibt vor, was Sie wann eingeben müssen. Das Programm läuft in diesem Fall nicht ereignisgesteuert, sondern linear ab.

Hier zwei Beispiele, wie der erste Teil des obigen Programms in zwei Programmiersprachen, Pascal und Python aussehen könnten; zunächst in Python:

```
temp_celsius = input('Bitte geben Sie die Temperatur in Grad Celsius ein:')  
ziel_skala = input('Umrechnung in Kelvin oder Grad Fahrenheit (K/F)?')
```

Wie Sie sehen, ist es ganz einfach, eine Eingabe von Benutzer abzufragen. Da Sie Variablen in Python nicht deklarieren müssen, ist jede Eingabe letztlich nur eine einzige Code-Zeile.

Jetzt das Ganze in Pascal:

```
program temp  
var  
    temp_celsius : real;  
    ziel_skala : char;  
begin  
    write("Bitte geben Sie die Temperatur in Grad Celsius  
ein:");  
    readln(temp_skala);  
    write("Umrechnung in Kelvin oder Grad Fahrenheit (K/F)?");  
    readln(ziel_skala);  
end.
```

In Pascal müssen die Variablen, die wir verwenden, deklariert werden, eine Fließkommazahlvariable (**real**) und ein Variable, die nur ein einziges Zeichen aufnimmt (**char**). Die Eingabe als solche erfolgt hier mit Hilfe der Funktion **readln()**, was für *read line* steht, das heißt, nach der Eingabe erfolgt automatisch ein Zeilenumbruch, damit die nächste Ausgabe auf einer neuen Zeile beginnt. Ohne das **In** würde der Cursor einfach hinter der Eingabe stehen bleiben. Die nächste Ausgabe würde also genau an dieser Stelle starten. Eine analoge Unterscheidung gibt es bei der Ausgabe, wo zwischen **write()** und **writeln()** unterschieden wird.

### ■ Warum überhaupt Konsolenanwendungen?

Eine wichtige Frage haben wir bislang noch nicht beantwortet: Warum sollte jemand überhaupt eine Konsolenanwendung entwickeln, heute, da doch so viele faszinierende Möglichkeiten für grafische Oberflächen zur Verfügung stehen? Der naheliegendste Grund ist, dass es einfacher schneller geht. Wenn Sie zum Beispiel etwas ausprobieren wollen, eine neue Funktionsbibliothek oder einen bestimmten Algorithmus etwa, dann wäre es reine Zeitverschwendug, viel Arbeit in die Ausgestaltung einer perfekten grafischen Oberfläche zu stecken. Meistens genügt eine ganz simple Konsolenanwendung, die nur eine rudimentäre Text-Interaktion mit dem Benutzer erlaubt. Konzentrieren Sie Ihre Arbeit doch auf das Wesentliche und nicht auf optische Attraktivität und gute Bedienbarkeit, die für Ihren eigentlichen Zweck gar nicht erforderlich ist!

Um noch besser zu verstehen, warum Konsolenanwendungen noch nicht aus der Mode gekommen sind lohnt sich ein Blick auf professionelle oder zumindest ambitionierte Software-Entwickler. Diese schätzen Kommandozeilenwerkzeuge, die in der Konsole laufen, außerordentlich. Und das aus mehreren Gründen: Zum einen ist die Arbeit mit Kommandozeilenprogrammen ergonomischer, denn man kann alle Aufgaben mit nur einem Eingabeinstrument, der Tastatur, erledigen und muss nicht mühsam mit der Maus die entsprechenden Schaltflächen oder Menüeinträge in einer grafischen Entwicklungsumgebung aufrufen. Außerdem lässt sich durch Kommandozeilenparameter, spezielle Steuerungsoptionen, die dem Konsolenprogramm direkt beim Aufruf mit übergeben werden, das Programmverhalten mühelos sehr präzise steuern, was auf einer grafischen Oberfläche unter Umständen mit einer ganzen Reihe von Klicks ungleich mühsamer wäre. Auch sind Konsolenanwendungen regelmäßig weniger anspruchsvoll, was den System-Ressourcenverbrauch (vor allem Speicher und Rechenleistung) betrifft, was bedeutet, dass sie schneller laufen. Nicht verwunderlich ist es daher, dass viele Entwickler-Werkzeuge als Konsolenanwendungen daherkommen, zum Beispiel das bekannte Versionierungstool *git* oder der Text-Editor *vi*. In der Regel haben diese Werkzeuge auf allen Betriebssystemplattformen dieselben Parameter und Steuerungsoptionen, sodass es für den Entwickler einfach ist, zwischen unterschiedlichen Plattformen hin- und herzuwechseln. Und schließlich, auch wenn das natürlich kein Entwickler je offen zugeben würde, ist es auch irgendwie cooler, nerdiger, auf der Kommandozeile der Konsole zu arbeiten, als sich seine Befehle auf einer grafischen Benutzeroberfläche zusammenzuklicken, wie jeder „normalsterbliche“ Benutzer auch.

Viele professionelle Entwickler nutzen einen Computer mit *MacOS* oder *Linux* als Betriebssystem, anstelle eines Windows-basierten Systems. Nach den Gründen

### 12.3 · Arbeiten mit Dateien

gefragt, äußern viele, dass diese beiden Betriebssysteme die bessere Konsole bieten, sodass sie auf diesen Systemen bequemer arbeiten können. Kein Wunder also, dass Microsoft, dessen strategischen Stoßrichtung es explizit ist, sich besonders um die Bedürfnisse von Entwicklern zu kümmern, im Sommer 2019 eine moderne Konsolen-App herausgebracht hat, die die Windows-Betriebssysteme im Wettbewerb um die Gunst der Entwickler wieder nach vorne bringen soll.

#### 12.1

Erläutern Sie zwei Arten, wie grafische Benutzeroberflächen entwickelt werden können.

#### 12.2

Erklären Sie den grundsätzlichen Unterschied im Programmablauf zwischen Konsolenanwendungen und solchen mit grafischer Benutzeroberfläche.

#### 12.3

Nennen Sie zwei Vorteile von Konsolenanwendungen gegenüber Anwendungen mit grafischer Benutzeroberfläche.

## 12.3 Arbeiten mit Dateien

### ■ Der schnellste Weg: Unmittelbares Lesen und Schreiben von Dateien

Daten können natürlich nicht nur vom Benutzer eingegeben werden, sondern auch aus Dateien kommen. Die Arbeit mit Dateien ist grundsätzlich immer gleich und sehr einfach strukturiert:

- Die betreffende Datei wird geöffnet
- Ihr Inhalt wird ausgelesen (oder es wird Inhalt in sie hineingeschrieben)
- Die Datei wird geschlossen.

Manche Programmiersprachen besitzen Funktionen, mit denen man Dateien direkt bearbeiten kann, ohne sich ausdrücklich um das Öffnen und Schließen kümmern zu müssen.

Hier drei Beispiele. Zunächst ein Beispiel aus F# (gelesen „F Sharp“):

```
File.WriteAllText("test.txt", "Ein Beispieltext, direkt in die  
Datei geschrieben")
```

Dasselbe in PHP:

```
file_put_contents("test.txt", "Ein Beispieltext, direkt in die  
Datei geschrieben")
```

Und schließlich noch in R:

```
cat("Ein Beispieltext, direkt in die Datei geschrieben", file =
"test.txt", append = TRUE)
```

In allen drei Fällen wird der Text „Ein Beispieltext, direkt in die Datei geschrieben“ in eine Datei namens **test.txt** geschrieben. Die Datei muss dazu weder ausdrücklich geöffnet noch geschlossen werden, das besorgen die verwendeten Funktionen ohne unser Zutun.

### ■ Modi der Dateiverarbeitung

Alle diese Sprachen verfügen aber zusätzlich über Funktionen, um die oben genannten drei Schritte in der Arbeit mit Dateien auch unabhängig voneinander auszuführen. Will man beispielsweise mehrfach hintereinander Daten in ein- und dieselbe Datei schreiben oder aus ihr lesen, wäre es ja völlig ineffizient, die Datei jedes Mal vorher zu öffnen und wieder zu schließen. In diesem Fall würde man sicher nicht mit den eben betrachteten Funktionen arbeiten, sondern die Datei zunächst öffnen, dann eben mehrmals hineinschreiben oder aus ihr auslesen, und die Datei erst nach Abschluss aller Arbeiten wieder schließen.

Die Schreiboperationen, die wir in den obigen Beispielen in F#, PHP, R gesehen haben, erzeugen eine neue Datei, wenn die Datei mit dem Namen **test.txt** nicht existiert, und schreiben dann unseren Beispieltext in diese neue Datei. Existiert die Datei aber bereits, so wird ihr Inhalt einfach überschrieben, ohne Rückfrage und Bestätigung. Nur im dritten Beispiel, dem Beispiel in der Statistiksprache R, wird über die Option **append=TRUE** gesteuert, dass ein eventuell vorhandener Dateiinhalt nicht überschrieben, sondern unser Text diesem bestehenden Dateiinhalt angehängt werden soll.

Auch Schreiben ist also nicht gleich Schreiben. Tatsächlich gibt es grundsätzlich drei Modi, in denen Dateien geöffnet werden können:

- Lesen (meist **read** oder **r**)
- Schreiben (meist **write** oder **w**)
- Anhängen (meist **append** oder **a**)

Die Modi „Schreiben“ und „Anhängen“ erzeugen regelmäßig eine neue Datei, wenn noch keine Datei mit dem gewünschten Namen existiert. Sie unterscheiden sich aber eben in Hinblick darauf, wie sie mit einem bereits vorhandenen Dateiinhalt umgehen, im Modus „Schreiben“ wird dieser nämlich normalerweise einfach vollständig ersetzt.

In vielen Programmiersprachen sind auch gemischte Modi möglich, etwa **ra** (Lesen = **r** und Anfügen = **a**). Aus historischen Gründen und der Art und Weise, wie Dateisysteme seit jeher arbeiten, gibt es übrigens keinen „Einfügen“-Modus. Sie können also normalerweise nicht einfach eine Datei öffnen, an eine bestimmte Stelle innerhalb der Datei gehen und dort dann einfach zusätzlichen Inhalt einfügen. Stattdessen müssen Sie – auch wenn es umständlich ist – den Inhalt der Datei in einer Variablen Ihres Programms neu aufbauen, also den bestehenden

ersten Teil der Datei in eine Variable (etwa eine Zeichenketten-Variable) auslesen, dann den einzufügenden Inhalt an diese Variable anhängen, und schließlich den hinteren Teil der Datei auslesen und diesen ebenfalls der neuen Inhalt-Variable anhängen. Im Anschluss können Sie die Datei dann im **Write**-Modus öffnen und den Inhalt der Variable in die Datei schreiben. Auf diese Weise haben Sie zwar den alten Dateiinhalt komplett überschrieben, tatsächlich aber lediglich etwas eingefügt.

Anders sieht es beim Lesen aus: Hier können Sie regelmäßig den „Dateizeiger“, der angeibt, an welcher Stelle die nächste Operation in der Datei stattfinden soll, an eine beliebige Stelle in der Datei rücken. Beim Öffnen einer Datei im **Write**-Modus wird der Dateizeiger dagegen automatisch an den Anfang der Datei gesetzt (und kann von dort auch nicht fortbewegt werden, zumindest nicht, ohne dass tatsächlich etwas geschrieben wird), beim Öffnen im **Append**-Modus ans Ende der Datei.

Neben der Art von Bearbeitungsoperation, die auf der Datei ausgeführt werden soll, unterscheiden sich die Modi zum Öffnen von Dateien auch noch in Hinblick auf ein anderes Merkmal, nämlich, ob die bearbeiteten Dateien *Text-* oder *Binärdateien* sind. Der Unterschied zwischen beidem wird am deutlichsten bei Zahlen. Die Zahl 32.000 ist binär ausgedrückt 0111110100000000, also eine 16 Bit lange Folge von Nullen und Einsen. Zwei Byte (à 8 Bit) genügen, um diesen Wert zu speichern. In einer Textdatei jedoch würde die Zahl 32.000 als Text betrachtet werden. Wird sie gespeichert, werden also die einzelnen Zeichen, „3“, „2“, „0“, „0“ und nochmal „0“ gespeichert. Die resultierende Datei wäre dann 5 Byte groß. Unterschiede zwischen Text- und Binärdateien bestehen auch in der Codierung von Zeilenumbrüchen und der Signalisierung des Dateiendes, für das Textdateien ein spezielles Zeichen, das sogenannten EOF-Zeichen (engl. *end of file*: Dateiende) besitzen. Textdateien sind normalerweise so beschaffen, dass man beim Öffnen Buchstaben und Zahlen sieht. Programmcode, den Sie schreiben, beispielsweise wird in einer Textdatei gespeichert. Öffnen Sie dagegen beispielsweise eine PDF-Datei oder eine ausführbare Programmdatei mit einem Texteditor, werden Sie lediglich ein scheinbar zufälliges Muster merkwürdiger Sonderzeichen sehen; es handelt sich um Binärdateien, die Ihr Texteditor als Text darzustellen versucht. Um mit den beiden unterschiedlichen Grundtypen von Dateien umzugehen, besitzen viele Programmiersprachen getrennte Modi für das Schreiben, Anfügen und Lesen von Text- und von Binärdateien.

Schauen wir uns nun das Öffnen, Bearbeiten und Schließen von Dateien einmal etwas genauer an. Ganz allgemein lassen sich diese Operationen in unserem Pseudo-Code so darstellen:

```
meine_datei = oeffnen("test.txt", "w")
schreiben(meine_datei, "Erster Beispieltext, der in die Datei
geschrieben wird.")
schreiben(meine_datei, "Ein weiterer Beispieltext.")
schliessen(meine_datei)
```

In den meisten Sprachen wird beim Öffnen einer Datei eine Variable von einem speziellen Typ zurückgegeben. In unserem Pseudo-Code erhalten wir von der Funktion **oeffnen()**, die wir verwenden, um die Datei **text.txt** im Schreibmodus (**w**) zu öffnen, eine Variable **meine\_datei** zurück. Fortan arbeiten wir mit dieser Variable, damit die Funktionen, mit deren Hilfe wir die Datei jetzt bearbeiten, wissen, auf welche Datei sich unsere Anweisungen beziehen; immerhin könnten wir ja eine ganze Reihe unterschiedlicher Dateien parallel geöffnet haben.

In objektorientierten Programmiersprachen wird die Dateivariable regelmäßig ein Objekt sein, das dann spezielle Methoden besitzt, mit denen man die Datei bearbeiten kann (blättern Sie nochmal einige Seiten in das letzte Kapitel zurück, wenn Ihnen das Thema Objekte und Methoden nicht mehr vertraut vor kommt). Dann könnte das Schreiben und Schließen der Datei ungefähr so aussehen:

```
meine_datei = oeffnen("test.txt", "w")
meine_datei.schreiben("Ein weiterer Beispieltext.")
meine_datei.schliessen()
```

## ■ Beispiele in unterschiedlichen Programmiersprachen

Schauen wir uns das einmal in einigen tatsächlich existierenden Programmiersprachen an.

Hier ein Beispiel in C:

```
#include <stdio.h>

int main() {
    FILE *meine_datei;

    meine_datei = fopen("test.txt", "w");
    fprintf(meine_datei, "Erster Beispieltext, der in die Datei
    geschrieben wird.");
    fprintf(meine_datei, "Ein weiterer Beispieltext.");
    fclose(meine_datei);
}
```

12

Der Code sieht erheblich komplizierter aus, als er es tatsächlich ist. Damit er funktioniert, muss mit der ersten Anweisung eine spezielle Standard-Programmbibliothek namens **stdio.h** eingebunden werden, die die Funktionen für Ein- und Ausgabe bereitstellt. Das Hauptprogramm in C ist selbst eine Funktion, **main()**, die automatisch aufgerufen wird, wenn das Programm aus-

geführt wird. Was dann passieren soll, steht im Inneren der Funktion und ist das, was uns hier eigentlich interessiert. Zunächst wird, wie in C üblich, eine Variable deklariert (in C müssen Variablen vor erstmaliger Verwendung anmeldet werden), und zwar vom Typ **FILE** (genauer gesagt, wird mit dem Sternchen ein *Zeiger* auf eine solche Variable erzeugt, aber der Unterschied soll uns an dieser Stelle nicht weiter beschäftigen). Die soeben angelegte Variable nimmt dann den Rückgabewert der Funktion **fopen()** (für *file open*) auf, mit der wir eine Datei **test.txt** im Schreibmodus (**w**) öffnen. Existiert diese Datei noch nicht, wird sie angelegt.

Danach schreiben wir mit der Funktion **fprintf()** (für *file print formatted*) in die soeben geöffnete Datei, auf die wir mit Hilfe unserer Variable **meine\_datei** referenzieren. Im Anschluss wird die Datei mit **fclose()** wieder geschlossen.

Derselbe Vorgang sähe in Pascal so aus:

```
program DateiSchreiben;
var
    meine_datei: TextFile;
begin
    AssignFile(meine_datei, 'test.txt');
    rewrite(meine_datei);
    write(meine_datei, 'Erster Beispieltext, der in die Datei
geschrieben wird.');
    write(meine_datei, 'Ein weiterer Beispieltext.');
    CloseFile(meine_datei);
end.
```

Hier wird zunächst eine Variable **meine\_datei** deklariert. Dieser Variablen wird dann mit einer speziellen Funktion namens **AssignFile()** die Referenz zu unserer Datei **test.txt** zugewiesen. Bis zu diesem Punkt ist noch gar nicht klar, in welchem Modus die Datei geöffnet werden soll. Das wird erst durch den Aufruf der Funktion **rewrite()** festgelegt, der die Datei zum Schreiben öffnet. Nach dem Schreiben wird die Datei am Ende des Programmstücks mit **CloseFile()** geschlossen.

Pascal weicht also insofern von C ab, als dass hier beim Öffnen der Datei im Schreibmodus die spezielle Funktion **rewrite()** zum Einsatz kommt, in C dagegen die allgemeine Funktion **fopen()**, bei der durch ein Funktionsargument, also eine Einstellung, die wir der Funktion beim Aufruf übergeben, festgelegt wird, in welchem Modus die Datei geöffnet werden soll.

Analog existiert in Pascal eine spezielle Funktion zum Öffnen von Dateien im Lesemodus: **reset()**. Soll die erste Zeile aus Datei **test.txt** (deren Existenz wird jetzt natürlich voraussetzen müssen) gelesen werden, würde der entsprechende Pascal-Code so aussehen:

```
program DateiSchreiben;
var
    meine_datei: TextFile;
    erste_zeile: string;
begin
    AssignFile(meine_datei, 'test.txt');
    reset(meine_datei);
    readln(meine_datei, erste_zeile);
    CloseFile(meine_datei);
end.
```

Dabei kommt die Funktion **readln()** (*read line*) zum Einsatz, die aus der im Lese- modus geöffneten Datei (erstes Argument der Funktion) eine Zeile einliest und in der Variablen **erste\_zeile** (zweites Argument) ablegt. Danach wird der Dateizeiger auch ohne unser Zutun ganz automatisch auf die nächste Zeile vorgeschoben. Würden wir nun erneut eine Zeile einlesen, wäre das dieses Mal die zweite Zeile.

### ■ Arbeit mit Dateien über das Lesen und Schreiben lokaler Dateien hinaus

Die Dateien, die gelesen werden, müssen übrigens nicht unbedingt auf Ihrem lokalen System liegen. Vorausgesetzt, die Dateien unterliegen keinem entgegenstehenden Zugriffsschutz, kann in den meisten Programmiersprachen regelmäßig auch eine Internetadresse (URL, *Uniform Ressource Locator*) als Dateiname angegeben werden. Dateien, die zum Beispiel auf einem Webserver stehen, und die auch Ihr Browser ausliest und als Webseite darstellt, können Sie in vielen Programmiersprachen genauso über ein selbstgeschriebenes Programm lesen als handele es sich um eine Datei auf Ihrem lokalen Computer.

12

Neben den Funktionen zum Öffnen und Schließen, sowie dem Lesen und Schreiben von Dateien, bieten viele Programmiersprachen zudem weitere Funktionen an, um mit dem Dateisystem zu arbeiten, beispielsweise, um im Dateisystem Verzeichnisse anzulegen oder deren Inhalt auszulesen, oder um Dateien und Verzeichnisse zu kopieren, zu verschieben, zu löschen oder umzubenennen. Auch Funktionen zur Ermittlung der Dateigröße oder zur Prüfung, ob eine Datei an einem bestimmten Pfad tatsächlich existiert, gehören regelmäßig zum Standardumfang von Programmiersprachen. Eine Prüfung, ob eine Datei tatsächlich existiert, ist sinnvoll, um einen Fehler oder gar unkontrollierten Programmabsturz zu vermeiden, wenn Ihr Code auf eine Datei zuzugreifen versucht, die es überhaupt nicht gibt.

Die hier besprochenen Ansätze stellen natürlich nur die Grundfunktionalität dar, über die praktisch alle Programmiersprachen verfügen, wenn sie auch, wie ge- sehen, in der konkreten Ausgestaltung im Einzelfall leicht voneinander abweichen mögen. Daneben bieten viele Programmiersprachen – entweder von Haus aus oder durch Erweiterungsbibliotheken – zahlreiche weitere Funktionen, mit denen Da- teien bearbeitet werden können, etwa, um Dateien in speziellen Dateiformaten zu schreiben bzw. zu lesen (zum Beispiel Bilddateien oder Dateien in den proprietären Formaten bestimmter populärer Softwareanwendungen), oder um Dateien über

entsprechende Netzwerkprotokolle wie FTP (File Transfer Protokoll) mit Servern auszutauschen.

#### 12.4

Erläutern Sie die Unterschiede zwischen den unterschiedlichen Möglichkeiten, die beim Öffnen einer Datei bestehen.

### 12.4 Arbeiten mit Datenbanken

In der Praxis der professionellen Software-Entwicklung spielt die Arbeit mit Datenbanken eine große Rolle. Die meisten Web-Services, wie wir sie heute kennen, sind – zugegebenermaßen stark vereinfacht – nichts weiter als eine Datenbank, über der eine Web-Benutzeroberfläche läuft, die die Interaktion des Benutzers mit den Daten der Datenbank erlaubt. Das gilt für Web-Shops ebenso wie für soziale Netzwerke. Im professionellen Umfeld ist das Lesen aus und Schreiben in Datenbanken daher eher die Regel als die Ausnahme, wenn es um Datenein- und -ausgabe geht.

Im Bereich der nicht-professionellen Software-Entwicklung spielt die Arbeit mit Datenbanken sicherlich keine ganz so große Rolle. Deshalb, und weil die Materie keine ganz einfache ist, werden wir in diesem Abschnitt lediglich einige Grundlagen behandeln, mit denen Sie einen Überblick über die Thematik erhalten, und überlassen die Details fortgeschritteneren Programmierkursen.

Die meisten Datenbanken, die sogenannten *relationalen* Datenbanken, sind im Grunde Sammlungen von Datentabellen, die untereinander in Beziehung stehen können. *Relational* heißen sie, weil die Tabellen letztlich Relationen (Zusammenhänge) zwischen bestimmten Objekten und deren Eigenschaften beschreiben.

Betrachten Sie als stilisiertes Beispiel die □ Tab. 12.1, 12.2, und 12.3. □ Tab. 12.1 enthält die Daten von Kunden. Die einzelnen Kunden in den Zeilen (wir sprechen im Zusammenhang mit Datenbank auch von *Datensätzen* oder *data records*) lassen sich über die Spalte (in Datenbanksprache: das *Feld*) **KUNDENID** eindeutig identifizieren. In □ Tab. 12.2 sehen wir zwei Produkte. Auch diese verfügen über einige beschreibende Felder sowie eine eindeutige ID, nämlich das Feld **PRODUKTID**. Tabelle □ Tab. 12.3 repräsentiert nun die Bestellungen, die die Kunden getätigten haben. Eine Bestellung ist darin gekennzeichnet durch eine eigene ID, den Kunden, die Produkte, die der Kunde bestellt hat, sowie das Bestelldatum. Sie sehen, dass Kunden und Produkte durch die IDs, und damit durch *Verweise* auf die Tabellen **KUNDE** bzw. **PRODUKT** repräsentiert werden. Jede Zeile repräsentiert eine Kombination aus Kunde und bestelltem Produkt. So erkennen Sie zum Beispiel an Bestellung **B0002**, dass Kunde **C00003** (Karl Kramer) *zwei* Produkte bestellt hat (nämlich **P001** – die Gartenschaukel, und **P002** – den Balkontisch). Dementsprechend wird in der Zuordnungstabelle **BESTELLUNG** diese Bestellung durch *zwei* Datensätze beschrieben.

Natürlich hätten wir in der Tabelle **BESTELLUNG** alle Daten des Kunden und der Produkte wiederholen können; das würde aber nicht nur die Tabelle unnötig groß machen, sondern auch die Datenpflege erheblich erschweren, denn Änderun-

■ Tab. 12.1 Beispieldatensatz KUNDE

KUNDENID	NAME	VORNAME	EMAIL	STRASSE	HAUSNR	PLZ	ORT	LAND	LETZTERLOGIN
C00001	Müller	Peter	pmuellter32@my-emails.de	Spatzenweg	5	21129	Finkenwerda	DE	07.01.2019 20:21
C00002	Schulze	Anna	anna_s@schulze.com	Wiener Straße	12A	8036	Graz	AT	14.05.2019 23:55
C00003	Kramer	Karl	karl@kramer-gmbh.de	Carolus-Magnus-Strasse	1	76131	Karlsruhe	DE	22.12.2018 07:53
C00004	Karstens	Markus	markus.karstens@mkarstens.at	Innsbrucker Strasse	5	8036	Graz	AT	13.11.2019 21:08

**Tab. 12.2** Beispieldatensatz PRODUKT

PRODUKTID	BEZEICHNUNG	PREIS
P001	Gartenschaufel, Edelstahl	10.99
P002	Balkontisch, Kunststoff, grün	24.99

**Tab. 12.3** Beispieldatensatz BESTELLUNG

BESTELLUNGID	KUNDENID	PRODUKTID	DATUMZEIT
B0001	C00001	P001	06.01.2019 0:08
B0002	C00003	P001	22.12.2018 7:49
B0002	C00003	P002	22.12.2018 7:49
B0003	C00002	P001	16.05.2019 19:58

gen an den Stammdaten eines Kunden (zum Beispiel seine Adresse) müssten dann an zwei Stellen in der Datenbank, der Tabelle **KUNDE** und der Tabelle **BESTELLUNG**, vorgenommen werden. Die Gefahr inkonsistenter Daten würde steigen. Diese Probleme werden durch die hier verwendete Verweistechnik umgangen.

Das Vorgehen, die wesentlichen Objekte nur einmal abzubilden und Zusammenhänge zwischen Ihnen durch Verweise mit Hilfe ihrer IDs, der sogenannten *Schlüssel*, abzubilden, bezeichnet man auch als *Normalisierung*. Das Datenbanksystem stellt dabei sicher, dass diese Verweise auch immer funktionieren, dass also nicht etwa ein Kunde aus der **KUNDE**-Tabelle gelöscht wird, auf den noch aus der **BESTELLUNG**-Tabelle heraus verwiesen wird.

Übrigens sind unsere Beispiel-Tabellen hier noch nicht perfekt normalisiert: Das Bestell-Datum hängt eigentlich an der Bestellung selbst, nicht an den einzelnen Kombinationen aus Kunde und Produkt, die zu einer Bestellung gehören und in der **BESTELLUNG**-Tabelle gespeichert werden. Vollständige Normalisierung könnte man erreichen, indem man aus der Tabelle **BESTELLUNG** die Produkte herauslöst (es blieben dann also nur die ID der Bestellung, der Kunde und das Datum in **BESTELLUNG** übrig) und die Zuordnung zwischen Bestellung und den bestellten Produkten über deren Schlüssel in einer separaten Tabelle abbildet.

Daten, die auf diese Weise in Datenbanken gespeichert sind, können natürlich abgefragt werden. Dazu wird eine spezielle (Abfrage-)Programmiersprache verwendet: SQL, die *Structured Query Language*. In dieser Sprache können Abfragen formuliert werden, die dann vom Datenbanksystem verarbeitet werden. Die abgefragten Daten erhält man als Ergebnis zurück.

Der wichtigste SQL-Anweisung ist **SELECT**. Die Syntax von **SELECT** ist im Grundsatz ganz einfach (auch wenn sie sich natürlich noch an vielen Stellen erwei-

**Tab. 12.4** Ergebnis unserer SQL-Beispiel-Abfrage mit SELECT

VORNAME	NAME
Anna	Schulze
Markus	Karstens

tern lässt, um auch komplexere Abfragen erzeugen zu können): **SELECT Felder FROM Tabelle WHERE Bedingung.**

Die Anweisung

```
SELECT VORNAME, NAME FROM KUNDE WHERE ORT = 'Graz'
```

fragt also alle Datensätze aus der Tabelle **KUNDE** ab, wo das Feld **ORT** den Wert "**Graz**" hat. Dargestellt werden sollen aber nicht alle Felder, sondern nur die Felder **VORNAME** und **NAME**. Das Ergebnis dieser Anweisung ist also eine Tabelle, die nur diese beiden Felder für die betreffenden Datensätze enthält, im Beispiel also wie in **Tab. 12.4** dargestellt.

Mit SQL können aber nicht nur Daten aus Datenbanken abgefragt, sondern auch in Datenbanken geschrieben werden. Dazu werden vor allem die Anweisungen **INSERT** und **UPDATE** verwendet, die *neue* Datensätze in eine Tabelle einfügen (**INSERT**) oder *bestehende* Datensätze updaten (**UPDATE**), also den Wert eines oder mehrerer Felder eines Datensatzes (oder auch mehrerer Datensätze gleichzeitig) ändern.

Aus den meisten Programmiersprachen heraus lassen sich Datenbanken ansprechen (oft mit Hilfe von Erweiterungsbibliotheken, dazu mehr im folgenden Kapitel), mit SQL-Anweisungen beschicken und deren Ergebnisse entgegennehmen und verarbeiten.

## 12.5 Ihr Fahrplan zum Erlernen einer neuen Programmiersprache

Wenn Sie eine neue Programmiersprache lernen...

- finden Sie heraus,
- mit welchen Arten von Benutzeroberflächen (GUIs, Konsole) Sie ihre Programme ausstatten können,
- wenn Konsolenanwendungen unterstützt werden, welche Funktionen zur Ausgabe von Daten sowie zum Eingeben von Daten (durch den Benutzer) zur Verfügung stehen,
- wenn grafische Oberflächen unterstützt werden, wie diese gestaltet werden (grafische WYSISYG- Entwicklung in IDE, Beschreibung der Oberfläche im Code),

- welche Steuerelemente Ihnen für die Gestaltung Ihrer grafischen Benutzeroberfläche zur Verfügung stehen und welche wesentlichen Eigenschaften/Konfigurationsmöglichkeiten diese bieten,
- wie man Dateien zum Lesen, Schreiben oder Anhängen von Daten öffnet (und auch wieder schließt), insbesondere, welche unterschiedlichen Bearbeitungsmodi beim Öffnen zur Verfügung stehen und wie zwischen dem Öffnen von Text- und Binärdateien unterschieden wird,
- welche Funktionen zur Verfügung stehen, um Daten in die geöffneten Dateien hineinzuschreiben oder aus ihnen auszulesen,
- wie man sich mit Datenbanken verbindet sowie SQL-Anweisungen an die Datenbank schickt und die zurückgelieferten Ergebnisse verarbeitet (nur, wenn Sie tatsächlich vorhaben, mit Datenbanken zu arbeiten).

## 12.6 Lösungen zu den Aufgaben

---

### ■ Aufgabe 12.1

Grafische Benutzeroberflächen können zum einen über eine integrierte Entwicklungsumgebung (IDE) im What-You-See-Is-What-You-Get-Modus (WYSIWYG) designt werden, das heißt, die einzelnen Bedienelemente werden ausgewählt und mit der Maus auf der Programmoberfläche platziert. Typischerweise lassen sich dabei die Eigenschaften der Steuerelemente bequem über Dialogfenster oder Toolboxen einstellen. Die Oberfläche entsteht also, ohne dass man sie „programmieren“ müsste. Aber auch das ist möglich. Viele Programmiersprachen erlauben es, die Oberfläche als Teil des Programmcodes zu erstellen. Die einzelnen Bedienelemente sind dann zumeist Objekte, die sich – wie andere Objekte auch – aus dem Programm heraus erzeugen und auf der Oberfläche platzieren lassen. Auf die Eigenschaften der Steuerelemente kann regelmäßig über die Attribute der sie repräsentierenden Objekte zugegriffen werden.

### ■ Aufgabe 12.2

Konsolenanwendungen laufen normalerweise vollkommen linear ab, der Benutzer wird gewissermaßen durch das Programm geführt, das Programm bestimmt den Ablauf der Arbeit des Benutzers. Bei Anwendungen mit grafischen Oberflächen hat der Benutzer normalerweise mehr Kontrolle darüber, in welcher Reihenfolge er welche Arbeitsschritte ausführt. Er kann beispielsweise in beliebiger Reihenfolge auf Buttons und Schaltflächen klicken. In diesem Sinne bestimmt also nicht das Programm das Benutzerverhalten; es reagiert stattdessen auf *Ereignisse*, die der Benutzer auslöst: Klickt dieser zum Beispiel auf einen Button, wird das damit verbundene Ereignis ausgelöst, und der Programmcode, der für den Eintritt dieses Ereignisses vorgesehen ist, wird angesprungen und ausgeführt. Diese Art der ereignisorientierten Programmsteuerung ist lässt sich zwar grundsätzlich auch in Konsolenanwendungen nachbilden, ist dort aber ungleich seltener anzutreffen als in Anwendungen mit grafischen Benutzeroberflächen.

### ■ Aufgabe 12.3

Eine Konsolenanwendung ist, wenn man im Umgang mit ihr geübt ist, meist sehr schnell und einfach zu bedienen. Das Verhalten des Programms lässt sich dabei häufig beim Aufruf aus der Konsole durch Kommandozeilenparameter steuern. Die Bedienbarkeit wird zudem dadurch erleichtert, dass sich alle Steuerungsoperationen mit der Tastatur vornehmen lassen, wohingegen bei grafischen Benutzeroberflächen oft ein Zeigegerät (meist die Maus) und damit der Wechsel der Hand oder der Hände zwischen den Eingabegeräten notwendig ist. Auch in der Entwicklung haben Konsolenanwendungen Vorteile: Sie lassen sich in aller Regel mit geringerem Aufwand programmieren als Anwendungen mit grafische Benutzeroberfläche. Außerdem genügt für die Entwicklung einer Konsolenanwendung in aller Regel bereits ein geringerer Wissensstand über die verwendete Programmiersprache, als wenn eine grafische Benutzeroberfläche entwickelt werden soll.

### ■ Aufgabe 12.4

In den meisten Programmiersprachen unterscheiden sich die Modi zum Öffnen einer Datei zunächst durch den Bearbeitungsmodus, der jeweils erlaubt wird. Die Datei kann regelmäßig zum Lesen (**read/r**), Schreiben (**write/w**) oder Anhängen (**append/a**) geöffnet werden. Beim Öffnen im Schreibmodus wird eine bereits vorhandene Datei komplett neu überschrieben. Sollen Daten an eine bestehende Datei angehängt werden, muss sie im Anhängen-Modus geöffnet werden. Ein zweiter Unterschied liegt darin, ob die Datei als Textdatei oder als Binärdatei geöffnet wird, was sich vor allem auf die Art auswirkt, wie Daten, die in sie hineingeschrieben werden, codiert werden.



# Wie arbeite ich mit Programm-funktionen, um Daten zu bearbeiten und Aktionen auszulösen?

## Inhaltsverzeichnis

- 13.1   Funktionen – 152**
- 13.2   Bibliotheken – 160**
- 13.3   Frameworks – 163**
- 13.4   Application Programming Interfaces (APIs) – 164**
- 13.5   Ihr Fahrplan zum Erlernen einer neuen Programmiersprache – 166**
- 13.6   Lösungen zu den Aufgaben – 166**

## Übersicht

Funktionen sind das wichtigste Werkzeug, mit dem in den meisten Programmiersprachen die eigentliche Arbeit erledigt wird, sei es das Entgegennehmen von Daten vom Benutzer, die Bearbeitung von Daten, die Darstellung von Informationen, die Steuerung des Verhaltens von Benutzeroberflächen, das Lesen aus bzw. Schreiben in Dateien und Datenbanken, oder vieles andere mehr. Deshalb müssen Sie sich, wenn Sie eine Programmiersprache lernen, damit beschäftigen, wie Sie mit Funktionen arbeiten. Genau darum geht es in diesem Kapitel.

In diesem Kapitel werden Sie folgendes lernen:

- was Funktionen eigentlich genau sind,
- wie Sie Funktionen im Programmcode definieren,
- wie Sie Funktionen mit ihren Argumenten aufrufen, und welche Arten von Argumenten und Argument-Übergaben es gibt,
- welche Rolle der sogenannte Gültigkeitsbereich von Variablen spielt, wenn Sie mit Funktionen arbeiten,
- wie Sie Funktionen in der objektorientierten Programmierung einsetzen und welche Vorteile es hat, dass Funktionen in vielen Sprachen selbst Objekte sind,
- wie Funktionen in Bibliotheken zusammengefasst sind und wie Sie geeignete, frei verfügbare Bibliotheken finden können,
- was Frameworks sind und wie sie sich von Bibliotheken unterscheiden,
- was Application Programming Interfaces (APIs) sind und wie Sie mit ihnen arbeiten können.

### 13.1 Funktionen

#### ■ Was sind Funktionen?

Wie in der Mathematik auch (keine Sorge, wir werden diesen Vergleich nicht überstrapazieren!) sind Funktionen letztlich Zuordnungsvorschriften, die Werte, die sogenannten *Argumente*, auf einen anderen Wert, ihren *Funktions-* oder *Rückgabewert* abbilden. Die Funktion  $f(x) = x^2$  beispielsweise ordnet einem Wert  $x$  dessen Quadrat zu; man übergibt der Funktion also einen Wert  $x$  als Argument und erhält von der Funktion einen verarbeiteten Wert (in diesem Fall das Quadrat des Arguments) als Rückgabewert zurück.

Funktionen in Programmiersprachen funktionieren nach demselben Grundprinzip, mit dem Unterschied jedoch, dass nicht alle Funktionen einen Rückgabewert erzeugen. Wir haben im Pseudo-Code der letzten Kapitel bereits mit einer Funktion **anzeigen()** gearbeitet, die eine Ausgabe auf dem Bildschirm vornimmt. Diese Funktion liefert keinen Rückgabewert. Sie führt lediglich eine bestimmte Aktion aus, nämlich die Darstellung auf dem Bildschirm, verarbeitet die ihr als Argument übergebenen Daten aber sonst nicht weiter. Manche Programmiersprachen unterscheiden strikt zwischen Funktionen, die einen Rückgabewert besitzen und *Prozeduren*, die das nicht tun. Wir werden diese Unterscheidung im weiteren Verlauf aber nicht machen und stattdessen stets von Funktionen sprechen. In man-

chen Programmiersprachen liefern Funktionen immer einen Wert zurück; wenn sie keinen „echten“ Rückgabewert haben, geben Sie einen speziellen Wert zurück, der signalisiert, dass es eben kein „echtes“ Funktionsergebnis gibt (zum Beispiel **undefined** in JavaScript oder **void** in C/C++).

### ■ Funktionen definieren

Funktionen bestehen üblicherweise aus einem *Funktionskopf* und einem *Funktionsrumpf* oder *Funktionskörper*. Im Kopf finden sich regelmäßig der Bezeichner der Funktion und die Liste der Argumente, die die Funktion erwartet. Der Rumpf ist ein Code-Block, der immer dann ausgeführt wird, wenn die Funktion aufgerufen wird. Er enthält das „Fleisch“ der Funktion, der darin enthaltene Quelltext beschreibt, was die Funktion eigentlich tut.

Betrachten Sie das Pseudo-Code-Beispiel einer einfachen Funktion, die zwei Zahlen miteinander multipliziert:

```
Funktion vorstellung(name, alter)
Beginn
    ausgeben("Mein Name ist ", name, " , ich bin ", alter,
              "Jahre alt.")
Ende
```

Damit die Programmiersprache weiß, dass jetzt die Definition einer Funktion beginnt, beginnt diese mit dem Schlüsselwort **Funktion**. Es folgen der Funktionsbezeichner sowie die beiden Argumente der Funktion, **zahl1** und **zahl2**. Der Funktionsrumpf besteht aus zwei Anweisungen, einer, die das eigentliche Ergebnis der Funktion errechnet, und einem Aufruf der Funktion **liefere()**, die das Funktionsergebnis zurück gibt.

Die beiden Anweisungen stehen in einem Code-Block, der mit dem Schlüsselwort **Beginn** eingeleitet und mit dem Schlüsselwort **Ende** abgeschlossen wird. Code-Blöcke gibt es in praktisch allen Programmiersprachen. Sie werden meist wie in unserem Pseudo-Code durch spezielle Schlüsselwörter (insbesondere die englischen Schlüsselwörter **begin** und **end** sind häufig anzutreffen) oder Symbole, zum Beispiel öffnende und schließende geschweifte Klammern (also { und }) abgegrenzt. Einige Sprachen wie Python markieren einen Code-Block ganz ohne besondere Schlüsselwörter oder Symbole, ausschließlich durch gleichmäßiges Einrücken aller Code-Zeilen des Blocks.

Aber zurück zu unserer Funktionsdefinition. Deren Programmcode tut per se noch gar nichts. Die Funktion tritt erst dann in Erscheinung, wenn Sie auch tatsächlich aus dem Programm heraus aufgerufen wird. Das könnte in unserem Beispiel so aussehen:

```
anzeigen(multiplizierte(3, 57.8))
```

Mit diesem Aufruf multiplizieren wir die Zahlen 3 und 57,8 und lassen das Ergebnis sofort ausgeben. Was hier nun passiert, ist, dass die Ausführung des Programms

in die Definition der Funktion `multipliziere` verzweigt. Der im Rumpf der Funktionsdefinition enthaltene Code wird ausgeführt, wobei die Funktionsargumente `zahl1` und `zahl2` eben den Wert der übergebenen Faktoren 3 und 57,8 annehmen. Das Ergebnis der Multiplikation wird mit `liefere()` zurückgegeben, womit die Programmausführung den Funktionsrumpf verlässt und wieder ins Hauptprogramm zurückkehrt. Durch die Ausführung der Funktion tritt nun an die Stelle des Funktionsaufrufs der Rückgabewert der Funktion. Dieser kann nun wieder als Argument einer anderen Funktion, in unserem Beispiel `anzeigen()`, übergeben werden. Nach dem Durchlaufen unserer Funktion `multipliziere()` verkürzt sich der Programmcode also faktisch zu: `anzeigen(173.4)`

Weil der Funktionsaufruf nach seiner Ausführung durch den Rückgabewert ersetzt wird, können Funktionsaufrufe auch Variablen zugewiesen werden:

```
rechenwert = multipliziere(3, 57.8)
```

Viele Programmiersprachen verlangen, dass die Funktion *vor* dem ersten Aufruf definiert worden ist, die Funktionsdefinition also „weiter oben“ im Programmcode stehen muss, als der erste Aufruf.

Übrigens: Funktionen können natürlich auch ohne Argumente definiert werden. Eine Funktion zum Beispiel, die einfach nur die Anzeige auf dem Bildschirm löscht, benötigt keine weiteren Informationen, die ihr übergeben werden müssten. In den meisten Programmiersprachen müssen auch solche Funktionen mit den runden Klammern, in denen normalerweise die Werte der Argumente zu finden sind, aufgerufen werden. Zwar sind die runden Klammern in diesem Fall leer, aber der Interpreter/Compiler der Sprache erkennt dann trotzdem, dass es sich hier um einen Funktionsaufruf handelt und nicht etwa um den Zugriff auf eine gleichnamige Variable.

Der Ablauf eines Funktionsaufrufs aus dem Programm heraus ist in

■ Abb. 13.1 schematisch dargestellt.

## 13

### ■ Optionale Argumente

Manchmal möchte man dem Benutzer die Möglichkeit geben, das Verhalten der Funktion über einen Parameter zu steuern, diesen aber mit einem Standardwert vorzubelegen. Gibt der Aufrüfer der Funktion dann keinen Wert für das betreffende Argument an, wird einfach der Standardwert verwendet.

Angenommen, wir wollten unsere Funktion `multipliziere()` so ausgestalten, dass immer `zahl1` mit der Kreiszahl Pi (3,14159...) multipliziert wird, wenn nicht ausdrücklich ein Wert für das Argument `zahl2` beim Funktionsaufruf übergeben wird. Dann müssten wir diesen Standardwert im Funktionskopf angeben: **Funktion `multipliziere(zahl1, zahl2 = 3.14159)`**. Nun könnte ein Aufruf von `multipliziere` auch so aussehen:

```
zweimal_pi = multipliziere(2)
```

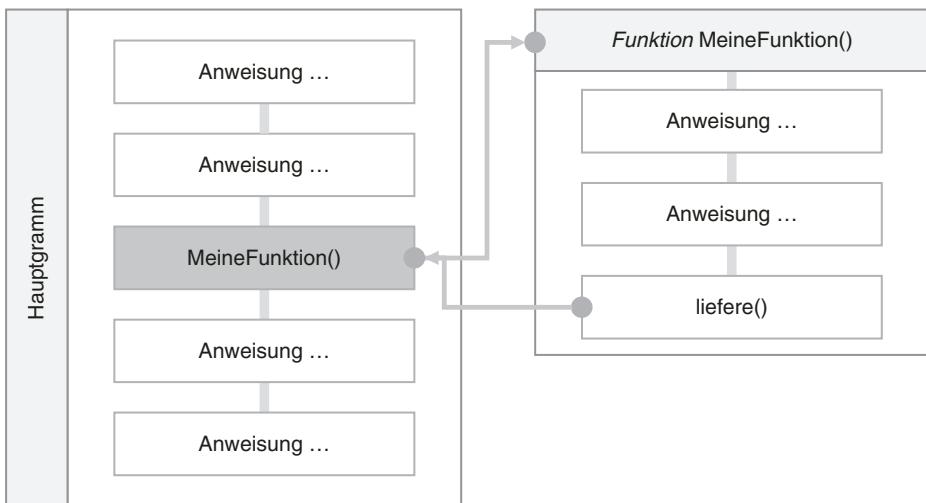


Abb. 13.1 Aufruf einer Funktion

Man spricht in diesem Zusammenhang auch davon, **zahl2** sei ein *optionales Argument*, weil es eben beim Aufruf der Funktion auch weggelassen werden kann.

#### ■ Funktionsargumente mit ihrem Bezeichner übergeben

In vielen Programmiersprachen lassen sich die Argumente beim Aufruf von Funktionen auch über ihre Bezeichner ansprechen. Das könnte dann so aussehen:

```
rechenwert = multiplizierte(zahl1 = 3, zahl2 = 57.8)
```

Der Vorteil dieser sogenannten *Schlüsselwortargumente* besteht darin, dass – gerade beim Aufruf von Funktionen mit vielen Argumenten – klarer wird, welcher Wert zu welchem Argument gehört; man muss also die Reihenfolge der Argumente nicht im Kopf haben. Denn weil nun die Bezeichner der Argumente vorhanden sind, ist der Interpreter/Compiler nicht mehr auf die Reihenfolge der Argumente angewiesen (in diesem Fall spricht man in Abgrenzung von den *Schlüsselwortargumenten* von *Positionsargumenten*), um die übergebenen Werte den einzelnen Argumenten zuzuordnen. Dementsprechend kann man von der eigentlichen Reihenfolge der Argumente auch abweichen; ein gültiger Funktionsaufruf wäre damit auch der folgende:

```
rechenwert = multiplizierte(zahl2 = 57.8, zahl1 = 3)
```

Insbesondere, wenn die Argumente inhaltlich sinnvoll und bestenfalls sogar intuitiv bezeichnet sind, wird der Programmcode nicht nur leichter zu schreiben, sondern vor allem auch leichter zu verstehen.

## ■ Gültigkeitsbereiche von Variablen

Funktionen führen in den meisten Programmiersprachen gewissermaßen ein „Eigenleben“. Sie sind eine abgeschottete Welt ganz für sich allein. Denken Sie nochmal an die Definition unserer Funktion **multiplizierte()** zurück. Innerhalb dieser Funktion wird eine Variable namens **ergebnis** erzeugt, die das berechnete Produkt der beiden Zahl aufnimmt. Diese Variable existiert *nur innerhalb* der Funktion. Mit dem Ende des Funktionsaufrufs, wenn also die Funktion vollständig durchlaufen ist, hört auch die Variable **ergebnis** auf, zu existieren. Auf sie kann aus dem Hauptprogramm heraus nicht zugegriffen werden. Man sagt daher auch, ihr *Gültigkeitsbereich* sei beschränkt auf den Funktionsrumpf.

Angenommen, unser Hauptprogramm sähe folgendermaßen aus:

```
rechenwert = multiplizierte(3, 57.8)
anzeigen(ergebnis)
```

Dann würde die Anweisung `anzeigen(ergebnis)` zu einer Fehlermeldung führen (oder, je nach Sprache, einen Standardwert wie **0** liefern), denn im Gültigkeitsbereich des Hauptprogramms gibt es keine Variable namens **ergebnis**. Diese Variable steckt im Gültigkeitsbereich der Funktion **multiplizierte()**. Weil ihr Gültigkeitsbereich beschränkt ist, spricht man auch davon, **ergebnis** sei eine *lokale Variable*. Die Variable **rechenwert** dagegen wird im Hauptprogramm erzeugt und ist überall gültig. Solche Variablen werden in Abgrenzung zu den lokalen Variablen *globale Variablen* genannt. Auf sie kann von überall her zugegriffen werden, auch aus unserer Funktion **multiplizieren()** heraus. Best Practice ist allerdings, solcherlei Zugriff auf globale Variablen aus einer Funktion heraus (man spricht auch von Nebenwirkungen, engl. *side effects*) zu vermeiden, denn es macht die Funktion anfälliger für Änderungen am Rest des Programms, vor allem gegenüber Änderungen der Bezeichner globaler Variablen.

13

Was aber nun, wenn die Funktion **multiplizieren()** selbst eine Variable namens **rechenwert** verwenden würde, zum Beispiel, wenn die zentrale Anweisung der Funktion **rechenwert = zahl1 \* zahl2** lauten würde? Dann hätten wir ja zwei Variablen namens **rechenwert**, eine lokale (innerhalb der Funktion **multiplizierte()**) und die globale im Hauptprogramm. Auf welche wird nun zugegriffen, wenn irgendwo im Programmcode der Bezeichner **rechnerwert** verwendet wird? In aller Regel suchen Programmiersprachen zuerst in der lokalen Umgebung nach einer Variablen mit diesem Bezeichner und erst dann, wenn sie dort keine finde, in der globalen Umgebung. Würde also die Variable **rechenwert** innerhalb der Funktion **multiplizierte()** verwendet werden, würde die lokale Variable zum Einsatz kommen, jene, die im aktuellen Gültigkeitsbereich definiert worden ist. Ein Zugriff auf **rechenwert** aus dem Hauptprogramm führt hingegen zu einem Zugriff auf die globale Variable.

Funktionsargumente verhalten sich in der Regel wie lokale Variablen, auch auf sie kann nur aus dem Code der Funktion heraus zugegriffen werden.

### ■ Argumentübergaben als Referenz statt als Wert

Betrachten Sie folgende, leicht veränderte Variante unserer Funktion **multipliziere()**:

```
Funktion multipliziere(zahl1, zahl2, AlsReferenz ergebnis)
BeginnFunktion
    ergebnis = zahl1* zahl2
EndeFunktion
```

Dieses Mal hat unsere Funktion keinen Rückgabewert; stattdessen wird das Ergebnis der Berechnung einer Variablen zugewiesen, die ebenfalls als Argument übergeben wurde, allerdings anders als **zahl1** und **zahl2** mit dem Schlüsselwort **AlsReferenz**.

Normalerweise werden die Argumente einer Funktion als Kopien der Originalwerte übergeben. Betrachten Sie das folgende Beispiel eines Aufrufs unserer Funktion:

```
zahl1 = 3
rechenwert = 0
multipliziere(zahl1, 57.8, rechenwert)
anzeigen(rechenwert)
```

Hier ist das erste Argument der Funktion selbst eine Variable. Deren Wert ist als Argument **zahl1** im Inneren der Funktion **multiplizieren()** verfügbar, allerdings nur als Kopie der Originalvariable. Würden wir dort nun den Wert von **zahl1** anpassen, hätte das auf die Variable **zahl1** im Hauptprogramm keinerlei Auswirkungen. Nur die lokale Variable **zahl1** im Funktionsrumpf von **multiplizieren()** würde sich im Wert ändern.

Anders im Fall der Variablen **rechenwert**. Diese wird der Funktion nicht als Wert übergeben, sondern als sogenannte *Referenz*. Das bedeutet, Änderungen, die innerhalb der Funktion an dieser Variablen vorgenommen werden, haben Auswirkungen auf die Originalvariable; im Beispiel verwenden wir diese Variable, um das Ergebnis der Berechnung „zurückzugeben“. Manche Programmiersprachen erlauben es, Variablen *als Wert* (engl. *by value*) oder *als Referenz* (engl. *by reference*) zu übergeben, manche kennen lediglich die Übergabe als Wert.

### ■ Beispiele für die Definition von Funktionen

Bisher haben wir unsere Funktionsdefinition nur in unserem Pseudo-Code geschrieben. Deshalb hier zwei Implementierungen der ursprünglichen Funktion **multipliziere()** in echten Programmiersprachen, nämlich PHP und Pascal.

Zunächst die PHP-Version:

```
function multipliziere($zahl1, $zahl2)
{
    $ergebnis = $zahl1 * $zahl2;
    return $ergebnis;
}
```

Die Rückgabeanweisung **return** ist, wie in manchen Sprachen, keine Funktion, sondern ein Schlüsselwort (daher auch keine runden Klammern um das „Argument“).

Dann dasselbe in Pascal:

```
function multipliziere(zahl1, zahl2: real);
var
    ergebnis: real;
begin
    ergebnis := zahl1 * zahl2;
    multipliziere := ergebnis;
end;
```

Pascal ist eine stark typisierte Programmiersprache. Variablen werden hier stets mit einem festen Typ deklariert. Deshalb haben sowohl die beiden Argumente der Funktion, **zahl1** und **zahl2**, als auch die Funktion selbst (am Ende des Funktionskopfes) jeweils eine Typangabe: In allen Fällen handelt es sich um Fließkomma-zahlen (**real**). Damit ist nicht nur klar, welche Art von Argumenten die Funktion erwartet, sondern auch, von welchem Typ ihr Rückgabewert sein wird.

Die Rückgabeanweisung arbeitet in Pascal nicht, wie in vielen anderen Programmiersprachen mit einem Schlüsselwort **return** oder einer Funktion **return()**, sondern wird dadurch erreicht, dass dem Bezeichner der Funktion der Funktionswert zugewiesen wird (Zuweisungen werden in Pascal mit **:=** als Zuweisungsoperator formuliert).

### ■ Funktionen in der objektorientierten Programmierung

In ► Abschn. 11.7.4 hatten wir gesehen, dass im Rahmen der Objektorientierten Programmierung Funktionen auch Bestandteil von Klassen sein können. Solche Funktionen nennt man, wie Sie sich erinnern werden, auch *Methoden*. Der Aufruf dieser Methoden erfolgt immer im Zusammenhang mit einer konkreten Objektinstanz der Klasse, also einem Objekt, das nach der „Schablone“ der Klasse erzeugt worden ist. Die meisten Programmiersprache benutzen zur Trennung von Objekt und Methode beim Aufruf den Punkt (manche aber auch andere Symbole wie etwa **->**). Der Aufruf einer Methode könnte demnach zum Beispiel so aussehen:

```
objekt.methode(argumente)
```

Je nach Programmiersprache ist die Definition der Methode Bestandteil der Klassendefinition oder erfolgt außerhalb der Klassendefinition. In jedem Fall aber findet sich in der Klassendefinition ein Hinweis auf die Methode (oft der Methodenkopf), wie wir es in ► Abschn. 11.7.4 auch gesehen haben (blättern Sie gegebenenfalls noch mal einige Seiten zurück!).

Außer dass Funktionen als Methoden Bestandteil von Objekten (bzw. deren Klassen) sein können, haben Funktionen und Objekte häufig noch einen anderen Bezug: In vielen Programmiersprachen nämlich *sind* Funktionen Objekte. Dort existiert eine spezielle Klasse (häufig **function**) und alle Funktionen sind Objekte (also Instanzen) dieser Klasse. Als solche können sie u. U. auch Eigenschaften besitzen, wie etwa ihre Argumente oder ihren Funktionsrumpf, also den eigentlichen Code. Wenn Funktionen selbst Objekte sind, hat das einige bemerkenswerte Auswirkungen: So können Funktionen selbst als Argumente anderer Funktionen dienen. Zudem sind Klassendefinitionen logisch sehr stringent: Sie bestehen dann strenggenommen ausschließlich aus Attributen. Nur sind einige Attribute, nämlich die Methoden, dann aufrufbar (engl. *callable*) und andere (die „normalen“ Attribute im Sinne von Werte-Eigenschaften) eben nicht.

### 13.1 [3 min]

Was ist falsch an der folgenden Funktionsdefinitionen:

a. –

```
Funktion exponential(basis)
Beginn
    liefere (basis^exponent)
Ende
```

b. –

```
Funktion exponential(basis, exponent)
Beginn
    ergebnis = basis^exponent
Ende
```

### 13.2 [3 min]

Angenommen, wir hätten eine Funktion, die wie folgt definiert ist:

```
Funktion vorstellung(name, alter)
Beginn
    ausgeben("Mein Name ist ", name, " , ich bin ", alter, "
    Jahre alt.")
Ende
```

Worin unterscheiden sich die folgenden Aufrufe der Funktion, und warum führen der erste und der dritte Aufruf zum gewünschten Verhalten der Funktion, der zweite jedoch nicht?

```
Vorstellung("Ulrike", 25)
vorstellung(25, "Ulrike")
vorstellung(alter = 25, name = "Ulrike")
```

### 13.3 [5 min]

Betrachten Sie den folgenden Programmausschnitt:

```
alter_person = 25

Funktion werdeAelter(alter_person)
Beginn
    alter_person = alter_person + 1
    ausgeben("Neues Alter der Person: ", alter_person)
    liefere(alter_person)
Ende

alter_neu = werdeAelter(alter_person)
ausgeben("Derzeitiges Alter: ", alter_person)
ausgeben("Ergebnis der Funktion werdeAelter(): ", alter_neu)
```

- Welche Ausgaben generiert dieses Programm und warum?
- Welchen beiden Möglichkeiten gibt es, das Programm so zu verändern, damit sich das neue, um ein Jahr höhere Alter auch in der globalen Variablen **alter\_person** widerspiegelt?

## 13

### 13.2 Bibliotheken

#### ■ Bibliotheken als Werkzeugkasten für Programmierer

Als Programmierer können Sie – wie im vorangegangenen Abschnitt gesehen – selbst Funktionen entwickeln. Das macht immer dann Sinn, wenn Sie Programmcode *wiederverwenden* wollen, denn das Angenehme an Funktionen ist ja gerade, dass Sie eine bestimmte Funktionalität aus Ihrem eigentlichen Programm herauslösen und jederzeit von überallher aufrufen können.

Die Programmiersprachen kommen aber von Haus aus natürlich regelmäßig bereits mit einem umfangreichen Satz an Standardfunktionen, mit dem Sie viele häufig vorkommende Aufgaben erledigen können.

Häufig allerdings werden diese Standardfunktionen nicht ausreichen für das, was Sie vorhaben. Funktionen etwa zum Versenden von E-Mails, Durchsuchen von Webseiten (Webscraping) oder dem Trainieren neuronaler Netze sind normalerweise nicht im Standardsprachumfang enthalten. In diesen und zahllosen ande-

ren Anwendungsfällen müssen Sie – zumindest, wenn sie die betreffende Funktionalität nicht selbst entwickeln wollen – den Sprachumfang auf eigene Faust erweitern, indem Sie die benötigten Funktionen von anderswoher dazustellen.

### ■ Geeignete Bibliotheken finden

Einige Programmiersprachen wie Python, R und JavaScript haben ein sehr umfangreiches und lebendiges „Ökosystem“ mit einer Vielzahl von Entwicklern, die die von ihnen entwickelten Funktionalitäten für andere unentgeltlich bereitstellen. Das geschieht regelmäßig in Form inhaltlich zusammenhängender Sammlungen von Funktionalitäten, die je nach Programmiersprache meist als *Bibliothek*, *Modul*, *Package* oder *Distribution* bezeichnet werden. Wir sprechen hier der Einfachheit halber immer von „Bibliotheken“, die anderen Entwicklern (wie Ihnen) zur Benutzung zur Verfügung stehen.

In manchen Fällen sind diese Bibliotheken auf einer Plattform zusammengeführt, die durch die Organisation, die die Entwicklung der Programmiersprache betreut, zentral verwaltet wird. So ist es beispielsweise bei Python mit dem *Python Package Index (PyPI)*, ► <https://pypi.org/>), bei Perl mit dem *Comprehensive Perl Archive Network (CPAN)*, ► <https://www.cpan.org/>), bei PHP mit der *Perl Extension Community Library (PECL)*, ► <https://pecl.php.net/>) oder bei R mit dem *Comprehensive R Archive Network (CRAN)*, ► <https://cran.r-project.org>). Je nach Ausgestaltung geht mit der Aufnahme in einen dieser zentralen Kataloge auch eine (meist weitestgehend automatisierte) Qualitätskontrolle einher, die beispielsweise sicherstellt, dass der Programmcode der Bibliothek lauffähig ist und wenigstens über ein Minimum an Dokumentation verfügt.

Für zahlreiche Programmiersprachen jedoch existiert eine solche zentral organisierte Bibliotheksplattform nicht. Oft legen die Entwickler Ihre Werke dann auf *GitHub* ab. *GitHub* ist eine sprachübergreifende Plattform, die es Entwicklern erlaubt, Quellcode mit anderen zu teilen, indem er in einem sogenannten *Repository* gespeichert wird. Unter *GitHub* liegt das vom Linux-Erfinder Linus Torvalds entwickelte Versionsmanagement-Werkzeug *Git*, mit dem Änderungen am Quelltext auf elegante Art und Weise versioniert werden können. Nötigenfalls kann man dann leicht auf einen alten Stand einer einzelnen Code-Datei oder sogar des ganzen Repositories zurückgehen. Auch können sich Entwickler mit *GitHub* eigene Kopien (engl. *branches*) des Quellcodes ziehen, Features darin weiterentwickeln und diese schließlich wieder mit der Originalversion zusammenbringen (engl. *mergen*), zumindest, wenn der Entwickler der Originalversion dies zulässt. Mit diesen und einer ganzen Reihe weiterer Features erleichtert *GitHub* auf Basis des zugrundeliegenden Versionsmanagement-Tools *Git* die Zusammenarbeit unterschiedlicher Entwickler. *GitHub* ist dabei vollkommen sprachagnostisch, zu den meisten heute verwendeten Programmiersprachen werden sich Repositories mit Code in dieser Sprache finden. Für Entwickler, die ihren Quellcode im Rahmen einer Open-Source-Lizenz anbieten, ist die Benutzung von *GitHub* zum Zeitpunkt, als dieses Buch geschrieben wird, kostenfrei. Unternehmen und Organisationen, die Ihren Code nach außen abschirmen wollen, bezahlen für eine private Umgebung. Neben *GitHub* existiert zwar noch eine Reihe vergleichbarer Plattformen, *GitHub* ist aber die populärste.

Natürlich wird *GitHub* nicht nur von denjenigen Entwicklern verwendet, die in einer Sprache ohne zentrale Bibliotheksplattform arbeiten. Viele Entwickler betreiben *GitHub*-Repositories einfach für ihre normale Arbeit am Programmcode und um sich mit anderen Entwicklern auszutauschen; nur die fertigen Versionen machen sie dann auf den zentralen Bibliotheksplattformen wie *PyPI* oder *CRAN* verfügbar. Manchmal wollen sich Entwickler auch nicht den strengen Regeln und den automatisierten Qualitätsprüfungen unterwerfen und bieten ihre Bibliotheken deshalb ausschließlich auf *GitHub* an.

Es ist also nicht verwunderlich, dass die zentralen Bibliotheksplattformen und *GitHub* beide gute Anlaufstellen sind, wenn Sie eine Bibliothek suchen, die Ihnen hilft, ein bestimmtes Problem zu lösen. Trotzdem ist diese Suche mitunter kein ganz leichtes Unterfangen, und zwar aus mehreren Gründen. Die Suchfunktionalitäten der Plattformen sind durchaus unterschiedlich gut ausgeprägt, ebenso die verfügbaren Informationen über die einzelnen Bibliotheken. Es ist also manchmal gar nicht leicht, eine Bibliothek zu finden, wenn man denn eine gefunden hat, zu beurteilen, ob diese tatsächlich geeignet ist, das eigene Problem zu lösen. Erschwert wird diese Beurteilung zusätzlich dadurch, dass häufig mehr als nur eine Bibliothek in Frage kommen wird. Auch auf den zentral administrierten Bibliotheksplattformen gibt es für viele Aufgaben mehrere, gewissermaßen konkurrierende Bibliotheken. Zweierlei Ansätze sind dann zu empfehlen. Zum einen *Ausprobieren*; laden Sie sich die „Kandidaten“ herunter und arbeiten Sie damit, um festzustellen, welcher der geeignetste für Ihre Fragestellung ist. Zum anderen die *Recherche* im Internet, wo Sie in einschlägigen Foren (etwa dem bereits erwähnten *StackOverflow*) regelmäßig auch Informationen zu vielen Bibliotheken finden; das ist insbesondere dann nützlich, wenn Sie anhand von Beispielen sehen wollen, wie man die Bibliothek überhaupt verwendet. Die Dokumentation der Bibliotheken ist durchaus von unterschiedlichem Umfang und unterschiedlicher Qualität, sodass Foren wie *StackOverflow* auch dabei eine gute Hilfestellung bieten können, die bereits als geeignet befundene Bibliothek wirklich zum Einsatz zu bringen.

## 13

Abgesehen davon sind Foren wie *StackOverflow* natürlich bereits häufig eine gute *erste* Anlaufstation, um – wenn man direkt nicht über eine zentrale Bibliotheksplattform oder auf *GitHub* suchen kann oder will – Bibliotheken-Kandidaten zu ermitteln, die für die Lösung des eigenen Problems in Frage kommen; das liegt daran, dass die Fragen in den Foren ja meist nach dem Muster „Wie löse ich das Problem, dass...“ oder „Wie schaffe ich es, dass...“ aufgebaut sind und sich dann in den Antworten oft Hinweise auf eine oder mehrere Bibliotheken finden, die bei der Lösung des Problems helfen können. Viele Entwickler ziehen diese Art der Foren-Suche dem direkten Suchen auf zentralen Bibliotheksplattformen oder *GitHub* vor, weil die Foren mit den Beispielen und Kommentaren von Nutzern häufig wertvolle Zusatzinformationen bieten.

### ■ Bibliotheken installieren und einbinden

Wenn Sie eine geeignete Bibliothek gefunden haben, müssen Sie sie nur noch installieren (sofern die Programmiersprache das vorsieht bzw. erforderlich macht) und in den Programmcode einbinden.

Das Einbinden erfolgt durch eine Anweisung, die entweder die Bibliothek als ganzes oder einzelne Elemente daraus (einzelne Funktionen oder Klassen) in den Programmcode importiert. Teilweise kann dabei ein Name angeben werden, unter dem das importierte Element angesprochen werden kann, was nützlich ist, um Konflikte mit den Bezeichnern von bereits bestehenden Variablen, Klassen, Objekten oder Funktionen zu vermeiden.

Ein Beispiel aus Python:

```
from bibliothek import klasse as meine_klasse  
import bibliothek
```

Mit der ersten Anweisung wird lediglich die Klasse **klasse** in den Programmcode importiert und diese dann unter dem Bezeichner **meine\_klasse** zugänglich gemacht. Mit der zweiten, alternativen Anweisung wird die gesamte Bibliothek importiert (in diesem Fall ohne ihr einen anderen Bezeichner zuzuweisen).

Einfacher ist es in Pascal

```
uses bibliothek;
```

oder R

```
library(bibliothek)
```

wo jeweils die gesamte Bibliothek eingebunden wird.

### 13.3 Frameworks

Viel ist insbesondere im Umfeld der Web-Entwicklung die Rede von sogenannten *Frameworks*. Die Begriffe Framework und Bibliothek werden dabei manchmal nicht ganz trennscharf verwendet. Auch wenn die Funktionsweise und Benutzung von Frameworks über das hinausgeht, was wir uns in diesem Buch eingehender anschauen werden, so soll doch die Unterscheidung zwischen beiden Konzepten kurz ein wenig genauer beleuchtet werden.

Bibliotheken enthalten Funktionalitäten, auf die der Verwender, also der Programmierer, dann zugreifen kann, wenn er sie benötigt, um eine bestimmte Aufgabe zu erledigen. Es ist also der Programmierer, von dem die Initiative ausgeht.

Anders dagegen bei den Frameworks. Frameworks steuern den Ablauf der gesamten Anwendung und rufen den Code des Programmierers auf, wenn es notwendig ist. Frameworks bilden also, wie der Begriff bereits vermuten lässt, einen Rahmen, den der Programmierer „nur“ noch füllen muss. Das ist sehr praktisch, denn gerade im Bereich der Web-Entwicklung gibt es viele immer wiederkehrende Auf-

gaben, wie etwa die Authentifizierung (zum Beispiel durch Login des Benutzers), die Anbindung von Datenbanken oder die Ausgabe von Daten in Template-artigen Seiten; all<sup>e</sup> diese Aufgaben kann das Framework dem Entwickler abnehmen. Sie sind als Funktionalitäten in den Rahmen, mit dem der Entwickler arbeitet, bereits eingebaut. Der Programmierer selbst muss nur noch das entwickeln, was für seine Anwendung speziell ist. Durch diese Arbeitsteilung zwischen Framework und Entwickler wird die Steuerung der Anwendung praktisch umgedreht (man spricht daher im Zusammenhang mit Frameworks auch vom Konzept der *inversion of control*): Das Framework steuert als Rahmen die Anwendung, der Entwickler liefert den anwendungsspezifischen Programmcode, den das Framework dann an der richtigen Stelle aufruft. Das erlaubt es dem Entwickler, sich auf die wichtigen Themen konzentrieren zu können, und das ganze eher langweilige „Drumherum“ dem Framework zu überlassen.

Bekannte Frameworks sind etwa *AngularJS* und *React* (für JavaScript), *django* (für Python), *CakePHP* und *Zen* (für PHP) sowie *Ruby on Rails* (für Ruby).

Weil der Schwerpunkt dieses Buchs auf dem Erlernen der Grundlagen des Programmierens liegt, werden wir hier nicht mit Frameworks arbeiten. Voraussetzung für die Anwendung von Frameworks ist natürlich aber die Kenntnis der zugrundeliegenden Sprache, und genau damit beschäftigen wir uns hier im Buch.

## 13.4 Application Programming Interfaces (APIs)

---

Ein weiterer Begriff, der in aller Munde ist, ist der des *Application Programming Interface*, kurz API. APIs sind zunächst mal – wie der Name ja bereits sagt – nichts weiter als *Programmierschnittstellen*. In diesem Sinne bietet eine Bibliothek, die eine Reihe von Funktionen zu einem bestimmten Zweck zur Verfügung stellt, auch eine Programmierschnittstelle an; diese besteht aus den Funktionen der Bibliothek, auf die Sie aus Ihrem Programm heraus zugreifen können.

13

Wenn heute von APIs gesprochen wird, sind häufig aber ganz bestimmte Programmierschnittstellen gemeint, und zwar Web-APIs durch die Sie auf Funktionalitäten und Daten von Web-Diensten zugreifen können. Viele Web-Dienste bieten Entwicklern heute Programmierschnittstellen an. So liefert Ihnen eine durch Google bereitgestellte *GoogleMaps*-API die Geokoordinaten einer Stadt zurück, wenn Sie der API den Namen der Stadt übergeben. Hier wird eine API also dazu benutzt, Informationen, die der Web-Dienst zur Verfügung stellt, abzufragen. Genauso können APIs aber auch verwendet werden, um einen Web-Dienst dazu zu bringen, eine bestimmte Aktion auszulösen. Die API von Twitter zum Beispiel erlaubt es, Tweets zu posten. Alles, was Sie dazu benötigen, ist ein Account, der die Benutzung der API zulässt und eine entsprechende Anweisung in Ihrem Programm, die die API anspricht und den Tweet absetzt.

APIs können also dazu verwendet werden, Daten abzufragen und Funktionen auszulösen. Es gibt buchstäblich tausende Web-Dienste, die über das Inter-

net ansprechbare APIs anbieten. Die Website *ProgrammableWeb* (► <https://www.programmableweb.com/apis/directory>) liefert einen sicher nicht vollständigen, aber doch zumindest sehr umfangreichen Überblick über Anbieter von APIs. Ob Sie nun aus Ihrem Programm heraus Zahlungen mit PayPal abwickeln oder aber die letzten Bundesliga-Ergebnisse abfragen wollen, mit den heute verfügbaren Web-APIs bleibt kaum ein Wunsch unerfüllt.

Technisch gesehen funktionieren die Web-APIs meist mit dem *Hypertext Transfer Protocol (HTTP)* und können wie Webseiten aufgerufen werden. Um beispielsweise die *GoogleMaps*-API anzusprechen, um die Geoposition von München zu ermitteln, würde ein Programm lediglich eine HTTP-Anfrage der Form ► <https://maps.googleapis.com/maps/api/geocode/json?address=Munich&key=XX> stellen müssen, wobei **key** ein Parameter ist, der den Aufrüfer der API als rechtmäßigen Nutzer authentifiziert. Zurückgeben würde Google dann die Koordinaten, und zwar im JSON-Format, einem bei Web-APIs überaus populären, weil leicht zu erzeugenden und gut lesbaren, Datenaustauschformat, das wir in ► Abschn. 31.5.6 genauer kennenlernen werden. Würden Sie die Anfrage, so wie sie ist, in Ihren Web-Browser eingeben, würden Sie ebenfalls ein Ergebnis im JSON-Format zurück erhalten, allerdings eine Fehlermeldung, weil **XX** natürlich kein Schlüssel ist, der Sie als rechtmäßigen API-Verwender ausweist:

```
{  
    "error_message" : "The provided API key is invalid.",  
    "results" : [],  
    "status" : "REQUEST_DENIED"  
}
```

Ein Programm benötigt allerdings keinen Web-Browser, um solche Anfragen an einen Web-Dienst via API zu stellen. Die entsprechenden HTTP-Anfragen lassen sich aus den meisten Programmiersprachen heraus (häufig unter Verwendung einer Bibliothek, die die entsprechenden Funktionen zur Verfügung stellt), auslösen und ihre Rückgaben verarbeiten, ohne, dass der Anwender des Programms davon etwas mitbekommen muss.

Auch mit Web-APIs werden wir im weiteren Verlauf des Buches nicht weiterarbeiten, weil wir uns in diesem Buch auf die Grundlagen des Programmierens konzentrieren. Diese Grundlagen vorausgesetzt, ist es aber ein leichtes, den nächsten Schritt zu gehen und zu lernen, wie genau man Web-APIs in seine Programme einbinden kann. Wichtig an dieser Stelle zu verstehen ist lediglich, dass Web-APIs ein vielfältig einsetzbares Werkzeug sind, um die Funktionalität Ihrer Programme zu erweitern. Letztlich arbeiten sie wie die Funktionen einer Bibliothek auch: Sie werden aufgerufen (wenn auch etwas anders), lösen eine Aktion aus und geben einen Wert zurück (häufig ein JSON-Objekt). Das Grundkonzept, nämlich das einer Programmierschnittstelle, eben einer API, ist in beiden Fällen dasselbe.

## 13.5 Ihr Fahrplan zum Erlernen einer neuen Programmiersprache

---

### ■ Wenn Sie eine neue Programmiersprache lernen...

finden Sie heraus,

- wie Funktionen definiert werden; dabei insbesondere auch, wie optionale Argumente verwendet werden, wie (zunächst einmal: ob) Argumente als Referenz statt als Wert entgegengenommen werden können, und wie Funktionen Werte zurückliefern können,
- wie Funktionen aufgerufen werden; dabei insbesondere auch, wie (zunächst einmal: ob) Argumente beim Funktionsaufruf mit ihrem Namen übergeben werden können,
- wie die Gültigkeitsbereiche von Variablen gestaltet sind und ob es möglich ist, aus einer Funktion heraus auf eine globale Variable zuzugreifen (insbesondere dann, wenn sie denselben Namen wie eine lokale Variable besitzt)
- welche guten Bezugsquellen für Bibliotheken es gibt und wie Sie nach geeigneten Bibliotheken suchen können; insbesondere, ob eine zentral administrierte Plattform existiert, die de facto der wichtigste Anlaufpunkt bei der Suche nach geeigneten Bibliotheken ist,
- wie Sie Bibliotheken in Ihr Programm einbinden und auf deren Funktionen zugreifen.

## 13.6 Lösungen zu den Aufgaben

---

### ■ Aufgabe 13.1

- a. In der Rückgabe-Anweisung **`liefere()`** wird zwar ein Argument **`exponent`** verwendet, dieses Argument kommt aber nicht in der Argumente-Liste im Kopf der Funktion vor. Der Kopf müsste richtig also **Funktion exponential(basis, exponent)** lauten.
- b. Dieses Mal ist zwar das Argument **`exponent`** in der Argumente-Liste im Funktionskopf mit aufgeführt. Allerdings gibt die Funktion keinen Wert zurück. Es wird zwar eine Variable **`ergebnis`** berechnet, diese dann aber nicht mit Hilfe von **`liefere()`** zurückgegeben. Die Anweisung **`liefere(ergebnis)`** müsste also als letzte Anweisung der Funktion noch ergänzt werden.

### ■ Aufgabe 13.2

**Vorstellung("Ulrike", 25)** ruft die Funktion so auf, wie sie gedacht ist. Das erste Argument in der Argumente-Liste im Kopf der Funktion ist der Name, das zweite das Alter. **`vorstellung(25, "Ulrike")`** dreht die Argumentenwerte beim Funktionsaufruf herum. Jetzt wird das erste Argument (**`name`**) mit dem Wert **25**, das zweite Argument (**`alter`**) mit dem Wert "**Ulrike**" belegt. Im besten Fall gibt die Funktion dann einen etwas merkwürdig anmutenden Text aus, schlimmstenfalls aber bricht sie mit einem Fehler ab, weil die erwarteten Typen der Argumente und der übergebenen Werte nicht übereinstimmen. Der Aufruf **`vorstellung(alter = 25, name =`**

"Ulrike") schließlich dreht die Argumente beim Aufruf der Funktion zwar auch herum, aber durch die Angabe der Argumentennamen wird deutlich, wie die übergebenen Werte den Argumenten der Funktion zugeordnet werden sollen, sodass die abweichende Reihenfolge der Argumente beim Funktionsaufruf kein Problem darstellt.

### ■ Aufgabe 13.3

- Das Programm wird folgende Ausgaben erzeugen:

```
Neues Alter der Person: 26  
Derzeitiges Alter: 25  
Ergebnis der Funktion werdeAelter(): 26
```

Die Funktion **werdeAelter()** erhöht den Wert des ihr übergebenen Arguments **alter\_person** und gibt diesen als Funktionswert zurück. Die gleichnamige *globale* Variable **alter\_person** bleibt davon unberührt. Vorrang im Gültigkeitsbereich der Funktion hat das Argument, das als *lokale* Variable behandelt wird.

- Die erste Möglichkeit besteht darin, den Rückgabewert der Funktion **werdeAelter()** in der globalen Variable **alter\_person** aufzufangen:

```
alter_neu = werdeAelter(alter_person)
```

Alternativ kann das Argument **alter** der Funktion als ein Argument definiert werden, das *als Referenz* übergeben wird (sofern die verwendete Programmiersprache das zulässt). Dann würde der Funktionskopf so aussehen: **Funktion werdeAelter(AlsReferenz alter\_person)**. Auf diese Weise würde das Argument **alter\_person** zwar weiterhin als lokale Variable betrachtet werden (außerhalb des Code-Blocks der Funktion könnte man nicht auf diese Variable zugreifen), aber Änderungen an ihr würden unmittelbar zu Änderungen an der übergebenen Variable (also **alter**) führen. So würde die Funktion **werdeAelter()** letztlich die globale Variable **alter** verändern können.



# Wie steuere ich den Programmablauf und lasse das Programm auf Benutzeraktionen und andere Ereignisse reagieren?

## Inhaltsverzeichnis

- 14.1 Warum eine Ablaufsteuerung notwendig ist – 171**
- 14.2 Formen der Ablaufsteuerung – 172**
- 14.3 Wenn-Dann-Sonst-Konstrukte – 173**
- 14.4 Ein genauerer Blick auf Bedingungen – 179**
- 14.5 Komplexe Bedingungen mit logischen Operatoren (and, or, not) – 181**

- 14.6 Gleichartige Bedingungen mit Verzweige-Falls-Konstrukten effizient prüfen (switch/select... case) – 184**
- 14.7 Ereignisse (Events) – 187**
- 14.8 Ihr Fahrplan zum Erlernen einer neuen Programmiersprache – 191**
- 14.9 Lösungen zu den Aufgaben – 192**

## Übersicht

Programme müssen flexibel auf neue Situationen reagieren können, zum Beispiel, wenn der Benutzer eine Eingabe macht oder auf einen Button klickt. Je nachdem, welche Eingabe er macht oder auf welchen Button er klickt, werden dann im Programm unterschiedliche Programmanweisungen ausgeführt, das Programm verzweigt also in unterschiedliche Pfade/Äste. Mit dieser Form der Ablaufsteuerung, die Programme überhaupt erst dynamisch macht, werden wir uns nun eingehender beschäftigen.

In diesem Kapitel werden Sie folgendes lernen:

- wie Sie in Abhängigkeit von einer Bedingung in den einen oder den anderen Programmteil verzweigen
- wie man solche Bedingungen formuliert und wie sie daraufhin geprüft werden, ob sie erfüllt sind, oder nicht
- wie Sie komplexe Bedingungen, die aus mehreren Teilbedingungen zusammengesetzt sind, formulieren
- welche Möglichkeiten es gibt, eine ganze Reihe gleichartig aufgebauter Bedingungen auf einfache und übersichtliche Art und Weise zu prüfen
- wie Sie auf Ereignisse reagieren können, von denen Sie a priori noch gar nicht wissen können, wann genau im Programmablauf sie ausgelöst werden (zum Beispiel der Klick auf einen Button).

## 14.1 Warum eine Ablaufsteuerung notwendig ist

In den vorangegangenen Kapiteln haben Sie gesehen, was Sie benötigen, um Daten vom Benutzer entgegenzunehmen, zu verarbeiten und die Ergebnisse dieser Verarbeitung wieder auszugeben. Trotzdem wären Programme, die Sie nur mit Hilfsmitteln aus diesem Werkzeugkasten schreiben, in ihren Möglichkeiten und damit ihrer Nützlichkeit sehr beschränkt, denn der Programmablauf wäre vollkommen starr: Er würde *immer* mit einer Eingabe beginnen, die sodann *stets auf dieselbe Weise* vom Programm verarbeitet wird, und deren Resultat schließlich *immer in der gleichen Form* an den Benutzer zurückgemeldet wird. Solcherart unflexible Programme würden wir aber in der Realität nie akzeptieren.

Stellen Sie sich vor, Sie würden ein neues Ziel in das Navigationssystem ihres Autos eingeben, das Navigationssystem würde die optimale Route zu diesem Ziel berechnen, aber sobald Sie versehentlich von dieser optimalen Route abweichen, würde das Navigationssystem strikt an der ursprünglichen Wegführung festhalten, obwohl Sie sich ja derzeit gar nicht mehr auf dieser Route befinden und jetzt eigentlich Hilfe bräuchten, um zunächst einmal auf den vom Navigationssystem errechneten Kurs zurückzukommen (oder auf eine ganz neue Route, ausgehend von Ihrem aktuellen Standort, zu finden).

Ähnlich merkwürdig käme uns das Verhaltens unseres Navigationssystems vor, wenn wir die Einstellung vornehmen würden, dass das Navigationssystem bei seiner Routenberechnung Mautstraßen vermeiden soll, das System aber diese Anwei-

sung komplett ignorieren und uns auf unserem Weg in den Urlaub, etwa in Österreich, der Schweiz, Italien oder Frankreich einfach schnurstracks auf die erstbeste kostenpflichtige Autobahn lotsen würde. Wir würden das wohl (und vollkommen zurecht) für einen Fehler im Navigationssystem halten.

Was wir also benötigen, ist eine Möglichkeit, ein Programm auf Ereignisse („Fahrer hat vorgeschlagene Route verlassen“) reagieren zu lassen und Bedingungen zu berücksichtigen („Fahrer will Mautstraßen vermeiden“).

In beiden Fällen werden bestimmte Teile des Programms nur dann ausgeführt, wenn eben das entsprechende Ereignis eingetreten, bzw. die entsprechende Bedingung erfüllt ist. Das Programm *verzweigt* also in unterschiedliche Äste, je nachdem, wie die aktuelle Situation sich gerade darstellt. Mit dieser Art der *Ablaufsteuerung* des Programms beschäftigen wir uns in diesem Kapitel.

## 14.2 Formen der Ablaufsteuerung

---

In der Praxis realisiert man die Ablaufsteuerung des Programms meist durch Einsatz sogenannter *Wenn-Dann-(Sonst-)Konstrukte*: *wenn* eine bestimmte Bedingung erfüllt ist, *dann* tue dies, *sonst* tue etwas anderes.

Die dabei geprüften Bedingungen können auch *komplexe Bedingungen* sein, die sich aus mehreren Teilbedingungen zusammensetzen. Eine solche komplexe Bedingung könnte in unserem Navigationssystem-Beispiel aus dem letzten Abschnitt lauten: „Wenn ‚Mautstraßen vermeiden‘ eingestellt ist *UND* die nächste Abzweigung auf eine Mautstraße führt, dann empfehle keinesfalls die Ausfahrt auf diese Abzweigung“. Ein weiteres Beispiel wäre: „Wenn die nächste Abzweigung keine Mautstraße ist *ODER* das Befahren von Mautstraßen erlaubt ist, dann empfehle die nächste Abzweigung“. Hier werden jeweils zwei Teilbedingungen durch *UND* bzw. *ODER*, sogenannte logischen Operatoren, miteinander verknüpft.

Manchmal hat man eine ganze Reihe gleichartiger Bedingungen zu prüfen, zum Beispiel, wenn der Benutzer eine Ziffer eingeben und auf jede Ziffer anders reagiert werden soll. Die Bedingung ist dabei strukturell stets dieselbe, nämlich „Eingabe ist gleich Ziffer X“ (der Programmcode, der ausgeführt wird, wenn eine bestimmte dieser Bedingungen erfüllt ist, mag natürlich jeweils ganz verschieden sein, je nachdem, welche Ziffer eingegeben wurde). Solche Anwendungsfälle von Bedingungen lassen sich natürlich mit Hilfe von Wenn-Dann-(Sonst-)Konstrukten realisieren, die aber regelmäßig zu einem komplizierten, schwer zu lesenden (und damit auch schwer zu wartbaren) Code führen. Deshalb kennen viele Programmiersprachen ein *Verzweige-Falls-Konstrukt*, mit dem sich die Prüfung solcher gleichartigen Bedingungen sehr einfach und übersichtlich umsetzen lässt.

Nicht immer ist der Programmablauf linear und man kommt nach einer genau definierten Folge von Anweisungen an die Stelle, an der eine Bedingung geprüft und dann in Abhängigkeit vom Ergebnis der Prüfung in den einen oder den anderen Programmteil verzweigt wird. Die wichtigste Ursache für solche nicht-linearen Programmabläufe ist der Benutzer (ohne den Benutzer wäre Programmieren viel einfacher!). Er kann beispielsweise auf einer grafischen Benutzeroberfläche aus ei-

ner Vielzahl unterschiedlicher Programmfunctionen wählen und dabei relativ frei entscheiden, zu welchem Zeitpunkt er welche Funktionen in welcher Reihenfolge aufruft. Das bedeutet, der Benutzer bestimmt den Ablauf des Programms, und das Programm kann nicht einfach nur die sture Abarbeitung einer langen Folge von Anweisungen sein, sondern muss irgendwie flexibler aufgebaut sein. Diese Art der Ablaufsteuerung führt uns zum Konzept der *Ereignisse* und des ereignisorientierten Programmier-Paradigmas, das in vielen Programmiersprachen Anwendung findet.

Mit diesen Elementen der Ablaufsteuerung, den Wenn-Dann-Sonst-Konstrukten, den in ihnen verwendeten Bedingungen, der Verknüpfung mehrerer Teilbedingungen zu einer komplexeren Gesamtbedingung, den Verzweige-Falls-Konstrukten und der Ereignissteuerung beschäftigen wir uns im Rest dieses Kapitels.

### 14.3 Wenn-Dann-Sonst-Konstrukte

Stellen Sie sich vor, wir würden gerade an einer Software für Online-Banking arbeiten, also den Programmen, die hinter der Online-Banking-Website einer Bank ablaufen.

Hier beschäftigen wir uns mit der Abfrage des Kontostands. Dabei soll der Benutzer eine Warnung angezeigt bekommen, wenn sich sein Konto im Soll befindet, der Kontostand also negativ ist.

Ein Programm, dass genau das leistet, könnte so aussehen:

```
kontostand = KontostandAbfragen()
Wenn kontostand < 0 Dann Anzeigen("Achtung: Ihr Kontostand ist
negativ!")
Anzeigen("Aktuelles Guthaben: ", kontostand, " EUR")
```

Hier weisen wir zunächst einer Variablen **kontostand** den Wert der Funktion **KontostandAbfragen()** zu. Wir wollen annehmen, diese Funktion gebe uns den aktuellen Kontostand als Dezimalzahl zurück.

Nachdem wir den Kontostand in der gleichnamigen Variablen gesichert haben, prüfen wir mit **Wenn kontostand < 0**, ob der Kontostand negativ ist. Ist das der Fall, wird eine Warnmeldung ausgegeben, was wir hier mit der Funktion **Anzeigen()** bewerkstelligen. Danach geben wir mit **Anzeigen("Aktuelles Guthaben: ", kontostand, "EUR")** den Kontostand selbst, also den Inhalt der Variable **kontostand**, aus. Beachten Sie, dass diese Anweisung immer ausgeführt wird, unabhängig davon, ob das Konto im Soll ist oder nicht. Die einzige Anweisung, die von der Bedingung abhängig ist, dass der Kontostand einen negativen Wert hat, ist die Anzeige der Warnmeldung hinter dem Schlüsselwort **Dann**. Alles, was danach folgt, wird auf jeden Fall ausgeführt.

Angenommen nun, der Kontostand sei 1000 EUR. In diesem Fall wäre der Output unseres Programms:

```
1000 EUR
```

Wäre das Konto aber im Soll, zum Beispiel bei einem Kontostand von -280 EUR, würde der Benutzer folgende Ausgabe erhalten:

```
Achtung: Ihr Kontostand ist negativ!
-280 EUR
```

An diesem einfachen Beispiel sehen Sie bereits sehr schön, wie das Programm verzweigt und bestimmte Teile – in unserem Fall die Warnung – nur dann ausführt, wenn eine bestimmte Bedingung erfüllt ist.

Im nächsten Schritt erweitern wir das Programm so, dass zusätzlich angezeigt wird, wieviel vom Dispositionskredit (Dispo), von dem wir annehmen wollen, dass er 500 EUR beträgt, noch übrig ist, *sofern* das Konto im Soll ist. Ist das Konto dagegen im Haben, der Kontostand also größer als 0, soll in Bezug auf den Dispositionskredit keine Nachricht angezeigt werden.

Die Erweiterung unseres Programms könnte so aussehen:

```
kontostand = KontostandAbfragen()
Wenn kontostand < 0 Dann
    Beginn
        Anzeigen("Achtung: Ihr Kontostand ist negativ!")
        disporest = 500 + kontostand
        Anzeigen("Sie habe noch ", disporest, " EUR
                    Dispositionskredit.")
    Ende
Anzeigen("Aktuelles Guthaben: ", kontostand, " EUR")
```

Um den Dispo-Stand abzubilden, erzeugen wir eine Variable namens **disporest**, der wir die Summe aus 500 EUR und **kontostand** zuweisen. Ist der Kontostand negativ (und nur dann wird dieser Programmteil durchlaufen) ergibt die Summe gerade den verbleibenden Betrag auf dem Dispositionskredit.

Anders als im vorangegangenen Beispiel folgen hier auf die Prüfung, ob das Konto im Soll ist, *mehrere* Anweisungen, die in einen Code-Block zwischen den Schlüsselwörtern **Beginn** und **Ende** eingefasst sind. Code-Blöcke haben wir bereits im Zusammenhang mit Funktionen in ► Abschn. 13.1 kennengelernt. Alle Anweisungen des Code-Blocks werden nur dann ausgeführt, wenn die Bedingung **kontostand < 0** erfüllt ist. Die Anweisung **Anzeigen(kontostand, "EUR")**, mit der wir den aktuellen Kontostand ausgeben, ist nicht eingerückt und wird deshalb immer ausgeführt, egal, ob das Konto im Soll oder im Haben ist.

Wenn der Code-Block nur eine Zeile umfasst, können in vielen Programmiersprachen die Begrenzer (in unserem Pseudo-Code **Beginn** und **Ende**) auch weggelassen werden, wie wir es in den Beispielen oben getan haben.

### 14.3 · Wenn-Dann-Sonst-Konstrukte

Nehmen wir wieder unser Beispiel mit einem negativen Kontostand in Höhe von -280 EUR. In diesem Fall produziert unser Programm nun folgenden Output:

```
Achtung: Ihr Kontostand ist negativ!
Sie habe noch 220 EUR Dispositionskredit.
Aktuelles Guthaben: -280 EUR
```

Im Falle eines positiven Kontostandes von 1000 EUR reduziert sich die Anzeige entsprechend:

```
Aktuelles Guthaben: 1000 EUR
```

Angenommen nun, wir wollten auch etwas anzeigen, wenn der Kontostand positiv (oder gleich 0) ist, zum Beispiel: „Ihr Konto ist im Haben.“

Das lässt sich mit den bisher kennengelernten Mitteln leicht umzusetzen:

```
kontostand = KontostandAbfragen()
Wenn kontostand < 0 Dann
    Beginn
        Anzeigen("Achtung: Ihr Kontostand ist negativ! ")
        disporest = 500 + kontostand
        Anzeigen("Sie haben noch ", disporest, " EUR
                    Dispositionskredit.")
    Ende

Wenn kontostand >= 0 Dann
    Beginn
        Anzeigen("Ihr Konto ist im Haben.")
    Ende

Anzeigen("Aktuelles Guthaben: ", kontostand, " EUR")
```

Allerdings können wir das auch einfacher bewerkstelligen, wenn wir berücksichtigen, dass jede der Bedingungen **kontostand < 0** und **kontostand >= 0** ja gerade der Umkehrfall der jeweils anderen ist, beide zusammen also alle möglichen Fälle abdecken. In dieser Situation können wir darauf verzichten, die zweite Bedingung explizit zu formulieren:

```
kontostand = KontostandAbfragen()
Wenn kontostand < 0 Dann
    Beginn
        Anzeigen("Achtung: Ihr Kontostand ist negativ!")
        disporest = 500 + kontostand
        Anzeigen("Sie haben noch ", disporest, " EUR
```

```

        Dispositionskredit.")

Ende
Sonst
Beginn
    Anzeigen("Ihr Konto ist im Haben.")
Ende

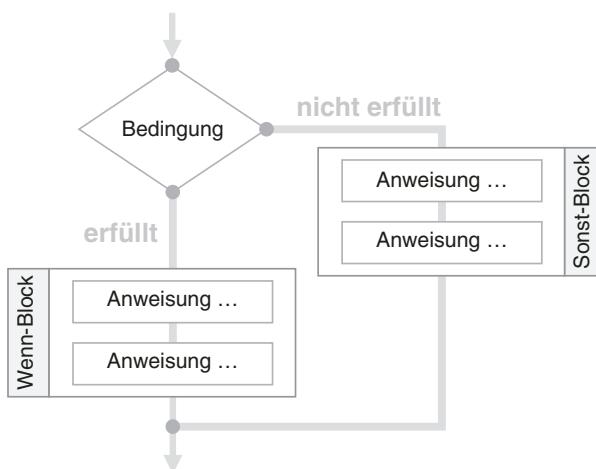
Anzeigen("Aktuelles Guthaben: ", kontostand, " EUR")

```

Hier zeigt uns das Schlüsselwort **Sonst** an, dass jetzt der Code-Block beginnt, der nur dann ausgeführt wird, wenn die Bedingung oben (**kontostand < 0**) nicht erfüllt ist, wenn also der Kontostand größer oder gleich 0 ist. Wir haben damit zwei Code-Blöcke: Den Block zwischen **Wenn kontostand < 0 Dann** und **Ende** und den zwischen **Sonst** und **Ende**. Bei jeder Ausführung des Programms wird nur *ein* Block durchlaufen. Das Programm verzweigt also dieser Stelle. Ist die Bedingung **Wenn Kontostand < 0** erfüllt, macht es erst mal weiter, zeigt die Warnmeldung an, berechnet den Dispo-Rest und zeigt auch diesen an. Dann stößt es auf das **Sonst**. Da ja bereits die erste Bedingung erfüllt war, wird der **Sonst**-Block übersprungen und die Ausführung erst bei der nächsten Anweisung nach diesem Block, in unserem Fall **Anzeigen("Aktuelles Guthaben: ", kontostand, "EUR")** fortgesetzt.

Eine schematische Darstellung des Ablaufs eines Wenn-Dann-Konstrukts sehen Sie in □ Abb. 14.1.

Solche Wenn-Dann-Sonst-Konstrukte kennen die meisten Programmiersprachen. Die Struktur dieser Konstrukte ist dabei meist sehr ähnlich. Um das zu verdeutlichen schauen wir uns einmal an, wie man das obige Problem in drei unterschiedlichen Programmiersprachen lösen würde.



□ Abb. 14.1 Ablaufschema eines Wenn-Dann-Konstrukt

Zunächst in C/C++:

```
if (kontostand < 0)
{
    printf("Achtung: Ihr Kontostand ist negativ!");
    float disporest = 500 + kontostand;
    printf("Sie haben noch %f EUR Dispositionskredit.", 
           disporest);
}
else
{
    printf("Ihr Konto ist im Haben.");
}
printf("Aktuelles Guthaben: %f EUR", kontostand);
```

Als nächstes in Python:

```
if kontostand < 0:
    print("Achtung: Ihr Kontostand ist negativ!")
    disporest = 500 + kontostand
    print("Sie haben noch ", disporest,
          " EUR Dispositionskredit.")
else:
    print("Ihr Konto ist im Haben.")

print("Aktuelles Guthaben: ", kontostand, " EUR");
```

Und schließlich noch in Visual Basic for Applications (VBA):

```
If kontostand < 0 Then
    MsgBox ("Achtung: Ihr Kontostand ist negativ!")
    disporest = 500 + kontostand
    MsgBox ("Sie haben noch " & disporest &
           " EUR Dispositionskredit.")
Else
    MsgBox ("Ihr Konto ist im Haben.")
End If
MsgBox ("Aktuelles Guthaben: " & kontostand & " EUR")
```

Wenn Sie die drei Beispiele vergleichen, wird Ihnen zunächst die grundsätzlich sehr ähnliche Struktur auffallen. Alle drei Sprachen kennen ein Wenn-Dann-Sonst-Konstrukt. Die Umsetzung unterscheidet sich nur im Detail:

- **Schlüsselwörter:** In allen drei Sprachen existieren die Schlüsselwörter **if** und **else** für **Wenn** und **Sonst**. Ein explizites **Dann** wird als **then** allerdings nur in VBA geschrieben.
- **Bedingungen:** C/C++ erfordert, dass die Bedingung in Klammern geschrieben wird, was die beiden anderen Sprachen nicht tun. Dort schaden Klammern allerdings auch nicht; den Grund dafür werden wir weiter unten noch kennenlernen.

- **Code-Blöcke:** Erhebliche Unterschiede bestehen allerdings bei der Frage, wie die Code-Blöcke, die ausgeführt werden sollen, wenn die Bedingung erfüllt ist (Wenn-Fall), bzw. wenn die Bedingung nicht erfüllt ist (Sonst-Fall) abgegrenzt werden: In C/C++ werden geschweifte Klammern verwendet, um Beginn *und* Ende eines Code-Blocks zu markieren. In Python dagegen beginnen die Code-Blöcke mit einem Doppelpunkt und sind eingerückt. Alle Anweisungen auf der gleichen Einrückungsebene werden als Bestandteil des Code-Blocks betrachtet. Deshalb bedarf es auch am Ende des **else**-Blocks keines besonderen Schlüsselwortes mehr. Hier muss man also genau darauf achten, wie der Code formatiert ist, die Einrückungen haben eine Bedeutung und man kann deshalb nicht einfach einrücken, wie man möchte. Im VBA-Beispiel ist der liegen die Blöcke zwischen dem **Then** und dem **Else**, sowie zwischen dem **Else** und dem **End If**. Beachten Sie hierbei, dass das **End If**, anders als das weiter oben verwendete **Ende** nicht das Ende des Wenn-Blocks markiert, also desjenigen Code-Teils, der ausgeführt wird, wenn die Bedingung **kontostand < 0** erfüllt ist, sondern das Ende des gesamten Wenn-Dann-Sonst-Konstrukt. Der Wenn-Block endet einfach beim **Else**. Nur wenn kein **Else**-Zweig existiert, markiert das **End If** das Ende des Wenn-Blocks.

### 14.1 [3 min]

Was gibt das folgende Programm aus, wenn

- x = 10**
- x = 11**
- x = 25**
- x = -1**

eingegeben wird?

14

```

x = eingeben("Bitte geben Sie eine Zahl ein: ")
Wenn x > 10 Dann
    Beginn
        Wenn x > 20 Dann anzeigen("Ergebnis A")
        Anzeigen("Ergebnis B")
    Ende
Sonst
    Beginn
        Wenn x > 0 Dann anzeigen("Ergebnis B")
    Ende

```

### 14.2 [3 min]

Verändern Sie den Programmausschnitt aus Aufgabe 14.1 so, dass **Ergebnis B** nur angezeigt wird, wenn **x** größer als 10 und kleiner oder gleich 20 ist.

## 14.4 Ein genauerer Blick auf Bedingungen

Im vorangegangenen Abschnitt haben Sie gesehen, wie sich mit Hilfe von Wenn-Dann-Sonst-Konstrukten in verschiedene Programmteile verzweigen lässt, in Abhängigkeit davon, ob eine bestimmte Bedingung erfüllt ist, oder nicht. In folgenden wollen wir uns die Bedingungen, die in den Wenn-Dann-Sonst-Konstrukten geprüft werden, etwas genauer ansehen.

Die große Gemeinsamkeit aller Bedingungen ist, dass ihr Ergebnis entweder *wahr* oder *falsch*, die Bedingung als erfüllt oder nicht erfüllt ist. Jeder logische Ausdruck, der sich als wahr oder falsch bewerten lässt, ist damit geeignet, als Bedingung verwendet zu werden. Häufig sind solche logischen Ausdrücke Wertevergleiche, wie wir es auch im Online-Banking-Beispiel des vorangegangenen Abschnitts gesehen haben. Dort wurde beispielsweise überprüft, ob der Kontostand kleiner als 0 ist. Solche Wertevergleiche lassen sich mit den aus der Mathematik bekannten Vergleichsoperatoren wie  $>$  (größer),  $<$  (kleiner) und  $=$  (gleich) leicht formulieren. Dabei wird für „größer gleich“ normalerweise  $\geq$  und für „kleiner gleich“  $\leq$  geschrieben, weil die speziellen Zeichen  $\leq$  und  $\geq$  in den Zeichensätzen des frühen Computerzeitalters gar nicht existierten und sie selbst heute, da sie theoretisch zur Verfügung stünden, nicht einfach direkt über die Tastatur eingegeben werden können. Um das schnelle Tippen von Programmen zu ermöglichen, hat man sich daher auf die zusammengesetzte Schreibweise verständigt. Eine besondere Herausforderung dabei stellt das Ungleich-Zeichen  $\neq$  dar, weil sich hier keine natürliche „Übersetzung“ in ein zusammengesetztes Zeichen anbietet. Und tatsächlich haben unterschiedliche Programmiersprachen hierfür unterschiedliche Lösungen gefunden. Die beiden am häufigsten anzutreffenden sind  $<>$  (Kleiner- und Größerzeichen) und  $!=$  (Ausrufezeichen und Gleichheitszeichen).

Im vorangegangenen Abschnitt bestanden die Bedingungen stets aus dem Vergleich einer Variablen mit einem Wert. Tatsächlich ist aber als Bedingung jeder Ausdruck denkbar, der wahr oder falsch sein kann. Stellen Sie sich vor, wir hätten in unserem Online-Banking-Beispiel eine Funktion die prüft, ob der aktuelle Kunde einen positiven Kontostand hat. Deren Rückgabewert wiederum lässt sich in einer Variablen speichern:

```
kontohaben = IstKontostandPositiv()
```

Wir weisen also der Variablen **kontohaben** den Rückgabewert der Funktion **IstKontostandPositiv()** zu. Dementsprechend ist der Wert der Variablen **kontohaben** jetzt entweder **WAHR** oder **FALSCH**. Also können wir den Inhalt der Variablen in einer Bedingung prüfen:

```
Wenn kontohaben = WAHR Dann
Beginn
    Anzeigen("Achtung: Ihr Konto ist im Haben!")
Ende
```

Statt die Variable **kontostandhaben** zu prüfen, erlauben es die meisten Programmiersprachen, auch einfach die Funktion direkt in die Bedingung einzusetzen:

```
Wenn IstKontostandPositiv() = WAHR Dann
Beginn
    Anzeigen("Achtung: Ihr Konto ist im Haben!")
Ende
```

Diese Vorgehensweise bietet sich immer dann an, wenn man mit dem Rückgabewert der Funktion nicht mehr weiterarbeiten will. Möchte man aber an anderer Stelle auf diesen Wert noch einmal zurückgreifen, macht es Sinn, den Wert in einer Variablen zu speichern. Das erspart nämlich einen abermaligen Aufruf der Funktion und damit Rechenleistung und Zeit: Ist der Programmcode, der hinter der Funktion steht, sehr komplex, läuft Ihr Programm schneller, wenn Sie einfach auf den in der Variablen gespeicherten Wert zurückgreifen, anstatt ihn durch einen Funktionsaufruf noch einmal neu berechnen zu lassen.

Eine weitere Vereinfachung, die die meisten Programmiersprachen erlauben, ist es, den expliziten Vergleich mit dem logischen Wert **WAHR** einfach fallen zu lassen. Die Annahme ist also, dass immer, wenn ein Wert (entweder der Wert einer Variablen, der Rückgabewert einer Funktion oder das Ergebnis eines wie auch immer berechneten Ausdrucks) in einer Bedingung verwendet wird, ohne ihn ausdrücklich mit einem anderen Wert zu vergleichen, der Vergleich mit dem logischen Wert **WAHR** vorgenommen werden soll. In unserem Beispiel also könnten wir schreiben:

```
Wenn IstKontostandPositiv() Dann
Beginn
    Anzeigen("Achtung: Ihr Konto ist im Haben!")
Ende
```

Hat man der Funktion einen sprechenden Namen gegeben, wie wir es hier getan haben, lässt sich die Bedingung sehr einfach lesen und verstehen.

Wie man leicht sehen kann, ist auch ein normaler Wertevergleich letztlich immer ein Vergleich mit dem Wert **WAHR** und damit die Prüfung einer Bedingung nach dem gängigen Schema. Denn statt etwa

```
Wenn kontostand > 1000000 Dann
```

könnte man auch schreiben:

```
Wenn (kontostand > 1000000) = WAHR Dann
```

Dann würde zunächst der Ausdruck auf der linken Seite ausgewertet werden. Wäre der Kontostand nun größer als eine Million, würde der Ausdruck den Wert **WAHR** annehmen und die Bedingung wäre erfüllt.

Bedingungen sind also letztlich stets Vergleiche mit dem Wert **WAHR**.

## 14.5 Komplexe Bedingungen mit logischen Operatoren (and, or, not)

In den Beispielen der vorangegangenen Abschnitte war es stets eine einzige elementare Bedingung, die in einem Wenn-Dann-Block darüber entschied, ob ein Programmteil ausgeführt wird, oder nicht. Natürlich kann die Bedingung in einem Wenn-Dann-Block auch eine aus mehreren Teilbedingungen zusammengesetzte sein.

Nehmen wir beispielsweise an, wir wollten in unserem Online-Banking-Beispiel eine Überweisung nur dann erlauben, wenn das Konto nicht gesperrt ist *und* die Summe aus Kontostand (der ja auch negativ sein kann) und Dispositionskredit wenigstens so hoch ist, wie der zu überweisende Betrag. Wäre also beispielsweise der Kontostand -150 EUR, der Dispositionskredit 500 EUR und der Benutzer unseres Onlinebankings wollte einen Betrag von 50 EUR überweisen, so sollte das Online-Banking dies erlauben, denn die Summe aus Kontostand und Dispositionskredit, und damit der für den Online-Banking-Kunden verfügbare Betrag, würde sich auf 350 EUR belaufen, also mehr, als tatsächlich überwiesen werden soll. Würde der Kunde dagegen 400 EUR überweisen wollen, sollte das Online-Banking diesen Transaktionswunsch zurückweisen, denn der zu überweisende Betrag würde die verfügbare Summe aus Kontostand und Dispositionskredit um 50 EUR übersteigen.

Wir wollen, um uns das Leben einfach zu machen, davon ausgehen, dass wir eine Funktion **IstKontoGesperrt()** zur Verfügung haben, die **WAHR** zurückgibt, wenn das Konto gesperrt ist und **FALSCH**, wenn es nicht gesperrt ist. Dann würde die Bedingung, die prüft, ob der Kunde eine Überweisung vornehmen darf oder nicht, in einem Wenn-Dann-Block so aussehen:

```
Wenn IstKontoGesperrt ()=FALSCH UND kontostand +
dispositionskredit >= betrag Dann
Beginn
...
Ende
```

Hier sehen Sie, dass wir zwei (Teil-)Bedingungen mit einem logischen UND verknüpfen. Die Gesamtbedingung des Wenn-Dann-Blocks ist also nur dann erfüllt, wenn *sowohl* die eine *als auch* die andere Teilbedingung erfüllt sind.

Neben dem UND existieren weitere logische Operatoren, mit deren Hilfe Sie komplexere Bedingungen zusammensetzen können. Zum Beispiel das logische ODER, das zwei (Teil-)Bedingungen dergestalt verknüpft, dass die Gesamtbedingung genau dann erfüllt ist, wenn die eine, die andere, oder beide Teilbedingungen erfüllt ist. Die Bedeutung des logischen ODER ist also eine andere als die des „oder“ in der Alltagssprache, in der regelmäßig ein *ausschließendes* ODER gemeint ist: *Entweder* das eine *oder* das andere, aber nicht beide zusammen.

Wie Sie leicht erkennen können, funktionieren die logischen Operatoren genauso wie die, die Sie aus der Mathematik kennen. Dementsprechend kennen auch die Programmiersprachen einen ausschließendes ODER (oft auch als XODER bezeichnet), das dem umgangssprachlichen „oder“ entspricht.

Ein weiterer wichtiger logischer Operator ist das logische NICHT, das den Wahrheitsgehalt einer Aussage umdreht. Die Umkehrung des Wahrheitsgehalts der Aussage „Das Konto ist gesperrt“ ist offensichtlich „Das Konto ist nicht gesperrt“. Beim Programmieren können wir das NICHT leider meist nicht so elegant zwischen die „Wörter“ unseres „Satzes“ stellen. Deshalb ergibt sich im Programmcode eher so etwas wie „NICHT Das Konto ist gesperrt“. Statt der obigen Bedingung könnten wir also auch schreiben:

```
Wenn NICHT IstKontoGesperrt() UND kontostand +
    dispositionscredit >= betrag Dann
Beginn
...
Ende
```

Wie Sie sich aus dem vorangegangenen Abschnitt erinnern, ist **IstKontoGesperrt()** eine Kurzschreibweise für **IstKontoGesperrt() = WAHR**, Sie können also, wenn Sie auf den Wert **WAHR** prüfen wollen, den expliziten Vergleich mit diesem Wert einfach weglassen, denn darauf wird standardmäßig geprüft, wenn Sie keinen anderen Vergleichswert angeben.

Die so gestaltete Bedingung ermittelt also zunächst den Wert der Funktion **IstKontoGesperrt()**. Mit Hilfe des NICHT-Operators wird dieser Wahrheitswert dann einfach herumgedreht. Ist also das Konto *ungesperrt*, liefert **IstKontoGesperrt()** den Wert **FALSCH**, das logische NICHT verkehrt den Wert zu **WAHR**. Somit verkürzt sich die Bedingung zu **WENN WAHR UND kontostand + dispositionscredit >= betrag**. Der Wahrheitswert der Gesamtbedingung hängt dann davon ab, ob die zweite Teilbedingung erfüllt ist oder nicht.

Natürlich lassen sich logische Ausdrücke wie die obigen auch mit Klammern beliebig komplex verschachteln. Durch die Klammern wird sichergestellt, dass der Inhalt der Klammer zuerst ausgewertet wird, bevor der so ermittelte Wert mit anderen Ausdrücken logisch verknüpft wird.

Hier ein einfaches Beispiel:

```
Wenn NICHT IstKontoGesperrt() UND (kontostand +
    dispositionscredit >= betrag
    ODER IstKundenHistoriePositiv()) Dann
Beginn
...
Ende
```

Bei der zweiten Teilbedingung wird ausgewertet, ob der zu überweisende Betrag der üblichen Voraussetzung genügt oder ob der Kunde eine so positive Historie hat

(etwa bestehend aus regelmäßigen Eingängen auf dem Konto, keinen signifikanten Überziehungen etc.), festgestellt mit Hilfe einer Funktion **IstKundenHistoriePositiv()**. Hätte der Kunde bislang ein vorbildliches Verhalten gezeigt, würde die Transaktion also selbst dann zugelassen werden, wenn sie den Dispositionsräumen überschreitet. Durch die Klammern wird sichergestellt, dass im ersten Schritt zunächst festgestellt wird, ob aufgrund des hinreichend kleinen Überweisungsbetrags oder aufgrund des bisherigen Kundenverhaltens die Transaktion erlaubt werden sollen. Dazu werden zwei Teilbedingungen mit logischem ODER verknüpft. Daraus ergibt sich ein Wahrheitswert, der dann im zweiten Schritt mit dem Wahrheitswert aus der Prüfung, ob das Kundenkonto gesperrt ist, mit logischem UND verknüpft wird.

Wie Sie sich sicher schon gedacht haben, heißen die logischen Operatoren in unterschiedlichen Programmiersprachen verschieden. Oftmals bleibt es bei den eingängigen englischen Schlüsselwörtern **AND**, **OR**, **XOR**, **NOT**, aus denen die Bedeutung des Operators sofort hervorgeht. Einige Sprachen wie C, C++, Java oder R benutzen aber statt der ausgeschriebenen Form besondere Zeichen: Das kaufmännische Und (**&**) für das logische UND, die Pipe (**|**) für das logische ODER und das Ausrufezeichen (**!**) für das logische NICHT. Jetzt erkennen Sie übrigens leicht, warum das Ungleich in diesen Sprachen als **!=** geschrieben wird. Es bedeutet nämlich einfach „NICHT gleich“.

In C, C++, Java oder R würde der obigen Wenn-Dann-Block so lauten:

```
if !IstKontoGesperrt() & (kontostand + dispositionscredit >=
betrag | IstKundenHistoriePositiv()) {
...
}
```

In der Makrosprache Visual Basic für Applikationen (VBA), die es erlaubt, die Anwendungen der Büroanwendungssuite Microsoft Office zu automatisieren, würde man diesen Wenn-Dann-Block unter Verwendung sprechender Bezeichner für die logischen Operatoren dagegen so schreiben:

```
If not IstKontoGesperrt() and (kontostand + dispositionscredit >=
betrag or IstKundenHistoriePositiv())
...
End If
```

### 14.3 [10 min]

Betrachten Sie den folgenden Programmausschnitt:

```
x = eingeben("Bitte geben Sie eine Zahl ein: ")
Wenn x > 100 Dann ausgeben("x größer 100!")
Wenn x > 50 Dann ausgeben("x größer 50!")
```

```

Wenn x > 10 Dann ausgeben("x größer 10!")
Wenn x < 0 Dann ausgeben("x kleiner 0!")
Wenn x >= 0 Und x <= 10 Dann ausgeben("x zwischen 0 und 10!")

```

- Wie viele Bedingungen werden geprüft, wenn der Benutzer einen Wert für x eingibt?
- Verändern Sie den Algorithmus so, dass im besten Fall nur eine Bedingung und nur im ungünstigsten Fall alle Bedingungen geprüft werden.

#### 14.4 [3 min]

Wo liegt der Fehler im folgenden Programmausschnitt?

```

x = eingeben("Bitte geben Sie eine Zahl ein: ")
Wenn x > 100 Dann ausgeben("x größer 100!")
Sonst
  Beginn
    Wenn x >= 110 Dann ausgeben("x größer 110!")
    Sonst ausgeben("x kleiner oder gleich 100!")
  Ende

```

## 14.6 Gleichartige Bedingungen mit Verzweige-Falls-Konstrukten effizient prüfen (switch/select...case)

Manchmal möchte man viele gleichartige Bedingungen auf einmal prüfen. Stellen Sie sich vor, wir wollten die Kunden unseres Online-Bankings danach klassifizieren, wie hoch der monatliche Geldeingang auf ihrem Konto ist, zum Beispiel, um besonders „guten“ Kunden spezielle Services anbieten zu können. Wir wollen annehmen, dass uns zur Abfrage des durchschnittlichen Geldeingangs über die letzten drei Monate eine Funktion **MonatlicherEingang3Monate()** zur Verfügung steht, die ebendiesen Betrag als Funktionswert zurückgibt. Die Einteilung der Kunden könnte jetzt so vonstattengehen:

```

Wenn MonatlicherEingang3Monate() < 1000 Dann
  Beginn
    kategorie="D"
  Ende
Sonst
  Beginn
    Wenn MonatlicherEingang3Monate() < 2000 Dann
      Beginn
        kategorie="C"
      Ende
    Sonst

```

```

Beginn
  Wenn MonatlicherEingang3Monate () < 4000 Dann
    Beginn
      kategorie="B"
    Ende
  Sonst
    Beginn
      kategorie="A"
    Ende
  Ende
Ende

```

Hier werden die unterschiedlichen Grenzwerte für den durchschnittlichen monatlichen Geldeingang in einer verschachtelten Wenn-Dann-Schleife abgebildet. Die ist aber verhältnismäßig schwierig zu lesen. Viele Programmiersprachen kennen ein Konstrukt, das es erlaubt, die Prüfung mehrerer gleichartiger Bedingungen eleganter zu schreiben. In unserem Beispiel würde die Formulierung dann so lauten:

```

Verzweigung MonatlicherEingang3Monate ()
  Beginn
    Fall      < 1000: kategorie="D"
    Fall      < 2000: kategorie="C"
    Fall      < 4000: kategorie="B"
    Sonst:   kategorie="A"
  Ende

```

Diese Schreibweise ist ungleich übersichtlicher und deshalb besser lesbar als auch leichter zu programmieren. Hinter dem Schlüsselwort **Verzweigung** steht zunächst die Variable, die Gegenstand der Prüfung ist. In unserem Fall ist es einfach der Rückgabewert unserer Funktion **MonatlicherEingang3Monate()**. Mit **Fall** wird jeweils eine Bedingung eingeleitet, die zu prüfen ist, zum Beispiel  $<1000$ . Hinter dem Doppelpunkt folgt dann, was in diesem Fall geschehen soll, in unserem Beispiel also das Festlegen der Kundenkategorie. Mit dem speziellen Schlüsselwort **Sonst** können alle übrigen Fälle, die nicht explizit abgeprüft worden sind, „aufgefangen“ werden. Diese Anweisung wird nur dann ausgelöst, wenn keine der anderen Bedingungen gegriffen hat. Greift aber doch eine der anderen Bedingungen, werden die diesem Fall zugeordneten Anweisungen (in unserem Beispiel nur eine, es könnten aber auch mehrere sein) hinter dem Doppelpunkt ausgeführt. Danach wird das gesamte Konstrukt verlassen, das heißt, es wird keine der weiteren Bedingungen geprüft. Stattdessen wird die Ausführung des Programms hinter **Ende-Verzweigung** fortgesetzt.

Die schematische Darstellung des Ablaufs eines Verzweige-Fall-Konstrukts sehen Sie in □ Abb. 14.2.

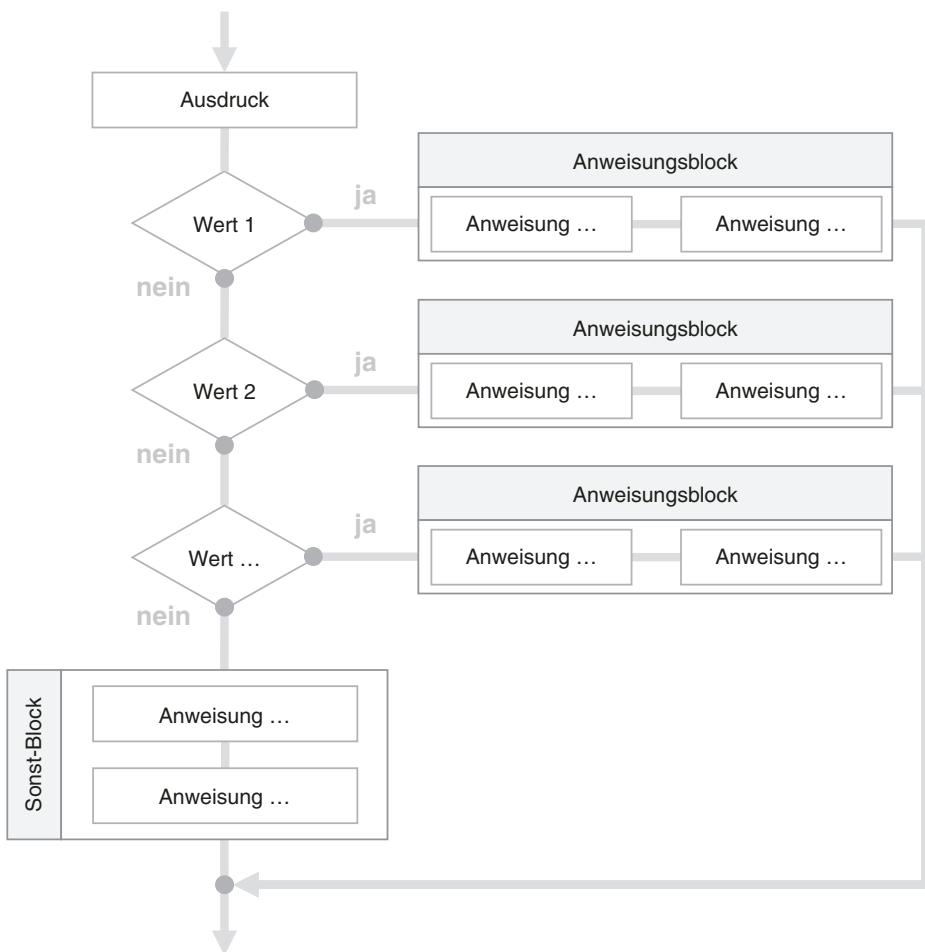


Abb. 14.2 Ablaufschema eines Verzweige-Fall-Konstrukts

14

Viele Programmiersprachen kennen diese elegante Abbildung der Prüfung mehrerer gleichartiger Bedingungen. Typischerweise werden solche Konstrukte als *switch-case-* oder *select-case*-Konstrukte bezeichnet, nach den beiden zentralen Schlüsselwörtern **switch** oder **select** (Verzweigung) und **case** (Fall), die in Sprachen, die dieses Konstrukt kennen, häufig verwendet werden. Verwirrenderweise benutzen einige Sprache das Schlüsselwort **case** anstelle von **switch** oder **select**, aber das soll uns hier nicht weiter beschäftigen.

In C beispielsweise, einer Sprache, die **switch-case**-Konstrukte unterstützt, würde die obige Prüfung des Kundenkategorie dann so lauten:

```
switch(MonatlicherEingang3Monate())
```

```
{  
    case      < 1000: kategorie="D";  
        break;  
    case      < 2000: kategorie="C";  
        break;  
    case      < 4000: kategorie="B";  
        break;  
    default:   kategorie="A";  
        break;  
}
```

## 14.7 Ereignisse (Events)

Neben Wenn-Dann-Bedingungen (einschließlich ihrer speziellen Abbildung als Verzweigung-Fall-Konstrukte, die Sie im vorangegangenen Abschnitt kennengelernt haben) gibt es noch eine weitere wichtige Art, während des Programmablaufs in unterschiedliche Programmteile zu verzweigen, und zwar über sogenannte *Ereignisse*.

Wenn-Dann-Bedingungen werden ausgeführt, wenn das Programm an die entsprechende Stelle gelangt ist, an der die Verzweigungsbedingung steht. Das Programm läuft also vollkommen *sequentiell* ab. Manchmal allerdings weiß man im Voraus noch gar nicht, wann genau im Programmablauf man verzweigen möchte. Dann helfen Ereignisse weiter.

Stellen Sie sich vor, in unserem Online-Banking-Beispiel gäbe es auf der Banking-Website drei Buttons, „Neue Überweisung“, „Umsätze als Textdatei exportieren“ und „Ausloggen“, die der Benutzer jederzeit anklicken kann. Hinter jeder dieser Schaltflächen stehen unterschiedliche Programmanweisungen und damit Funktionalitäten, aber wir wissen a priori nicht, wann – und ob überhaupt – der Benutzer die entsprechenden Funktionen auslöst.

Es gibt grundsätzlich zwei Möglichkeiten, mit diesem Problem umzugehen:

- Entweder das Programm läuft praktisch in einer Endlosschleife und beobachtet jederzeit aktiv das Verhalten des Benutzers, das heißt, es prüft, ob der Benutzer die Schaltfläche angeklickt hat oder nicht. Mit Schleifen werden wir uns erst etwas später in diesem Teil des Buchs befassen, aber die Grundidee einer sich praktisch unendlich wiederholenden Schleife, die an ihrem Ende immer wieder an ihren Anfang zurückspringt und von Neuem abläuft, sollte einigermaßen verständlich sein. Sobald der Benutzer nun tatsächlich auf eine der Schaltflächen klickt, führt das Programm den hinter der Schaltfläche stehenden Programmcode aus und kehrt danach in die Unendlich-Schleife, in der es die Aktionen des Benutzers ständig überwacht, zurück.
- Die zwei Möglichkeit besteht darin, dass das Programm nicht in einer Unendlich-Schleife auf den Benutzer wartet, sondern tut, was immer es auch tun soll, es aber sofort, wenn es von außen erfährt, dass der Benutzer einen der Buttons angeklickt hat, in eine Funktion springt, die dann die hinter der Schaltfläche stehenden Anweisungen ausführt.

Der Unterschied zwischen beiden Ansätzen besteht also darin, dass im ersten Fall das Warten auf den Benutzer und die Überwachung seiner Aktivitäten das Programm praktisch blockiert. Es befindet sich jederzeit in einem *aktiven Überwachungszustand* und prüft unentwegt, ob gerade auf eine der Schaltflächen geklickt worden ist. Im zweiten Fall erfährt das Programm *von außen*, dass ein bestimmtes Ereignis (engl. *event*) eingetreten ist. Sobald das geschehen ist, wird eine Funktion, die mit diesem Ereignis verbunden ist, ausgeführt. Solche Funktionen bezeichnet man auch als *event handler*, weil sie beschreibt, wie mit dem Ereignis umgegangen wird.

Dieser Ansatz spart Rechenleistung, denn das aktive Beobachten, das der erste Ansatz erfordert, ist wie aktives Zuhören: Ständig müssen die Signale abgefragt und verarbeitet werden, während bei der Eventsteuerung einfach ein anderer, zum Beispiel das Betriebssystem oder der Interpreter, Bescheid gibt, sobald ein Ereignis eingetreten ist. Das erlaubt es dem Programm sogar, zwischenzeitlich etwas zu anderes tun, und trotzdem auf ein Ereignis zu reagieren, wenn es eintritt.

Die Funktionsweise ist vergleichbar mit der eines Backofens, den man auf eine bestimmte Temperatur vorheizen muss, um zum Beispiel eine Pizza zu backen. Erst wenn die Vorheiztemperatur erreicht ist, kann die Pizza in den Ofen geschoben werden. Man könnte jetzt alle paar Minuten zum Backofen laufen und nachschauen, ob die Vorheiztemperatur bereits erreicht ist. Oder aber man hat einen Backofen, der selbstständig ein akustisches Signal gibt, wenn er soweit ist. Im ersten Fall befinden wir uns in der Unendlichschleife und schauen wieder und immer wieder selbst nach, ob der Ofen die Vorheiztemperatur bereits erreicht hat. Im zweiten Fall informiert uns ein anderer (hier der Ofen selbst), dass das Ereignis „Vorheiztemperatur ist erreicht“ eingetreten ist. Solange das nicht geschehen ist, können wir einfach etwas anderes machen, denn wir wissen ja, dass sich der Ofen schon meldet, wenn es soweit ist.

Viele Programmiersprachen unterstützen die Verwendung von Ereignissen, man spricht in diesem Zusammenhang vom *ereignisorientierten Programmierparadigma*. Sprachen, die die Verarbeitung von Ereignissen unterstützen, sind insbesondere Sprachen wie etwa JavaScript, die dazu verwendet werden, grafische Benutzeroberflächen zu entwickeln, bei denen man eben gerade nicht den Vorteil eines vollkommen sequentiellen Programmablaufs hat, sondern aufgrund des unvorhersehbaren Verhaltens des Benutzers *a priori* noch nicht genau weiß, welcher Programmteil als nächstes ausgeführt werden muss. Überhaupt sind grafische Benutzeroberflächen, wie wir sie in ► Abschn. 12.2.1 kennengelernt haben, die prototypische Anwendung einer ereignisorientierten Herangehensweise schlechthin.

Die schematische Darstellung eines ereignisgesteuerten Programms sehen Sie in □ Abb. 14.3.

Auslöser von Ereignissen kann übrigens nicht nur der Benutzer sein. Auch das Betriebssystem oder angeschlossene Geräte können Ereignisse auslösen, auf die die Programme dann reagieren können. So kann beispielsweise das Betriebssystem mit einem speziellen Ereignis ankündigen, dass es jetzt herunterfahren möchte und gibt Programmen, die einen für dieses Ereignis passenden Event Handler haben, die Möglichkeit, zu reagieren und zum Beispiel den aktuellen Zustand des Pro-

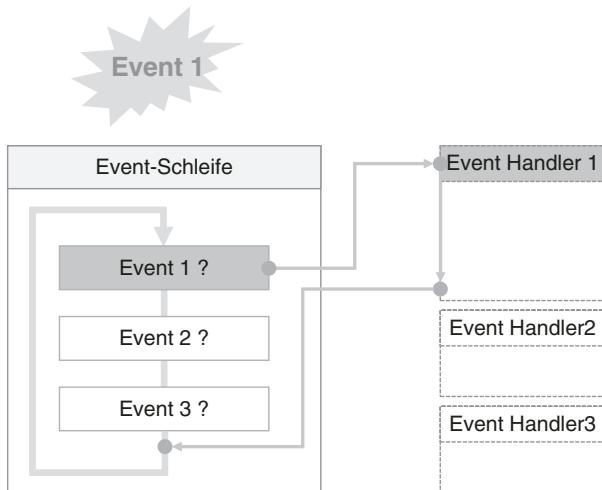


Abb. 14.3 Ablaufschema eines ereignisgesteuerten Programms

gramms zu speichern, bevor das Programm geschlossen wird. In ähnlicher Weise können Geräte Ereignisse auslösen, etwa ein Drucker oder Festplattenlaufwerk.

Bisher haben wir uns zwar mit der grundlegenden Funktionsweise von Event Handlern beschäftigt, aber wie sehen solche Funktionen nun aus? Hier ein Ausschnitt aus dem oben skizzierten Beispiel:

```

Funktion EreignisUeberweisung(Ereignis e)
Beginn
    ErzeugeTransaktion()
Ende

Funktion EreignisLogout(Ereignis e)
Beginn
    Logout()
Ende

```

Hier werden zwei Event Handler definiert, einer für das Ereignis, dass der Benutzer auf den Button „Neue Überweisung“ geklickt hat, einer für den Fall, dass er sich ausloggen möchte. Wie Sie hier sehen, sind die Event Handler einfach normale Funktionen. In manchen Sprachen erhalten Event Handler ein spezielles Objekt als Funktionsargument (hier das Argument e vom Typ Ereignis) übergeben, dass das Ereignis näher beschreibt. So könnten zum Beispiel bei einem Ereignis, das immer dann ausgelöst wird, wenn der Benutzer die Maus bewegt, die aktuellen „Koordinaten“ des Mauszeigers als Eigenschaften des Objekts mitgegeben werden. Die Event-Handler-Funktion kann diese Koordinaten dann aus dem Objekt abfragen und entsprechend darauf reagieren.

Von den Funktionen, die wir im ► Kap. 13 kennengelernt haben, unterscheiden sich diese hier lediglich dadurch, dass sie nicht von uns selbst aufgerufen werden, sondern „von außen“. Der Interpreter oder das Betriebssystem melden sich bei uns, wenn unser Ereignis eingetreten ist, und das tun sie, indem sie unseren Event Handler aufrufen, wobei sie ihm als zusätzliche Information noch das Ereignis-Objekt e mitgeben. Letztlich sind Event Handler also einfach nur Funktionen, die wir zwar entwickeln, aber nicht selbst aufrufen, sondern nur für denjenigen bereitstellen, der uns den Eintritt des Ereignisses signalisiert.

Die obigen Beispiele würden in JavaScript so aussehen:

```
function EreignisUeberweisung() {
    ErzeugeTransaktion();
}

function EreignisLogout() {
    Logout();
}
```

Damit der Interpreter weiß, bei welchem Ereignis er diese Event Handler aufrufen muss, müssten die Button-Definitionen im HTML-Quellcode der Website wie folgt lauten:

```
<button onclick="EreignisUeberweisung()">Neue Überweisung
</button>
<button onclick="EreignisLogout()">Abmelden</button>
```

Auf diese Weise wird das (fest definierte) Ereignis **onclick**, das immer dann ausgelöst wird, wenn jemand auf den Button klickt, mit unserem jeweiligen Event Handler verknüpft.

In Delphi würden unsere Event Handler so aussehen:

```
procedure BankingForm.AusloggenButtonClick(Sender: TObject);
begin
    ErzeugeTransaktion();
end;

procedure BankingForm.NeueUeberweisungButtonClick(
    Sender: TObject);
begin
    Logout();
end;
```

Hier hätten wir ein Fenster (eine sogenannte „Form“) namens **BankingForm** definiert, auf dem wir die zwei Buttons mit den Namen **NeueUeberweisungButton** und **AusloggenButton** platziert hätten. Die Event Handler tragen nicht den

## 14.8 • Ihr Fahrplan zum Erlernen einer neuen Programmiersprache

Namen des jeweiligen Buttons, sondern mit „Click“ auch den Namen des Ereignisses, das sie abdecken, in ihrem Funktionsnamen. Auf diese Weise versteht Delphi, um welches Ereignis es sich handelt und auf welches Element der Benutzeroberfläche es sich bezieht. Den Event Handlern wird mit dem Argument Sender auch der jeweilige Button mitgegeben, der angeklickt wurde, was dann interessant wird, wenn man denselben Event Handler für mehrere Oberflächenelemente verwendet, was ebenfalls möglich ist, und man innerhalb der Event-Handler-Funktion zwischen den unterschiedlichen Elementen, die das Ereignis ausgelöst haben können, unterscheiden muss.

### ?

#### 14.5 [3 min]

Worin unterscheidet sich ein Programm, das dem ereignisorientierten Programmierparadigma folgt, von vollkommen linear ablaufenden Programmen?

### ?

#### 14.6 [3 min]

Warum ist Ereignissesteuerung besonders für grafische Benutzeroberflächen gut geeignet?

## 14.8 Ihr Fahrplan zum Erlernen einer neuen Programmiersprache

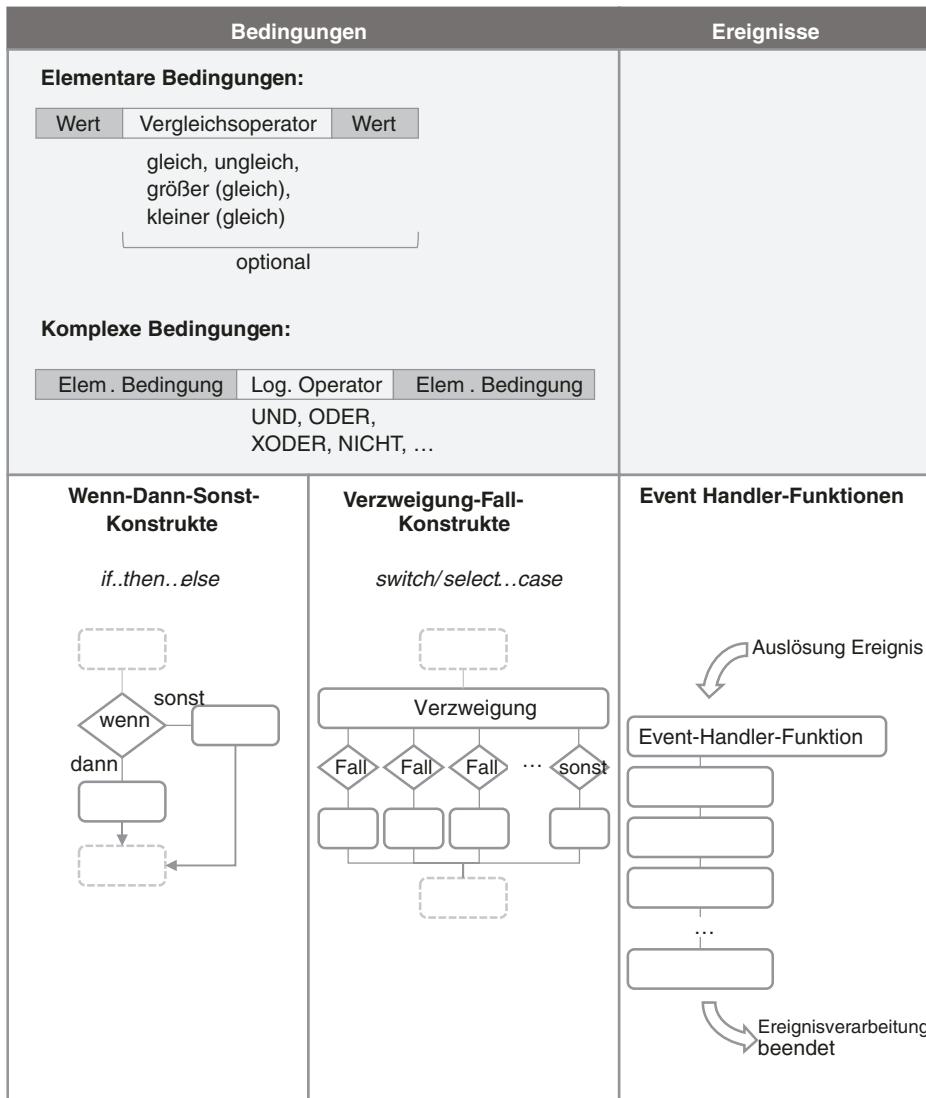
---

### ■ Wenn Sie eine neue Programmiersprache lernen...

finden Sie heraus,

- wie in der Sprache Wenn-Dann-Konstrukte formuliert werden (Schlüsselwörter, die dazu in vielen Sprachen verwendet werden, sind **if, else, then**),
- wie die Vergleichsoperatoren geschrieben werden, die dazu benutzt werden, Bedingungen zu formulieren,
- wie die logischen Operatoren geschrieben werden, die dazu benutzt werden, mehrere elementare Bedingungen zu einer Gesamtbedingung zu verknüpfen,
- ob die Sprache über ein Verzweigung-Fall-Konstrukt verfügt, und, falls ja, wie es formuliert wird (Schlüsselwörter, die dazu in vielen Sprachen verwendet werden, sind **switch, select, case**),
- wie Code-Blöcke in der Sprache abgegrenzt (also geöffnet und geschlossen) werden, und ob auf diese Begrenzer verzichtet werden kann, wenn der Code-Block nur eine einzige Anweisung enthält,
- ob die Sprache Ereignisse unterstützt, und falls ja, wie man Event Handler definiert (insbesondere auch, welche Argumente Event Handlern ggf. übergegeben werden) und wie man die Event Handler mit den Ereignissen, bei deren Auftreten sie aufgerufen werden sollen, verknüpft.

Eine Übersicht über die betrachteten Konstrukte zur Ablaufsteuerung sehen Sie in **Abb. 14.4**.



14

■ Abb. 14.4 Bedingungen und Ereignisse zur Ablaufsteuerung

## 14.9 Lösungen zu den Aufgaben

- **Aufgabe 14.1**
  - a. Ausgabe: **Ergebnis C**
  - b. Ausgabe: **Ergebnis B**
  - c. Ausgabe: **Ergebnis A, Ergebnis B**
  - d. Keine Ausgabe (keine der Bedingungen deckt den Fall ab, dass  $x$  negativ ist)

## ■ Aufgabe 14.2

```

x = eingeben("Bitte geben Sie eine Zahl ein: ")
Wenn x > 10 Dann
    Wenn x > 20 Dann anzeigen("Ergebnis A")
    Sonst Anzeigen("Ergebnis B")
Sonst
    Wenn x > 0 Dann anzeigen("Ergebnis B")
Ende Wenn

```

Innerhalb des Code-Blocks **Wenn x > 10** wird zunächst auf **x > 20** geprüft. Trifft diese Bedingung nicht zu, wissen wir, dass **x** zwar größer als 10 (sonst wäre das Programm gar nicht in diesen Code-Block eingetreten), aber kleiner oder gleich 20 sein muss. Deshalb genügt es, dass Schlüsselwort **Sonst** vor die Ausgabe von **Ergebnis B** zu setzen.

## ■ Aufgabe 14.3

- Fünf mal. Jede Bedingung wird geprüft, unabhängig davon, welches Ergebnis die Bedingungen zuvor hatten.
- Durch geschicktes Verschachteln der Wenn-Dann-Konstrukte kann die Zahl der Prüfungen von Bedingungen reduziert werden. Ist **x** tatsächlich kleiner als 0, wird nur einzige Bedingung geprüft. Die übrigen Bedingungen werden dann gar nicht mehr erreicht, da sie im **Sonst**-Zweig zur Bedingung **x < 0** liegen. Nur, wenn **x** größer oder gleich 0 und zugleich kleiner oder gleich 10 ist, durchläuft auch diese Formulierung des Programmausschnitts alle Bedingungen.

Die Kunst besteht hier in der Anordnung der Bedingungen dergestalt, dass jede Bedingung zumindest theoretisch erreicht werden kann. Hätten wir als oberste/äußerste Bedingung beispielsweise **x > 10** verwendet, würden Bedingungen wie **x > 50** gar nicht mehr geprüft werden, weil bereits **x > 10** als zutreffend bewertet wurde, und alles, was im **Sonst**-Zweig zu **x > 10** Bedingung liegt, nicht mehr durchlaufen wird.

Natürlich käme hier auch eine Formulierung mit einem Verzweige-Falls-Konstrukt (► Abschn. 14.6) als ungleich elegantere und besser lesbare Lösung in Frage.

```

x = eingeben("Bitte geben Sie eine Zahl ein: ")
Wenn x < 0 Dann ausgeben("x kleiner 0!")
Sonst
    Wenn x > 100 Dann ausgeben("x größer 100!")
    Sonst
        Wenn x > 50 Dann ausgeben("x größer 50!")
        Sonst
            Wenn x > 10 Dann ausgeben("x größer 10!")
            Sonst
                Wenn x >= 0 Und x <= 10 Dann

```

```

    ausgeben("x zwischen 0 und 10!")
Ende Wenn
Ende Wenn
Ende Wenn
Ende Wenn
Ende Wenn

```

#### ■ Aufgabe 14.4

Das Problem dieses verschachtelten Wenn-Dann-Konstrukts besteht darin, dass die Bedingung, die im inneren Wenn-Dann-Konstrukt **x** daraufhin prüft, ob es einen Wert größer oder gleich 110 hat, niemals zu einem positiven Ergebnis kommen kann. Wenn nämlich **x** tatsächlich größer als 110 ist, trifft bereits die äußere Bedingung **x > 100** zu, deren zugehöriger Code-Block (hier bestehend aus nur einer einzigen Anweisung) dann ausgeführt. Der **Sonst**-Block dagegen wird nur durchlaufen, wenn **x** kleiner oder gleich 100 ist, dann aber kann die Bedingung des inneren Wenn-Dann-Konstrukts (**x >= 110**) niemals zutreffen. Die Ausgabeanweisung für **x größer 110!** ist gewissermaßen isoliert.

#### ■ Aufgabe 14.5

Anders als ein linear ablaufendes Programm kann ein Programm, das dem ereignisorientierten Programmier-Paradigma folgt, auf Ereignisse reagieren (zum Beispiel den Klick des Benutzers auf einen Button), indem es an eine Stelle springt, die genau für dieses Ereignis bzw. diese Art von Ereignis entwickelt worden ist. Liegt gerade kein Ereignis vor, das verarbeitet werden müsste, beobachtet das Programm aufmerksam seine Umwelt und wartet darauf, wieder aktiv werden zu müssen. Während ein linear ablaufendes Programm eine lange Folge von Anweisungen ist, die Schritt für Schritt von Anfang bis Ende durchlaufen werden, besteht ein ereignisorientiertes Programm im Wesentlichen aus einer Sammlung von Event Handlern, also Funktionen, die immer dann aktiviert werden, wenn das Ereignis, das die jeweilige Funktion verarbeitet, aufgetreten ist. Die Event Handler selbst sind natürlich wieder Folgen von Anweisungen, doch statt eine lineare Ablaufsteuerung zu vollziehen, springt das Programm gewissermaßen zwischen den Event Handlern hin und her, je nachdem, welches Ereignis es gerade zu verarbeiten gilt.

#### ■ Aufgabe 14.6

Grafische Benutzeroberflächen lassen dem Benutzer meist viel Freiheit, zu entscheiden, welche Funktionen er aufrufen will. Der Benutzer wird oft nicht eng geführt, sondern wählt aus einer Palette von Möglichkeiten aus, die sich regelmäßig in Menüs, Symbolleisten, Schaltflächen, Registerkarten und anderen Steuerelementen widerspiegelt. Hier bietet sich eine ereignisorientierte Steuerung an, die einfach jenen Event Handler aufruft, der mit dem Steuerelement verbunden ist, das der Benutzer ausgelöst/aktiviert hat.



# Wie wiederhole ich Programmieranweisungen effizient?

## Inhaltsverzeichnis

- 15.1 Schleifen und ihre Erscheinungsformen – 196
- 15.2 Abgezählte Schleifen – 198
- 15.3 Bedingte Schleifen – 204
- 15.4 Ihr Fahrplan zum Erlernen einer neuen Programmiersprache – 208
- 15.5 Lösungen zu den Aufgaben – 208

## Übersicht

Oft müssen wir in Programmen eine oder mehrere gleichartige Anweisungen wiederholen. Dazu könnten wir die Anweisungen einfach mehrere Male hintereinander schreiben. Das allerdings ist nicht nur mühsam und fehleranfällig, sondern stellt uns auch immer dann vor ein Problem, wenn zu dem Zeitpunkt, als wir das Programm schreiben, noch gar nicht klar ist, wie oft genau die Anweisungen hintereinander ausgeführt werden sollen.

Deshalb kennen alle modernen Programmiersprachen sogenannte *Schleifen*, mit deren Hilfe sich die Wiederholung von Programmanweisungen elegant umsetzen lässt.

In diesem Kapitel werden Sie folgendes lernen:

- was Schleifen genau sind, und wie sie sich von Funktionen unterscheiden, mit denen ja ebenfalls Programmcode wiederholt ausgeführt werden kann
- was der Unterschied zwischen abgezählten und bedingten Schleifenkonstrukten ist
- wie Sie eine abgezählte Schleife entwickeln, und welche Rolle die sogenannte „Laufvariable“ dabei spielt
- wie Sie eine bedingte Schleife programmieren, und worin der Unterschied zwischen der Bedingungsprüfung zu Beginn (Kopfsteuerung) und am Ende der Schleife (Fußsteuerung) besteht
- wie Sie Schleifen vorzeitig beenden oder mit dem nächsten Schleifendurchlauf fortsetzen können.

## 15.1 Schleifen und ihre Erscheinungsformen

---

### ■ Schleifen versus Funktionen

Im Wesentlichen gibt es zwei Arten, Programmcode, den Sie einmal geschrieben haben, zu wiederholen: Sie können ihn zum einen in wiederaufrufbare Päckchen auslagern, in Form von Funktionen/Prozeduren. Wie das funktioniert, haben wir uns bereits in ► Kap. 12 angesehen. Zum anderen können Sie den zu wiederholenden Code in sogenannte *Schleifen* verpacken. Genau darum geht es in diesem Kapitel.

Der Unterschied zwischen beiden Ansätzen besteht darin, dass bei Schleifen ein- und derselbe Programmcode *direkt mehrfach hintereinander* durchlaufen und ausgeführt wird. Auf diese Weise könnten Sie zum Beispiel eine Produktpreisliste Zeile für Zeile, also Produkt für Produkt, durchgehen und den Preis aller Produkte, die zu einer bestimmten Kategorie gehören, um 10 % erhöhen.

Anders gelagert dagegen ist der Anwendungsfall bei Funktionen: Hier geht es nicht notwendigerweise darum, einen bestimmten Programmteil *unmittelbar hintereinander mehrfach* zu wiederholen, sondern, den Programmcode so weit unabhängig zu machen, dass man ihn jederzeit von *unterschiedlichen Stellen des Programms* aus aufrufen kann.

Beiden Ansätzen der Wiederholung gemein ist indes, dass man auf diese Weise vermeidet, denselben Programmcode mehrfach ins Programm aufzunehmen. Das erhöht die Wartbarkeit des Programms, denn eine Änderung an dem zu wiederholenden Teil, muss dann eben nur noch einmal vorgenommen werden. Würden Sie dagegen den Programmcode immer dort, wo Sie ihn aufrufen möchten (anstelle einer Funktion) bzw. mehrfach hintereinander (anstelle einer Schleife) duplizieren, müsste Sie eine Änderung an diesem Code auch in der Kopie dieses Programmteils nachziehen. Das ist mühselig und fehleranfällig. Ein weiteres Problem im Fall der Schleifen besteht darin, dass man zu der Zeit, wo man das Programm schreibt, in der Regel noch gar nicht weiß, wie oft genau der Code eigentlich wiederholt werden soll. Denken Sie an die Produktliste, die sie durchgehen wollen, um den Preis aller Produkte einer bestimmten Kategorie um 10 % zu erhöhen. Sie können ja, während Sie das Programm schreiben, noch gar nicht wissen, wie viele Produkte die Liste später einmal enthalten soll! Was passiert, wenn plötzlich zu den bestehenden 78 ein 79. Produkt hinzukommt, weil das Sortiment erweitert wird? Sollen Sie dann den Programmcode der den Preis erhöht, ein 79. Mal in Ihr Programm kopieren? Abhilfe schaffen hier die Schleifen.

### ■ Arten von Schleifen

Schleifen kommen grundsätzlich in zwei Varianten daher: Als *abgezählte* und als *bedingte* Schleifen. Bei den abgezählten Schleifen ist von vorneherein klar – zumindest prinzipiell – wie oft sie durchlaufen werden. Solche Schleifen bieten sich für unser Problem mit der Produktliste an: Wir dürfen leicht feststellen können, wie viele Produkte auf dieser Liste stehen. Dann kennen wir aber auch die Zahl der Schleifendurchläufe, die nötig sind, um die Produktliste einmal komplett zu durchlaufen. Nun stellen Sie sich aber folgendes Szenario vor: Wir haben ein einfaches Programm, das Temperaturangaben, die der Benutzer in Kelvin eingibt, in Grad Celsius umrechnet. Temperaturen in Kelvin sind immer größer oder gleich 0. Null Kelvin ist der absolute Nullpunkt, ohne jegliche Wärme, kälter kann es niemals werden; dieser absolute Nullpunkt entspricht  $-273,15^{\circ}\text{C}$ . Der Benutzer soll jetzt also der Reihe nach Temperaturen in Kelvin eingeben. Jedes Mal, wenn er eine Kelvin-Temperatur eingibt, wird diese in Grad Celsius umgerechnet. Gibt der Benutzer irgendwann eine negative Kelvin-Temperatur ein, soll das Programm abbrechen. Auch hier bietet sich natürlich eine Schleife an, ist der Umrechnungsvorgang von Kelvin in Grad Celsius doch stets der gleiche. In diesem Fall allerdings wissen wir a priori noch nicht, wie viele Schleifendurchläufe es wohl geben wird. Das hängt eben von der Benutzereingabe ab. Die Schleife der Wahl ist hier eine *bedingte* Schleife, eine Schleife, die solange läuft, wie eine bestimmte Bedingung erfüllt ist; in unserem Beispiel hier wäre die Bedingung, dass die vom Benutzer eingegebene Temperatur größer oder gleich 0 ist. Sobald der Benutzer eine negative Temperatur eingäbe, wäre die Bedingung nicht mehr erfüllt und die Schleife würde ein weiteres Mal durchlaufen.

## 15.2 Abgezählte Schleifen

### ■ Schleifen mit numerischer Laufvariable

Abgezählte Schleifen sind also unmittelbar aufeinanderfolgende Wiederholungen von Programmcode, wobei die Zahl der Wiederholungen im Vorhinein bestimmt werden kann.

Aber sind die Wiederholungen des Programmcodes wirklich absolut identisch? Denken wir noch einmal zurück an das Beispiel der Produktpreisliste, die wir durchlaufen wollen, um alle Produkte einer bestimmten Kategorie um 10 % teurer zu machen. Wenn wir nun ein Produkt der Kategorie gefunden haben und bereits das vorangegangene Produkt der relevanten Kategorie angehörte, ist der Code, der in beiden Fällen ausgeführt wird, dann derselbe? Nein, offenbar nicht; denn die Preiserhöhung muss ja einmal für das eine, einmal für das andere Produkte vorgenommen werden. Der Programmcode, so ähnlich er in beiden Fällen auch ist, muss sich also jedes Mal auf ein anderes Produkt „beziehen“. Anders formuliert: Wir müssen wissen, wo genau in unserer Schleife wir aktuell sind, welches Produkt also gerade durchlaufen wird.

Genau das erlaubt uns die sogenannte *Laufvariable*. Sie wird bei jedem Durchlauf der Schleife um einen bestimmten Wert (meist eins) hochgezählt. Damit weiß man stets in welchem Schleifendurchlauf man sich aktuell befindet. Und diese Variable können wir im Programmcode, der durch unsere Schleife wiederholt wird, benutzen, zum Beispiel um die einzelnen Elemente eines Feldes anzusprechen, das unsere Produktpreisliste repräsentiert.

Schauen wir uns das Ganze einmal als Pseudo-Code etwas genauer an:

```
Fuer p von 1 bis laenge(produkte)
Beginn
    Wenn produkte[p].kategorie = "Gartenmöbel" Dann
        produkte[p].preis = produkte[p].preis * 1.1
    Ende
```

15

In diesem Beispiel gehen wir davon aus, dass **produkte** ein Feld von Instanzen (also Objekten) der Klasse **Produkt** ist, die die Attribute **kategorie** und **preis** besitzen (blättern Sie noch einmal zu ▶ Abschn. 11.7 zurück, wenn Ihnen die Konzepte „Felder“, sowie „Klassen“ und „Instanzen/Objekte“ aus der objektorientierten Programmierung nicht mehr geläufig sind).

Die Laufvariable ist in diesem Beispiel die (Ganzzahl-)Variable **p**. Sie läuft, wie die Zeile **Fuer p von 1 bis laenge(produkte)** anzeigen, vom Wert **1** bis zum Funktionswert **laenge(produkte)**, von dem wir annehmen wollen, dass er die Länge des Feldes **produkte**, also die Anzahl der enthaltenen Produkte darstellt. Bei jedem Schleifendurchlauf wird **p** automatisch um eins erhöht.

Der zu wiederholende Programmcode steht zwischen den Schlüsselwörtern **Beginn** und **Ende** in einem Code-Block. Code-Blöcke haben wir bereits im Zusammenhang mit Funktionen (▶ Abschn. 13.1) und Wenn-Dann-Konstrukten (▶ Abschn. 14.3) gesehen.

## 15.2 · Abgezählte Schleifen

Der Code, der bei uns zwischen **Beginn** und **Ende** steht, wird so lange ausgeführt, wie die Laufvariable **p** kleiner oder gleich dem angegebenen Maximalwert ist, in unserem Fall **laenge(produkte)**.

Die Laufvariable nutzen wir nun im Inneren der Schleife bei **produkte[p]** als Index, um das jeweilige Feldelement, das im aktuellen Schleifendurchlauf gerade bearbeitet werden soll, aus dem Feld **produkte** herauszupicken. Hier sehen Sie, dass wir zwar immer den gleichen Code wiederholen, der Code aber jedes Mal etwas anderes macht, denn bei jedem Schleifendurchlauf wird mit einem anderen Produktobjekt, nämlich dem jeweiligen Element **produkte[p]** gearbeitet, beim ersten Schleifendurchlauf also mit **produkte[1]**, beim zweiten mit **produkte[2]** und beim letzten mit **produkte[laenge(produkte)]**.

Das Durchlaufen einer abgezählten Schleife ist schematisch in Abb. 15.1 dargestellt.

### ■ Schleifen mit Objekt-Laufvariable

Manche Programmiersprachen bieten eine spezielle Variante der abgezählten Schleife an, die Aufgabenstellungen wie in unserem Beispiel oben noch einfacher macht: Dabei wird nicht eine Laufvariable hochgezählt, die wir dann *als Index* für

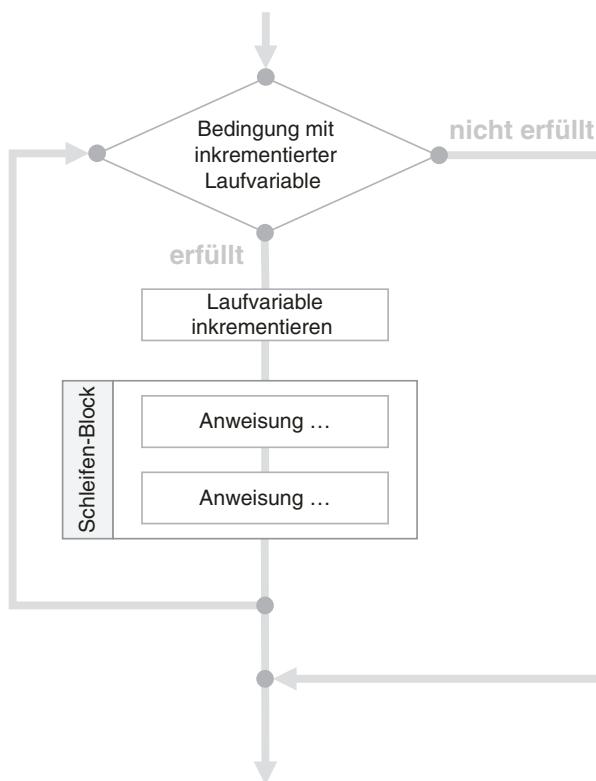


Abb. 15.1 Ablaufschema einer abgezählten Schleife

ein Feld von Objekten verwenden, was zugegebenermaßen etwas umständlich ist, sondern, die Schleife durchläuft selbstständig das Feld von Objekten und arbeitet der Reihe nach mit jedem einzelnen Objekt im Feld.

Das könnte dann so aussehen:

```
Fuer jedes p in produkte
Beginn
    Wenn p.kategorie = "Gartenmöbel" Dann
        p.preis = p.preis * 1.1
    Ende
```

Diese Schleife, die in unserem Pseudo-Code mit den Schlüsselwörtern **Fuer jedes** eingeleitet wird, durchläuft einfach der Reihe nach alle Elemente des Felds **produkte**. Das jeweils aktuelle Element des Feldes wird in einer Laufvariablen **p** abgelegt. Beachten Sie den Unterschied zu der abgezählten Schleife im Beispiel weiter oben: Die Laufvariable **p** ist hier keine Zahl, die angibt, zum wievielen Male die Schleife aktuell durchläuft, sondern das aktuelle Element des Feldes **produkte**, das gerade „seinen“ Schleifendurchlauf hat, das also gerade „an der Reihe“ ist.

Deshalb kann dann im Rumpf der Schleife, also in dem durch die Schleife wiederholten Code-Block mit diesem aktuellen Objekt **p** auch wie mit einem Produkt-Objekt gearbeitet werden, zum Beispiel also seine Attribute angepasst werden. Wichtig zu verstehen ist an dieser Stelle, dass in den meisten Programmiersprachen unsere Variable **p** nicht einfach *eine Kopie* des jeweiligen Elements des Feldes **produkte** ist, sondern letztlich *das Element selbst*. Klingt abstrakt, macht aber einen entscheidenden Unterschied aus: Wäre **p** nämlich einfach nur eine Kopie des jeweiligen Elements von **produkte**, das gerade „dran“ ist, dann hätten Änderungen, die wir an **p** vornehmen, natürlich keinerlei Auswirkungen auf das echte Element in **produkte**; schließlich arbeiten wir ja nur mit einer Kopie, das Original bliebe von unseren Änderungen unberührt. So ist es aber nicht. Tatsächlich *ist p* das jeweilige Element von **produkte**. Änderungen, die an **p** vorgenommen werden, ändern also unmittelbar das jeweilige Produkt in unserem Feld **produkte**.

Wie Sie sehen, ist diese Schleife etwas eleganter also die Schleife weiter oben. Der einzige Nachteil hier ist: Ohne, dass wir mit Hilfe einer Extra-Variablen, die wir bei jedem Schleifendurchlauf manuell erhöhen, mitzählen, wissen wir jetzt nicht, der wievierte Durchlauf der Schleife aktuell stattfindet. Unsere Laufvariable ist eben kein Zähler mehr. Wenn es aber einfach nur darum geht, die Produkte in unserem Feld **produkte** der Reihe nach zu durchlaufen, muss das natürlich kein Nachteil sein.

#### ■ Beispiele in verschiedenen Programmiersprachen

Schauen wir uns das ganze einmal in zwei konkreten Programmiersprachen an, in PHP, das für serverseitige Programme verwendet wird und auf den meisten Web-sites zum Einsatz kommt und in der Microsoft-Office-Makrosprache VBA.

Hier zunächst die beiden Schleifenvarianten in PHP:

## 15.2 · Abgezählte Schleifen

```

for ($p=0; $p <= count($produkte)-1; $p++) {
    if ($produkte[$p]->kategorie == "Gartenmöbel")
        $produkte[$p]->preis = $produkte[$p]->preis * 1.1
}

foreach ($produkte as &$p) {
    if ($p->kategorie == "Gartenmöbel")
        $p->preis = $p->preis * 1.1
}

```

Die abgezählte Schleife mit numerischer Laufvariable wird hier durch das Schlüsselwort **for** eingeleitet, die Schleife, die durch das Feld iteriert, mit **foreach**. Das sind auch tatsächlich die in den meisten Programmiersprachen wendeten Schlüsselwörtern für diese Schleifentypen.

Die **for**-Schleife enthält in Klammer drei Angaben:

1. Wie die Laufvariable heißt (Variablen-Namen werden in PHP immer mit einem vorangestellten Dollarzeichen gekennzeichnet) und bei welchem Wert sie starten soll (Felder beginnen in PHP standardmäßig beim Index 0, das erste Element wäre als **produkte[0]**).
2. Wie lange sie laufen soll; in unserem Fall also so lange, wie ihr Wert kleiner oder gleich der Anzahl der Feldelemente (**count(\$produkte)**) minus 1 ist; „minus 1“ deshalb, weil die Indizierung ja bei 0 startet. Wenn das erste Feldelement den Index 0 hat, dann hat das letzte Feldelement den Index **count(\$produkte)-1** (also bei 10 Feldelementen den **Index 9**).
3. Wie sie hochgezählt werden soll; wir sind bislang davon ausgegangen, dass die Laufvariable bei jedem Schleifendurchlauf um 1 erhöht wird. Das muss aber nicht zwingend so sein. Wir könnten zum Beispiel auch nur jedes zweite Produkt anschauen; dann würde der letzte Teil der **for**-Anweisung **\$p = \$p + 2** lauten (**\$p++** ist lediglich eine Kurzschreibweise für **\$p = \$p + 1**).

Eine Besonderheit bei PHP besteht noch darin, dass auf die Attribute eines Objekts mit Hilfe des Pfeil-Operators (**->**) zugegriffen wird. Wir hatten in unserem Pseudo-Code dafür bislang stets den Punkt verwendet.

Ein weiteres Spezifikum finden wir in der zweiten Schleifenvariante, der **foreach**-Schleife: Hier wird das aktuell durch die Schleife bearbeitete Element des Feldes **\$p** genannt. Dem vorangestellt ist aber noch ein kaufmännisches Und-Zeichen. Dieses bewirkt, dass die Variable **\$p**, wie wir es oben in unserem Pseudo-Code besprochen haben, auch tatsächlich das entsprechende Produkt-Objekt darstellt und nicht lediglich eine Kopie des aktuell bearbeiteten Produkt-Objekts auf dem **produkte**-Feld. Würden wir das auf das kaufmännische Und verzichten, würde die Zuweisung **\$p->preis = \$p->preis \* 1.1** lediglich die Kopie des Objekts bearbeiten und nicht das Objekt selbst, das Teil unseres Felds **produkte** ist.

Jetzt noch dasselbe in VBA:

```

For p = 1 To length(produkte) Step 1
    If produkte(p).kategorie = "Gartenmöbel" Then
        produkte(p).preis = produkte(p).preis * 1.1
    Next

Rem ACHTUNG: Diese Schleife führt nicht zum
Rem gewünschten Ergebnis!
For Each p In produkte
    If p.kategorie = "Gartenmöbel" Then
        p.preis = p.preis * 1.1
    Next

```

Wie Sie sehen, ist die Syntax der **For**-Schleife (und auch der Feld- und Objekt-Zugriffe sowie der Wenn-Dann-Bedingungen) in VBA ein wenig anders aufgebaut als in PHP, die Grundkonzepte sind aber vollkommen **identisch**. Einen wichtigen Unterschied gibt es jedoch bei der **For-Each**-Schleife: Anders als in PHP ist die Laufvariable **p** in VBA nämlich immer eine *Kopie* des jeweiligen Elements aus unserem Feld **produkte**. Es gibt keine Möglichkeit, die Laufvariable so zu erzeugen, dass sich Veränderungen an ihr im Originalelement unseres **produkte**-Felds wieder-spiegeln. Wollen wir das Feld **produkte** selbst verändern, müssen wir die erste Variante der abgezählten Schleife mit einer numerischen Laufvariable verwenden.

### ■ Verschachtelte Schleifen

Schleifen können auch ineinander verschachtelt werden. Stellen Sie sich vor, Sie hätten ein zweidimensionales Feld **bestand**, dessen Zeilen die unterschiedlichen Varianten ein- und desselben Produkts und dessen Spalten die unterschiedlichen Lager eines Unternehmens darstellen. Die Werte im Array repräsentieren die jeweils vorhandene Stückzahl. Gilt also zum Beispiel, dass **bestand[7, 3] = 65**, dann bedeutet das, dass von der 7. Produktvariante im 3. Lager aktuell 65 Stück verfügbar sind. Wollen Sie nun zählen, wie viele Exemplare des Produkts (egal, welche Variante) insgesamt, das heißt, über alle Lager, verfügbar sind, eignet sich dafür eine verschachtelte, abgezählte Schleife:

```

exemplare = 0
Für v von 1 bis anzahl_varianten
    Beginn
        Für g von 1 bis anzahl_lager
            Beginn
                exemplare = exemplare + bestand[v, g]
            Ende
        Ende
    Ende

```

Die äußere Schleife durchläuft die Zeilen des zweidimensionalen Feldes, also die Produktvarianten, die innere Schleife die Lager. Das bedeutet, die Schleife ar-

## 15.2 · Abgezählte Schleifen

beitet sich vor, indem sie eine neue Zeile beginnt, dann alle Spalten für diese Zeile durchgeht (innere Schleife) und dann in die nächste Zeile wechselt (äußere Schleife). Auf diese Weise wird das gesamte Feld einmal „abgefahren“. Seine einzelnen Elemente, **bestand[v, g]**, also die Anzahl der Produktvarianten im jeweiligen Lager, werden dabei mit Hilfe der Variablen **exemplare** aufsummiert, indem zum aktuellen Stand dieser Variable das gerade durch die Schleifen bearbeitete Feld **bestand[v, g]** hinzugefügt wird; nichts anderes tut die Zuweisung **exemplare = exemplare + bestand[v, g]**: Sie addiert den jeweiligen Feldinhalt zum aktuellen Wert von **exemplare** und weist das Ergebnis wiederum der Variable **exemplare** zu. Nachdem die verschachtelten Schleifen durchgelaufen sind, steht in der Variablen **exemplare** die Gesamtzahl aller Produktvarianten über alle Lager.

### ■ Schleifen vorzeitig verlassen

Die meisten Programmiersprachen bieten eine Möglichkeit, eine abgezählte Schleife vorzeitig zu verlassen. Das kann man nützlich sein, wenn man zum Beispiel einen Code-Teil schreibt, der prüfen soll, ob sich unter den Produkten auch ein Produkt einer speziellen Kategorie, sagen wir „Gartenmöbel“, befindet. Eine Möglichkeit, das zu prüfen, besteht darin, dass gesamte Feld zu durchlaufen und zu prüfen, ob das aktuelle Feld-Element ein Produkt dieser Kategorie ist. Sobald ein solches Produkt gefunden wurde, ist die Frage, ob es überhaupt Produkte der Kategorie „Gartenmöbel“ gibt, bereits beantwortet. Der Rest des Feldes muss eigentlich nicht mehr durchlaufen werden, das kostet nur unnötige Rechenzeit und ändert am Ergebnis natürlich nichts mehr. Also wäre es angebracht, die Schleife an dieser Stelle abzubrechen. In unserem Pseudo-Code könnte das so aussehen:

```
gefunden = Falsch
Für jedes p in produkte
Beginn
    Wenn p.kategorie = "Gartenmöbel" Dann
        Beginn
            gefunden = Wahr
            Verlassen
        Ende
    Ende
```

Anhand der Variable **gefunden** können wir nach dem Schleifendurchlauf feststellen, ob tatsächlich ein Produkt der Kategorie „Gartenmöbel“ gefunden worden ist. Wird bei einem Schleifendurchlauf nämlich ein solches Produkt gefunden, so wird die Variable **gefunden**, die wir zunächst mit dem Wert **Falsch** initialisiert haben, auf **Wahr** gesetzt und die Schleife sofort verlassen.

In ähnlicher Weise kennen die meisten Programmiersprachen eine Anweisung, die dafür sorgt, dass der aktuelle Schleifendurchlauf beendet und die Schleife einfach mit dem nächsten Durchlauf fortgesetzt wird.

## 15.3 Bedingte Schleifen

### ■ Funktionsweise und Arten von bedingten Schleifen

Bedingte Schleifen unterscheiden sich von abgezählten Schleifen dadurch, dass im Vorhinein noch nicht feststeht, wie oft die Schleife durchlaufen werden wird. Stattdessen hängt die Durchführung davon, ab, ob eine bestimmte Bedingung, die *Laufbedingung*, erfüllt ist. Solange das der Fall ist, läuft die Schleife. Ist die Laufbedingung irgendwann nicht mehr erfüllt, bricht die Schleife ab und die Programmausführung wird hinter der Schleife fortgesetzt.

Nehmen wir das Beispiel aus der Einleitung zu diesem Kapitel: Hier wird der Benutzer so lange nach einer Temperatur in Kelvin gefragt und diese Temperatur dann in Grad Celsius umgerechnet, bis er eine negative Kelvin-Temperatur eingibt. Dann weiß das Programm, dass es abbrechen soll, denn negative Temperaturen in Kelvin sind nach der Definition der Kelvin-Skala physikalisch unmöglich. Die Laufbedingung für unsere bedingte Schleife lautet also, dass der Benutzer eine positive Kelvin-Temperatur eingibt. Wenn er das tut, rechnet das Programm die Temperatur in Grad Celsius um und fragt ihn nach der nächsten Kelvin-Temperatur.

In unserem Pseudo-Code könnte diese Schleife dann so aussehen:

```
kelvin = eingeben("Bitte Temperatur in Kelvin eingeben:")
Solang kelvin >= 0
    Beginn
        anzeigen(kelvin, " Grad Kelvin sind:",
                 kelvin - 273.15, " Grad Celsius")
        kelvin = eingeben("Bitte Temperatur in Kelvin
                           eingeben:")
    Ende
```

15

Wir beginnen also damit, bereits vor der eigentlichen Schleife eine Temperatur in Kelvin einzulesen. Das ist notwendig, damit die Laufbedingung der Schleife sinnvoll geprüft werden kann. Die Laufbedingung folgt auf das Schlüsselwort **Solang**. Daran schließt sich der Code-Block an, der ausgeführt wird, wenn die Laufbedingung, **kelvin >= 0**, erfüllt ist. Konkret wird also die eingegebene Temperatur in Kelvin umgerechnet und das Ergebnis angezeigt. Sodann wird eine neue Kelvin-Temperatur vom Benutzer eingelesen. Da die Schleife danach auf das Schlüsselwort **Ende** stößt, springt sie wieder an den Anfang, und das bedeutet, in die Prüfung, ob die Laufbedingung jetzt, da der Benutzer eine neue Kelvin-Temperatur eingegeben hat, noch immer erfüllt ist.

In unserem Beispiel wird die Bedingung am Anfang der Schleife geprüft. Die Prüfung kann aber auch am Ende erfolgen. Dann hätten wir eine sogenannte *fußgesteuerte* Schleife, im Gegensatz zur obigen *kopfgesteuerten*.

Das sähe dann so aus:

```

Mache
Beginn
    kelvin = eingeben("Bitte Temperatur in Kelvin eingeben: ")
    Wenn kelvin >=0 Dann
        anzeigen(kelvin, " Grad Kelvin sind: ",
        kelvin - 273.15, " Grad Celsius")
    Ende
Solange kelvin >=0

```

Da die Bedingung hier erst am Ende geprüft wird, wird die Schleife also auf jeden Fall einmal durchlaufen, bis sie zur Prüfung der Bedingung gelangt. Diese Konstruktion ist in unserem Beispiel etwas aufwendig, denn wir müssen innerhalb der Schleife ja trotzdem prüfen, ob die Eingabe des Benutzers eine zulässige Kelvin-Temperatur (also größer oder gleich 0) ist, oder darauf hindeutet, dass der Benutzer die Schleife abbrechen will. Die Umrechnung in Grad Celsius findet in unserer fußgesteuerten Schleife nur dann statt, wenn die Kelvin-Temperatur tatsächlich größer oder gleich 0 ist. Andernfalls passiert im Inneren der Schleife nach der Eingabe überhaupt nichts. Die Schleife läuft dann auf die Prüfung der Laufbedingung, stellt fest, dass diese nicht mehr erfüllt ist und geht nicht in einen weiteren Schleifendurchlauf, sondern setzt das Programm hinter der Schleife fort.

In Abb. 15.2 und 15.3 sehen Sie jeweils eine schematische Darstellung des Ablaufs einer kopf- sowie einer fußgesteuerten Schleife.

Die Schleife in unserem Beispiel sieht insgesamt etwas sehr bemüht aus, man würde normalerweise in diesem Fall sicherlich eher mit einer kopfgesteuerten Schleife arbeiten.

#### ■ Beispiele in verschiedenen Programmiersprachen

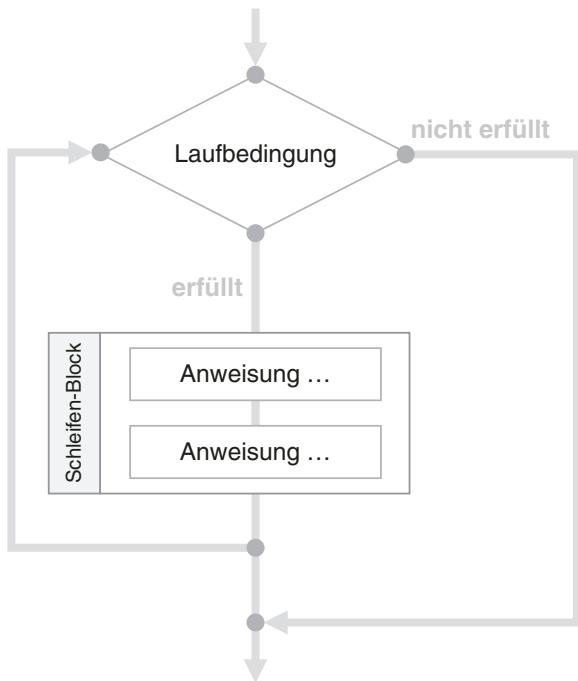
Hier zwei Umsetzungen unsers Kelvin-Celsius-Umrechnungsprogramms in realen Programmiersprachen, einmal in Pascal, einmal in VBA. Zunächst die Pascal-Version:

```

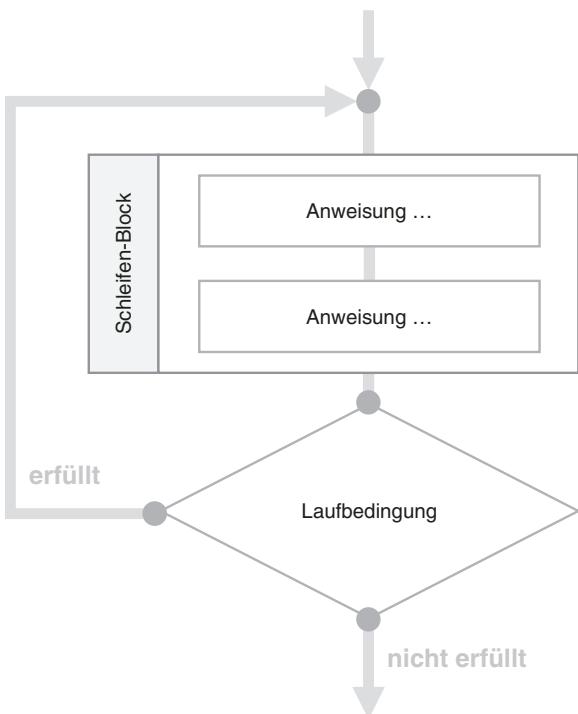
program DateiSchreiben;
var
    kelvin: real;
begin
    kelvin := readln('Bitte Temperatur in Kelvin eingeben: ');
    while kelvin >= 0 do
        begin
            writeln(kelvin, ' Grad Kelvin sind: ', kelvin - 273.15,
            ' Grad Celsius');
            kelvin := readln('Bitte Temperatur in Kelvin eingeben: ');
        end
    end.

```

■ Abb. 15.2 Ablaufschema einer kopfgesteuerten bedingten Schleife



■ Abb. 15.3 Ablaufschema einer fußgesteuerten bedingten Schleife



Jetzt das Ganze in VBA:

```
Dim kelvin As Double;

kelvin = InputBox("Bitte Temperatur in Kelvin eingeben: ")
While kelvin >=0
    MsgBox (Str(kelvin) & " Grad Kelvin sind: " & Str(kelvin - 273.15) &
        " Grad Celsius")
    kelvin = InputBox("Bitte Temperatur in Kelvin eingeben: ")
Wend
```

Wie Sie sehen, heißen die Schlüsselwörter für die bedingten Schleifen in Pascal und VBA **while...do** und **while**, wie auch in den meisten anderen Programmiersprachen. Der zu wiederholende Code-Block wird in Pascal mit **begin** und **end** eingefasst, in VBA beginnt er direkt nach der Laufbedingung und endet mit dem Schlüsselwort **wend**.

Genau wie im Fall der abgezählten Schleifen, besitzen die meisten Programmiersprachen auch spezielle Anweisungen, um eine bedingte Schleife vollständig zu verlassen (häufig **break**), oder, um den aktuellen Durchlauf abzubrechen und mit dem nächsten Durchlauf fortzufahren (häufig **continue**).

#### ■ Das Verhältnis von abgezählten und bedingten Schleifen

Abgezählte Schleifen können auch als bedingte Schleifen geschrieben werden, nicht aber umgekehrt. Alles, was wir dafür tun müssen, ist, eine Bedingung zu formulieren, die so lange erfüllt ist, wie die abgezählte Schleife laufen würden. Im Beispiel des vorangegangenen Abschnitts, indem wir für den Preis aller Produkte der Kategorie „Gartenmöbel“ um 10 % erhöhen wollen, sähe das dann in Pseudo-Code so aus:

```
p = 1
Solang p <= laenge(produkte)
    Beginn
        Wenn produkte[p].kategorie = "Gartenmöbel" Dann
            produkte[p].preis = produkte[p].preis * 1.1
            p = p + 1
    Ende
```

Auch hier arbeiten wir, ähnlich wie bei der abgezählten Schleife, mit einer Laufvariable. Nur müssen wir uns dieses Mal um das Hochzählen dieser Variable bei jedem Schleifendurchlauf selbst kümmern; das tun wir mit der Anweisung **p = p + 1**. Die Laufbedingung der Schleife lautet nun, dass der Wert dieser Laufvariable höchstens so groß ist, wie die Länge des Felds **produkte**. Auch das Initialisieren der Laufvariable, was uns die abgezählte Schleife ebenfalls abgenommen hatte, müssen wir hier vor dem ersten Schleifendurchlauf selbst übernehmen. Sie sehen aber, dass es durchaus möglich ist, abgezählte Schleifen in bedingte Schleifen „umzubauen“,

weil letztlich das abgezählte Wiederholen im Grunde auch auf einer Bedingung basiert. Umgekehrt ist dies natürlich nicht (immer) möglich, denn wir wissen im Fall der bedingten Schleife ja überhaupt nicht, wie oft sie wohl wiederholt werden wird, was aber gerade die Voraussetzung für eine abgezählte Schleife ist.

```
x = 1
Solange x <> 100
    Beginn
        anzeigen((x+1)/2, ". Schleifendurchlauf")
        x = x + 2
    Ende
```

### ? 15.2 [3 min]

Welche Arten von Schleifen gibt es und worin unterscheiden sie sich?

### ? 15.3 [3 min]

Warum kann man mit einer bedingten Schleife eine abgezählte Schleife nachbauen, umgekehrt aber nicht notwendigerweise?

### ? ! 15.4 [5 min]

Überlegen Sie, wie man mit einer Schleife die Benutzereingaben auf einer grafischen Benutzeroberfläche verarbeiten könnte.

## 15.4 Ihr Fahrplan zum Erlernen einer neuen Programmiersprache

### ■ Wenn Sie eine neue Programmiersprache lernen ...

finden Sie heraus,

- welche Formen von abgezählten Schleifen es gibt und welche Rolle die Laufvariable in ihnen einnimmt („Zählvariable“ oder aktuelles Element einer Menge von Objekten, die durchlaufen wird)
- welche Formen von bedingten Schleifen es gibt, insbesondere, ob es nur kopfgesteuerte oder auch fußgesteuerte Schleifen gibt,
- welche Möglichkeiten es gibt, Schleifen vorzeitig zu verlassen oder zumindest den aktuellen Schleifendurchlauf abzubrechen und die Schleife mit dem nächsten Durchlauf fortzusetzen.

## 15.5 Lösungen zu den Aufgaben

### ■ Aufgabe 15.1

Das Problem an dieser Schleife ist, dass sie niemals enden wird, es handelt sich also um eine Endlosschleife. Die Variable x, deren Wert vor jedem Schleifendurchlauf

geprüft wird, wird zunächst, vor der Schleife, mit dem Wert 1 initialisiert und dann bei jedem Schleifendurchlauf um 2 erhöht. Ihr Wert ist also stets ungerade. Daher wird die Variable niemals den Wert 100 annehmen, der zum Abbruch der Schleifen führen würde. Die Schleife läuft unendlich weiter (oder eben so lange, bis der Speicher nicht mehr reicht, um die hohen Werte der Variable `x` aufzunehmen).

### ■ Aufgabe 15.2

Es gibt abgezählte und bedingte Schleifen. Bei *abgezählten* Schleifen steht im Prinzip bereits vor dem ersten Schleifendurchlauf fest, wie oft die Schleife durchlaufen werden wird. Diese Schleifen arbeiten mit einer Laufvariable. Diese Laufvariable ist entweder ein numerischer Wert ist, der ausgehend von einem Startwert bei jedem Schleifendurchlauf solange nach einer festen Regel verändert (zum Beispiel um eins erhöht) wird, bis er einen festgelegten Endwert erreicht. Oder aber die Schleife durchläuft eine Menge von Objekten und die Laufvariable repräsentiert bei jedem Durchlauf ein anderes Objekt aus dieser Menge. Auf diese Weise lassen sich bestimmte, abgrenzbare Mengen von Objekten (etwa Kunden, Produkte, Verkaufstransaktionen) auf einfache Weise durchlaufen. Innerhalb der Schleife kann dann mit dem jeweiligen Objekt, das gerade „an der Reihe ist“, das durch die Laufvariable repräsentiert wird, gearbeitet werden.

Anders als abgezählte Schleifen richtet sich das Durchlaufen einer *bedingten* Schleife danach, ob eine Bedingung, die vor (kopfgesteuerte Schleifen) oder nach jedem Durchlauf (fußgesteuerte Schleifen) geprüft wird, erfüllt ist. Diese Bedingung kann auch von Größen abhängen, die sich während der Schleifendurchläufe überhaupt erst ergeben, etwas berechnete Werte oder Benutzereingaben. Deshalb kann bei bedingten Schleifen nicht zwingend bereits vor dem ersten Schleifendurchlauf gesagt werden, wie oft die Schleife insgesamt durchlaufen werden wird.

### ■ Aufgabe 15.3

Eine abgezählte Schleife ist letztlich ein Spezialfall der bedingten Schleife. Die Bedingung ist nämlich, dass der Wert Laufvariable sich in einem bestimmten Wertebereich (zwischen Startwert und Endwert) bewegt oder – wenn die Menge von Objekten durchlaufen wird, und die Laufvariable das „aktuelle“ Objekt repräsentiert – dass noch Objekte in der durchlaufenden Objektmenge übrig sind, die bisher noch nicht „an der Reihe“ waren. Weil also letztlich auch hier eine Laufbedingung zum Tragen kommt, kann eine abgezählte Schleife mit ihrer Laufbedingung auch als bedingte Schleife formuliert werden. Umgekehrt geht dies allerdings nicht, weil bei den bedingten Schleifen ja im Vorhinein noch gar nicht klar sein muss, wie oft eigentlich die Schleife durchlaufen wird, was aber eine Voraussetzung für eine abgezählte Schleife ist. Denken Sie an eine bedingte Schleife, deren Durchlaufen von einer Benutzereingabe abhängt. Da nicht absehbar ist, in welchem Schleifendurchlauf der Benutzer die entscheidende Eingabe machen wird, die zum Abbruch der Schleife führt, können wir, bevor das tatsächlich geschehen ist, nicht sagen, wie oft die Schleifen durchlaufen wird.

Natürlich gibt es Fälle, bei denen auch bei bedingten Schleifen im Vorhinein die Zahl der Durchläufe bereits feststehen kann, dann nämlich, wenn sich während der

Schleifendurchläufe nichts mehr an jenen Größen ändern kann, die in der Laufbedingung darüber entscheiden, ob die Schleife ein weiteres Mal durchlaufen wird, oder nicht. Dann ist auch bei einer bedingten Schleife von Anfang klar, wie oft sie durchlaufen wird, und sie könnte auch als abgezählte Schleife geschrieben werden.

Die besonders Spitzfindigen werden jetzt anmerken, dass man aber durchaus auch mit abgezählten Schleifen zum Beispiel den obigen Fall mit der Benutzereingabe abbilden könnte, etwa dadurch, dass man eine abgezählte Schleife baut, deren Endwert nie erreicht wird, die also ewig läuft, wenn wir nicht künstlich den Wert der Laufvariablen auf den Endwert setzen, der den letzten Schleifendurchlauf markiert. Innerhalb der Schleife würde dann mit einer Wenn-Dann-Bedingung prüfen, ob die Schleife abgebrochen werden soll. Ist das der Fall, setzt man einfach den Wert der Laufvariablen auf den Endwert und bei der nächsten Prüfung, ob die Laufvariable noch im „laufenden“ Bereich liegt, würde die Schleife dann abbrechen. Das könnte in unserem Pseudo-Code dann so aussehen:

```

ziel = 5
Für x von 1 bis ziel
Beginn
    ziel = ziel + 1
    Wenn Eingeben("Bitte machen Sie Ihre Eingabe") = "Ende"
        Dann x = ziel
Ende

```

Der Trick ist hier, dass wir den Endwert der Laufvariable bei jedem Schleifendurchlauf um eins erhöhen, sodass die Laufvariable ihn nicht erreichen kann.

Gibt der Benutzer in einem Schleifendurchlauf „Ende“ ein, wird **x** auf den Endwert gesetzt. Vor dem nächsten Durchlauf wird die Laufvariable um eins erhöht und es wird geprüft, ob sie noch im Bereich von **1** bis **ziel** ist. Da das dann nicht mehr der Fall ist, wird die Schleife kein weiteres Mal durchlaufen.

Es ist natürlich klar, dass diese Art, die Laufvariable bzw. ihren Endwert innerhalb der Schleife zu manipulieren, nicht gerade die feine Art ist. Eine solche mühsam zurechtgebogene abgezählte Schleife ist erheblich schwerer zu verstehen, als Ihr Pendant in Form einer echten, saubereren bedingten Schleife. Deshalb würde man sich in einem solchen Fall des letztgenannten Schleifentyps bedienen.

## 15

### ■ Aufgabe 15.4

Eine grafische Benutzeroberfläche zeichnet sich ja im Normalfall dadurch aus, dass der Benutzer aus unterschiedlichen Aktionen wählen kann. Diese lösen dann jeweils ein Ereignis aus, auf das das Programm mit einer entsprechenden Ereignisbehandlungsroutine reagieren kann. Dieser Ablauf ließe sich aber auch als Schleife darstellen. Dazu würden wir in einer Unendlich-Schleife vom Benutzer immer wieder aufs Neue eine Aktion einlesen und diese Aktion dann innerhalb der Schleife verarbeiten. Das könnte stilisiert so aussehen:

```
abbruch = Falsch
Solange abbruch = Falsch
    BeginnS
        aktion = Einlesen()
        Verzweigung aktion
            Fall    ...: ...
            Fall    ...: ...
            // Behandlung aller möglichen Fälle
            Fall    "Beenden": abbruch = Wahr
    Ende
Ende
```

Wenn Sie mit dem Verzweigung-Fall-Konstrukt nicht mehr ganz vertraut sind, blättern Sie einfach nochmal einige Seiten zurück zu ► Abschn. 14.6.

### ?

#### 15.1 [3 min]

Was ist das Problem mit der folgenden Schleife?



# Wie suche und behebe ich Fehler auf strukturierte Art und Weise?

## Inhaltsverzeichnis

- 16.1 Fehler zur Entwicklungszeit – 214
- 16.2 Fehler zur Laufzeit – 215
- 16.3 Testen – 217
- 16.4 Debugging-Methoden – 218
- 16.5 Ihr Fahrplan zum  
Erlernen einer neuen  
Programmiersprache – 220

## Übersicht

Fehler sind der ärgste Feind des Programmierers. So sorgfältig man auch arbeitet, Fehler schleichen sich immer ein. Das Testen des Programms, also das Aufspüren und die Beseitigung von Fehlern sind deshalb essentielle Bestandteile des Programmierens.

Fehler äußern sich entweder dadurch, dass das Programm gar nicht erst ausgeführt werden kann, dass es bei der Ausführung abstürzt, oder dadurch, dass es, selbst wenn es bis zum Ende durchläuft, trotzdem nicht das tut, was es soll.

Unterschieden werden können dabei Fehler, die bereits *zur Entwicklungszeit* geschehen, also während das Programm geschrieben wird, und solche, die erst *zur Laufzeit* entstehen, dann also, wenn das Programm ausgeführt wird.

Zum Glück gibt es einige Ansätze und Werkzeuge, die beim Verstehen von Fehlerursachen, dem sogenannten *Debugging*, helfen können.

In diesem Kapitel werden Sie lernen:

- welche Arten von Fehlern es gibt und wodurch sie verursacht werden
- wie man Laufzeitfehler durch die Behandlung sogenannter Ausnahmen abfangen und unschädlich machen kann
- wie man beim Testen von Programmen geschickt vorgeht
- was ein Debugger ist
- was die Debugger-Features „Haltepunkte“, „Einzelschrittmodus“ und „Variablenbeobachtung“ sind, und wie man sie einsetzt, um Ort und Ursache von Fehlern zu verstehen
- wie man vorübergehende, zusätzliche Ausgaben dazu nutzen kann, um Fehler im Programm auch ohne Debugger zu diagnostizieren.

### 16.1 Fehler zur Entwicklungszeit

Fehler, die bereits zur Entwicklungszeit geschehen, sind entweder Fehler in der Syntax (die „Grammatik“ des Programms stimmt also nicht, eine oder mehrere Anweisungen entsprechen nicht den „Satzbauregeln“) oder, selbst, wenn die Syntax korrekt ist, inhaltlich-algorithmische Fehler; Fehler also, die darin liegen, dass der Programmcode – obwohl formal korrekt – einfach nicht die exakte Umsetzung dessen ist, was der Programmierer eigentlich tun und erreichen möchte.

16

Bei *Syntaxfehlern* hilft regelmäßig der Compiler bzw. Interpreter, der das Komplizieren oder Ausführen des Programms mit einer Fehlermeldung abbricht. Diese Fehlermeldung gibt normalerweise Auskunft darüber, welche Art von Fehler aufgetreten ist und an welcher Stelle im Code (oft über die Zeilennummer oder den Namen des Programmteils) die Ursache vermutlich liegt. Ein solcher Fehler könnte etwa darin bestehen, dass eine Variable nicht deklariert worden ist, obwohl das in der verwendeten Programmiersprache notwendig ist. Verwenden Sie dann die Variable im Programm, wird der Compiler oder Interpreter an dieser Stelle den Dienst versagen. Manchmal sind die Fehlermeldungen allerdings etwas kryptisch, und nur mit dem inneren Aufbau der Programmiersprache sehr gut vertrautet Pro-

grammierer verstehen, was der Compiler/Interpreter hier mitteilen will. In solchen Fällen hilft meist eine Suche im Internet, denn auch andere Programmierer werden sich über diese Fehlermeldung schon gewundert haben. Für sehr viele Sprachen wird man auf *StackOverflow* fündig werden.

*Inhaltliche/algorithemische* Fehler sind oft schwieriger aufzuspüren und zu verstehen. Hierzu verwenden Sie die Debugging-Methoden, die in ► Abschn. 16.4 beschrieben werden.

## 16.2 Fehler zur Laufzeit

---

### ■ Nicht vorhergesehene Umstände während der Programmausführung

Der Unterschied zwischen Fehlern, die bereits während der Entwicklungszeit des Programms entstehen und Fehlern, die erst zur Laufzeit auftreten ist, dass bei letzteren das Programm im Prinzip funktioniert, aber während der Ausführung Umstände auftreten, die der Entwickler nicht vorhergesehen hat und für die er dementsprechend keine Vorkehrungen getroffen hat.

Ein solcher Umstand könnte zum Beispiel darin bestehen, dass das Programm versucht, auf eine Datei zuzugreifen, die nicht existiert. Dieses Problem ist während der Entwicklung nie aufgetreten, die Dateien, auf die das Programm zugreifen wollte, waren stets auch tatsächlich vorhanden. Weil das so ist, ist es dem Entwickler nie in den Sinn gekommen, eine Vorkehrung für diesen Fall zu treffen. Ein ähnliches, in der Praxis sehr häufig anzutreffendes Beispiel ist eine Situation, in der der Benutzer eine Eingabe macht, mit der das Programm nicht klarkommt. Beispielsweise gibt der Benutzer einen Text ein, wo eigentlich eine Zahl erwartet wird. Das Programm, das ansonsten vollkommen einwandfrei funktioniert, versucht nun mit der vermeintlichen Zahl, die in Wahrheit ein Text ist, zu rechnen und stürzt ab. Beim Testen hat der Programmierer natürlich stets eine Zahl eingegeben, wenn das Programm ihn dazu aufforderte. Dementsprechend kam er nie in die Situation, in der das Programm abstürzte und deshalb auch nie auf die Idee, eine entsprechende Vorkehrung zu treffen, zum Beispiel, indem er die Eingabe des Benutzers daraufhin prüft, ob es sich wirklich um eine Zahl handelt und die Eingabe zurückweist, sollte das nicht der Fall sein.

Fehler, die erst zur Laufzeit entstehen, sind also regelmäßig darauf zurückzuführen, dass irgendeine Situation, die in der Realität auftreten kann, während der Entwicklung nicht vorhergesehen worden ist. Unter idealen Bedingungen dagegen läuft das Programm ohne Beanstandungen.

Die Kunst besteht nun also darin, während der Entwicklung bereits möglichst viele denkbare Fehler abzufangen. So könnte das Programm beispielsweise testen, ob eine Benutzereingabe, mit der im Anschluss gerechnet werden soll, tatsächlich eine Zahl ist, und sie anderenfalls zurückweisen. Ebenso kann geprüft werden, ob numerische Variablen, durch die dividiert wird, jemals den Wert 0 annehmen können, was natürlich sofort einen Fehler verursacht. Auch die Existenz von Dateien, auf die das Programm zugreifen möchte, kann überprüft und der Programmablauf angepasst werden, sollte die Datei nicht vorhanden sein.

Dieses Antizipieren möglicher Fehler ist zwar aufwendig, aber wichtig, vor allem dann, wenn man Programme schreibt, die andere benutzen sollen; die können

unter Umständen nicht einfach in den Programmcode schauen, haben vielleicht kein technisches Verständnis und eine geringe Akzeptanz für Fehler, insbesondere dann, wenn sie für das Programm bezahlt haben.

#### ■ Abfangen von Fehlern mit Konstrukten des Typs **Versuche ... Bei Fehler**

Etliche Programmiersprachen unterstützen die Fehlerbehandlung im Programm durch spezielle Sprach-Konstruktionen. Die Grundidee dabei ist, dass Fehler sich in sogenannten *Ausnahmen* (engl. *exceptions*) wiederspiegeln. Ausnahmen sind nichts weiter als Ereignisse, die ausgelöst werden, wenn ein Fehler einer bestimmten Art auftritt. Programmierer sprechen in diesem Zusammenhang auch davon, die Ausnahmen würden „geworfen“ (engl. *throw an exception*).

Der Analogie des „Werfens“ entsprechend können diese Ausnahmen dann „aufgefangen“ und behandelt werden. Eine Ausnahme könnte zum Beispiel beim Öffnen einer Datei auftreten, wenn die Datei, aus der gelesen werden soll, nicht vorhanden ist. Das Programm bricht dann aber nicht ab, sondern fährt kontrolliert mit einem Programmteil fort, der für eben genau diesen Fall vorgesehen ist.

In unserem Pseudo-Code würde eine solche Konstruktion so aussehen:

```
Versuche
// Datei öffnen und Daten lesen

Bei Fehler
// Fehlermeldung anzeigen

EndeVersuche
```

Man „versucht“ also, einen bestimmten Programmteil auszuführen; wenn das funktioniert, geht es hinter **EndeVersuche** weiter. Wenn es aber zu einer Ausnahme kommt, wird zunächst der Code hinter dem Schlüsselwort **Bei Fehler** ausgeführt. Manche Programmiersprachen erlauben es, innerhalb einer solchen Konstruktion unterschiedliche Ausnahmen aufzufangen und jeweils unterschiedlich auf jede dieser Ausnahmen zu reagieren. Oftmals wird dabei ein spezielles *Exception*-Objekt bereitgestellt, aus dessen Eigenschaften man weitere Details über die aufgetretene Ausnahme herauslesen kann, um seine Reaktion darauf abzustimmen.

Schlüsselwörter, die in Programmiersprachen im Zusammenhang mit solchen Konstruktionen häufig zu finden sind, sind **Try** (für unser **Versuche**), **On Error** oder **Catch** (beides für das **Bei Fehler** in unserem Pseudo-Code). Im Schlüsselwort **Catch** erkennt man sehr schön die Idee des „Auffangens“ einer „geworfenen“ Ausnahme.

Was ist nun der Unterschied zwischen einer solchen **Versuche ... Bei Fehler**-Konstruktion und einem Ansatz, der ohne diese Konstruktion auskommt, indem er einfach prüft, ob die Datei vorhanden ist, bevor er sie zu öffnen versucht? Beim letztgenannten Vorgehen wird bereits im Vorfeld die mögliche Fehlerquelle abgeprüft und die Aktion, die dann zu einem Fehler führen könnte, gegebenenfalls gar nicht mehr ausgeführt. Bei der **Versuche ... Bei Fehler**-Konstruktion findet vor der

Aktion keinerlei Prüfung statt, stattdessen läuft man sozusagen direkt in den Fehler hinein. Nur wirkt sich der Fehler eben nicht so aus, dass das Programm abstürzt oder sich sonstwie unkontrolliert verhält, sondern, der Fehler wird, *nachdem* er aufgetreten ist (nachdem also die Ausnahme „geworfen“ worden ist), vollkommen kontrolliert bearbeitet. Der Vorteil dieses Vorgehens besteht darin, dass man a priori noch gar nicht ganz genau wissen muss, was alles schiefgehen kann. Vielleicht wird ja die Datei, die gelesen werden soll, trotzdem nicht lesbar sein, obwohl sie existiert, zum Beispiel, weil der Benutzer unzureichende Zugriffsrechte besitzt. Auch dieser Fehler würde mit unserer **Versuche ... Bei Fehler**-Konstruktion abgefangen werden, ohne, dass wir das explizit vorsehen müssten. Die Gefahr dieses scheinbar so praktischen Vorgehens ist natürlich, dass man sich über die möglichen Fehlerquellen und die optimale Reaktion auf die daraus resultierenden Fehler weniger Gedanken macht.

Besser ist deshalb ein Ansatz, bei dem man eine **Versuche ... Bei Fehler**-Konstruktion verwendet, aber darin möglichst viele Ausnahmen separat abfängt, individuell auf sie reagiert, und dann schließlich noch für alle Ausnahmen, die man nicht separat behandelt, eine allgemeine Fehlerbehandlungsroutine in die Konstruktion einbaut. Dieses Vorgehen setzt natürlich voraus, dass die Programmiersprache die separate Behandlung verschiedener Arten von Ausnahmen tatsächlich unterstützt, was allerdings sehr viele Sprachen tun.

### 16.3 Testen

Angesichts der vielfältigen möglichen Fehlerquellen ist natürlich *Testen* der Schlüssel zum Erfolg. Testen, Testen und noch einmal Testen. Das Testen ist beinahe eine Wissenschaft für sich, es gibt zahlreiche unterschiedliche Ansätze und Arten von Tests. Tatsächlich ist Tester sogar ein Berufsbild in der Softwareentwicklung. In manchen Firmen arbeiten Entwickler und Tester pärchenweise zusammen. Der Bedeutung des Testens entsprechend gibt es spezielle Werkzeuge, um Tests zu entwickeln, teilweise sogar automatisch durchzuführen, die Ergebnisse zu dokumentieren und die Beseitigung der Fehler projektmanagementtechnisch zu begleiten.

So aufwendig müssen Sie es sich natürlich nicht machen. Trotzdem ist Testen wichtig. In der Box finden Sie einige Tipps zum besseren Testen.

#### Tipp

- Überlegen Sie sich, wie ein Nutzer Ihr Programm verwenden würde, und spielen Sie diese „Use cases“ durch.
- Überlegen Sie, was alles schiefgehen könnte, das heißt, wie der Benutzer das Programm möglicherweise „unsachgemäß“ verwenden könnte.
- Setzen Sie das Programm „extremen“ Bedingungen aus. Während der Entwicklung neigt man dazu immer die gleichen einfachen Beispiele, mit denen das Programm einwandfrei funktioniert, zu verwenden, weil es natürlich das ist,

was man will: wenige Probleme während der Entwicklung. Arbeiten Sie dem entgegen und seien Sie Ihr eigener *Advocatus diaboli*.

- Testen Sie abschnittsweise *und* im Zusammenhang. Code-Passagen (zum Beispiel eine Funktion) einzeln zu testen, macht die Fehlersuche leichter, weil man im Falle eines Fehlers ja bereits weiß, wo die Ursache ungefähr liegen muss. Trotzdem ergeben sich manchmal unvorhergesehene Wechselwirkungen, wenn man das Programm einmal komplett laufen und die unterschiedlichen Code-Passagen ineinander greifen lässt. Deshalb sollte man stets beiden Ansätzen folgen, dem abschnittsweisen Testen und dem Testen im vollem Programmzusammenhang.

## 16.4 Debugging-Methoden

Wenn Sie feststellen, dass irgendetwas an Ihrem Programm nicht funktioniert, werden Sie sich auf die Suche nach dem Fehler begeben. Die Aktivitäten rund um die Suche und Behebung von Fehlern wird auch als *Debugging* bezeichnet.

Hier gibt es grundsätzlich zwei Ansätze: Entweder Sie arbeiten mit „Bordmitteln“, insbesondere mit geeigneten Hilfsausgaben im Programmablauf, oder Sie verwenden spezielle Werkzeuge, sogenannte *Debugger*. Letztere sind häufig Bestandteil der Integrierten Entwicklungsumgebungen der Programmiersprachen und bequem zu benutzen. Allerdings gibt es auch Kommandozeilen-Debugger.

Im Grunde geht es beim Debugging um zwei Probleme:

- Festzustellen, *wo* der Fehler passiert
- Festzustellen, *warum* der Fehler passiert

Dabei ist es keineswegs immer so, dass Sie stets zuerst herausbekommen, wo der Fehler geschieht und dann die Frage nach dem Warum beantworten können. Manchmal erkennen Sie zunächst, warum der Fehler auftritt und müssen dann nach der Code-Stelle forschen, die ihn verursacht.

Zur Beantwortung beider Fragen haben Sie im Wesentlichen folgende Möglichkeiten zur Verfügung:

### ■ Ausgaben

Wenn Sie mit „Bordmitteln“ arbeiten wollen, können Sie vorübergehend *zusätzliche Ausgaben* in Ihr Programm einbauen, die später, wenn das Debugging abgeschlossen ist, wieder entfernt werden und die der eigentliche Benutzer Ihres Programms niemals zu Gesicht bekommen soll. Diese Ausgaben können Ihnen helfen zu verstehen, welche Stellen des Programms bereits durchlaufen worden sind. Wenn Sie sich zum Beispiel mit dem Problem eines Programmabsturzes konfrontiert sehen, und Sie nicht genau wissen, an welcher Stelle Ihr Programm eigentlich abstützt, schalten zu kurzzeitig einige Ausgaben dazwischen. Jedes Mal, wenn eine solche Ausgabe erscheint, wissen Sie, dass das Programm zumindest bis zu dieser Stelle einwandfrei durchgelaufen ist, ohne abzubrechen. Manchmal ist auch nicht ganz klar, ob zum Beispiel bestimmte Bedingungen (Wenn-Dann-Konstrukte)

oder Schleifen überhaupt durchlaufen werden. Auch hier hilft es zur „Positionsbestimmung“ eine entsprechende Ausgabe einzubauen, die nur dann durchlaufen wird, wenn der Programmcode im Wenn-Dann-Konstrukt oder der Schleife auch tatsächlich ausgeführt wird.

Vorübergehende, zusätzliche Ausgaben können Sie aber nicht nur dazu verwenden, um festzustellen, welcher Teil Ihres Programmcodes gerade ausgeführt wird, sondern auch, um zu überprüfen, welchen Wert bestimmte Variablen im Programmablauf haben. Das Verständnis des Inhalts von Variablen hilft häufig bei der Diagnose des Problems.

### ■ Haltepunkte

Wenn Sie mit einem Debugger arbeiten, können Sie sogenannte *Haltepunkte* (engl. *breakpoints*) setzen; starten Sie das Programm dann, läuft es nur bis zu diesem Haltepunkt. Danach können Sie entscheiden, ob Sie es weiterlaufen lassen oder aber an dieser Stelle beenden wollen. Auch hier gilt: Gehen Sie einem Programmabsturz nach und kommt das Programm ohne Schwierigkeiten zum Haltepunkt, dann liegt das Problem *hinter* dem Haltepunkt.

### ■ Einzelschrittmodus

Eine weitere Möglichkeit, die Ihnen regelmäßig zur Verfügung steht, wenn Sie mit einem Debugger arbeiten, ist die Ausführung von Programmen im *Einzelschrittmodus*. Das bedeutet, es wird immer nur eine Programmanweisung ausgeführt. Erst, wenn Sie eine bestimmte Taste/Tastenkombination drücken, wird die nächste Programmanweisung ausgeführt. Die IDE zeigt Ihnen grafisch im Programmcode an, welche Anweisung die zuletzt ausgeführte ist. Auch damit ist es einfach möglich, festzustellen, an welcher Stelle ein Programm abbricht. Wenn Sie allerdings Schleifen im Programm haben, die oft wiederholt werden, ist es recht mühselig, diese im Einzelschrittmodus zu durchlaufen. In diesem Fall sollten Sie einen Haltepunkt hinter der Schleife setzen, das Programm bis dorthin durchlaufen lassen und von da an dann im Einzelschrittmodus weitergehen. Zumindest, wenn der Fehler nicht in der Schleife selbst liegt, sparen Sie sich damit eine Menge Tastendrücke (liegt das Problem allerdings doch an der Schleife selbst, kommen Sie nicht umhin, in Einzelschritten die Schleife durchzugehen).

### ■ Variablenbeobachtung

In Verbindung mit Breakpoints oder der Ausführung Ihres Programms im Einzelschrittmodus bietet sich noch eine weitere Funktionalität an, die viele Debugger unterstützen, die *Beobachtung von Variablen* (engl. *watches*). Dieses Feature erlaubt Ihnen, den aktuellen Inhalt einer Variablen anzuschauen, manchmal sogar zu verändern. Wenn Ihr Programm also an einem Haltepunkt angekommen ist, können Sie sich anschauen, welchen Wert bestimmte Variablen derzeit haben. Im Einzelschrittmodus können Sie dadurch mitverfolgen, wie sich die Werte der Variablen von einer Programmanweisung zur nächsten verändern. □ Abb. 16.1 zeigt eine solche Variablenbeobachtung für unser simples Celsius-Kelvin-Umrechnungsbeispiel. In Zeile 60 des Programms wurde ein Haltepunkt eingerichtet. Das Programm wurde ausgeführt und ist nun bis zu dieser Zeile durchgelaufen; die nächste An-

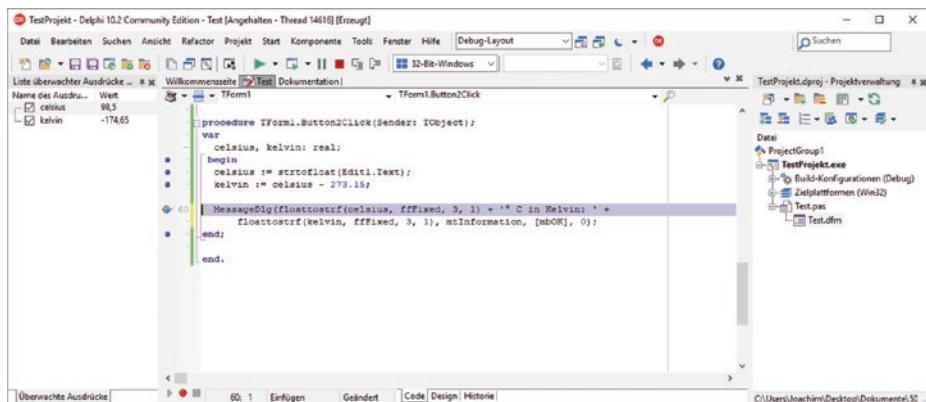


Abb. 16.1 Beispiel für Haltepunkt mit Variablenbeobachtung in Delphi

weisung, die jetzt zur Ausführung stünde, ist die in Zeile 60. Hier aber steht der Haltepunkt (markiert durch einen roten Punkt vor der Zeile; dieser wird allerdings gerade von einem blauen Pfeil überlagert, der andeutet, dass diese Zeile nun die als nächstes ausgeführte wäre). Links sehen Sie ein Fenster „Liste überwachter Ausdrücke“. Dort wurden zwei Variablen zur Beobachtung eingetragen, deren Werte an der aktuellen Stelle der Programmausführung, also unmittelbar vor der Ausführung von Zeile 60, wir hier sehen.

## 16.5 Ihr Fahrplan zum Erlernen einer neuen Programmiersprache

- Wenn Sie eine neue Programmiersprache lernen ...
  - finden Sie heraus,
  - welche Debugging-Tools Ihnen zur Verfügung stehen und wie Sie diese erreichen können (Kommandozeilen-Werkzeuge, Integration in IDE),
  - ob, und wenn ja, wie Sie Haltepunkte setzen und aufheben können,
  - ob, und wenn ja, wie Sie Ihr Programm im Einzelschrittmodus ausführen können,
  - ob, und wenn ja, wie Sie den Inhalt von Variablen überwachen können.



# Python

## Inhaltsverzeichnis

**Kapitel 17 Einführung – 223**

**Kapitel 18 Was brauche ich zum Programmieren? – 227**

**Kapitel 19 Wie bringe ich ein Programm zum Laufen? – 235**

**Kapitel 20 Wie stelle ich sicher, dass ich (und andere) mein Programm später noch verstehe? – 245**

**Kapitel 21 Wie speichere ich Daten, um mit Ihnen zu arbeiten? – 257**

**Kapitel 22 Wie lasse ich Daten ein- und ausgeben? – 301**

**Kapitel 23 Wie arbeite ich mit Programmfunctionen, um Daten zu bearbeiten und Aktionen auszulösen? – 357**

- Kapitel 24** Wie steuere ich den Programmablauf und lasse das Programm auf Benutzeraktionen und andere Ereignisse reagieren? – 385
- Kapitel 25** Wie wiederhole ich Programmanweisungen effizient? – 399
- Kapitel 26** Wie suche und behebe ich Fehler auf strukturierte Art und Weise – 417



# Einführung

## Übersicht

Jetzt geht es richtig los!

In diesem Teil des Buches starten wir mit unserer ersten Programmiersprache, nämlich Python. Wie werden uns dabei an den 9 Fragen orientieren, anhand derer wir im vorangegangenen Teil die Grundkonzepte des Programmierens kennengelernt haben.

Zunächst beginnen wir aber mit einem kompakten Überblick über die Sprache, ihre Entwicklung, Verbreitung und Verwendung.

Python wurde ab Ende 1989 durch den niederländischen Mathematiker und Informatiker *Guido van Rossum* entwickelt; angeblich aus Langeweile über die Weihnachtstage (was machen Sie so an Weihnachten?). Der Name lehnt sich an die von Van Rossum geliebte britische Komiker-Truppe *Monty Python* an, hat also eigentlich gar nichts mit der Schlange zu tun, deren stilisiertes Bild heute im Python-Logo zu sehen ist.

Aus dem Hobby-Projekt des Guido van Rossum ist längst eine der populärsten Programmiersprachen überhaupt geworden. Zwei von vielen Indikatoren dafür sind die Positionierung von Python im *TIOBE-Ranking* (2020: Platz 3, hinter Java und C) und die über 1,3 Millionen Beiträge auf *StackOverflow*, die sich Fragen rund um Python widmen. Für praktisch alle Internet- und Tech-Giganten wie Google oder Amazon gehört Python heute zu den ständig verwendeten Sprachen.

Python ist eine General-Purpose-Language und wird dementsprechend auch für die unterschiedlichsten Zwecke eingesetzt, darunter – nicht zuletzt dank Frameworks wie *Django* – zunehmend auch für Anwendungen im Web. Außerordentlich populär ist Python dank Erweiterungsbibliotheken wie *NumPy* für Data Science geworden und scheint sich de facto zur Standardsprache für Anwendungen im Bereich Künstliche Intelligenz zu entwickeln, für den mit Bibliotheken wie *TensorFlow* und *Keras* ebenfalls einige bedeutende Python-Erweiterungen zur Verfügung stehen.

Die Popularität von Python ist zweifelsohne auch darauf zurückzuführen, dass die Sprache verhältnismäßig leicht zu lernen ist. Python hat eine einfache und schnörkellose Syntax und fördert die Entwicklung eines gut verständlichen Programmecodes. Aufgrund dessen gilt Python vielen als die Einsteiger-Programmiersprache schlechthin. Zahlreiche Menschen, und Sie scheinen da keine Ausnahme zu sein, machen mit der im Rahmen von Van Rossums Hobbyprojekt entwickelten Programmiersprache heute ihre ersten Schritte hin zum Programmieren.

Ein Hobby-Projekt ist Python natürlich schon lange nicht mehr. Mit der *Python Software Foundation* gibt es seit 2001 eine gemeinnützige Organisation, die sich um die Pflege und Weiterentwicklung von Python kümmert. In ihr spielte Guido van Rossum bis vor einigen Jahren die herausragende Rolle, weshalb er gelegentlich auch als *benevolent dictator for life* bezeichnet wurde. Mittlerweile hat er sich aber in weiten Teilen aus den Aktivitäten zurückgezogen.

Noch ein Tipp, bevor wir richtig starten: Probieren Sie Dinge aus! Versuchen Sie, nicht nur die Beispiele, die das Buch und die Übungen präsentieren nachzu-

vollziehen, sondern probieren Sie auch selbst Dinge aus, abseits der Pfade die Ihnen das Buch vorgibt. Aus kaum etwas anderen werden Sie soviel lernen, wie wenn Sie sich einfach die Fragen stellen: Was passiert eigentlich, wenn ich dieses und jene leicht anders mache? Neugierde und die Bereitschaft, selbst Neues auszuprobieren, werden Ihnen das Erlernen jeder neuen Programmiersprache ungemein viel einfacher machen.

Nun aber genug der hehren Worte. Lassen Sie uns beginnen.



# Was brauche ich zum Programmieren?

## Inhaltsverzeichnis

- 18.1 Den Python-Interpreter installieren – 228
- 18.2 Die PyCharm-IDE installieren – 230
- 18.3 Hilfe zu Python erhalten – 231
- 18.4 Zusammenfassung – 233

## Übersicht

Als erstes werden wir uns damit beschäftigen, was Sie zum Programmieren mit Python alles brauchen. Das ist zum Glück gar nicht viel, so dass wir schnell mit der eigentlichen Arbeit beginnen können.

In diesem Kapitel werden Sie lernen:

- wie Sie den Python-Interpreter installieren
- wie Sie die *PyCharm*-Entwicklungsumgebung installieren
- wie Sie die wichtigsten Funktionen von *PyCharm* benutzen
- wie Sie die Hilfe von Python verwenden.

### 18.1 Den Python-Interpreter installieren

Das Installationsprogramm von Python können Sie bequem vom ► <http://www.python.org> herunterladen. Die Webseite wird von der *Python Software Foundation* betrieben, einer Organisation, die die Weiterentwicklung von Python koordiniert. Wählen auf der Webseite unter „Downloads“ einfach die aktuellste Python-Version für Ihr Betriebssystem. Wir arbeiten hier im Buch mit Python für Windows, aber alles, was wir hier besprechen werden, gilt für die MacOS- und Linux-Versionen ganz genauso. Das Python, das Sie dann installieren, ist das sogenannte *C**Python*, die – wenn man so will – „offizielle“ Python-Version, an deren Entwicklung auch der Python-Erfinder *Guido van Rossum* beteiligt ist. Daneben existieren noch zahlreiche andere Python-Implementierungen, also Interpreter, die Python-Code interpretieren und ausführen. Es gibt sogar Python-Interpreter, die selbst in Python geschrieben sind! *C**Python* dagegen ist in der Programmiersprache C entwickelt worden. Und genau dieses *C**Python* laden Sie jetzt herunter, auch wenn es auf der ► [python.org](http://python.org)-Webseite nicht unter diesem Namen geführt wird.

Wenn der Download abgeschlossen ist, starten Sie den Installer und folgen den Anweisungen auf dem Bildschirm. Merken Sie sich einmal den Pfad, in den Python installiert wird. An den Standardeinstellungen, die das Setup-Programm vorgibt, müssen Sie nichts ändern. Die Installation ist in der Regel innerhalb weniger Minuten vollständig abgeschlossen.

Übrigens gibt es *C**Python* nur in Englisch, das heißt, alle Meldungen, Warnungen und sonstigen Ausgaben werden nur in englischer Sprache erzeugt (die Programmiersprache Python als solche verwendet natürlich ebenfalls nur englische Begriffe). Insofern ist Ihnen der Dialog, in dem Sie bei der Installation die Sprache auswählen können, nicht entgangen, es gab einfach keinen! Abgesehen davon, ist es jedoch ohnehin zu empfehlen, von allen Tools jeweils die englische Version zu installieren, denn das macht die Suche nach Hilfe im Internet erheblich leichter, wenn Sie etwa eine Fehlermeldung nicht verstehen oder herausbekommen wollen, wie man einen bestimmten Arbeitsgang mit dem Tool bewerkstelligt bekommt.

Nach der Installation sind Sie eigentlich bereits startklar. Wenn Sie in das Verzeichnis wechseln, in das Python installiert worden ist (dieses trägt standardmäßig

## 18.1 · Den Python-Interpreter installieren

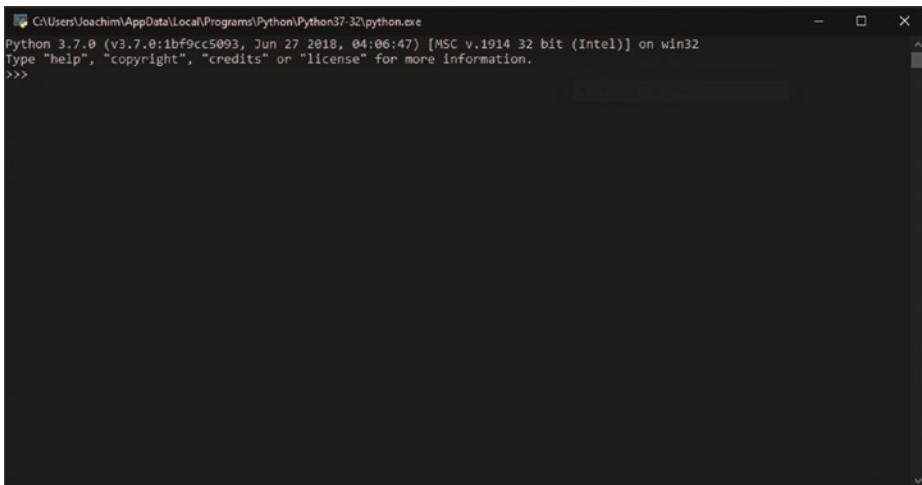


Abb. 18.1 Die Python-Konsole nach dem ersten Start

immer die Versionsnummer mit im Namen, zum Beispiel **python38-32** für Python 3.8, 32bit-Version), können Sie dort das Programm **python** aufrufen. Es öffnet sich ein Fenster, das aussieht, wie in Abb. 18.1 dargestellt.

Das ist die *Python-Konsole*. Sie erlaubt es, Python im *interaktiven Modus* zu betreiben, also eine Anweisung einzugeben und direkt auszuführen (blättern Sie nochmal einige Seiten zurück zu ► Kap. 9, wenn Ihnen der Unterschied zwischen interaktivem und Skript-Modus nicht mehr geläufig ist). Ein solcher Befehl, den die Konsole verarbeitet, ist **quit()**, er beendet die Python-Konsole und schließt das Fenster.

Neben diesem interaktiven Modus bietet Python natürlich auch einen Skript-Modus, mit dem wir längere Programme am Stück ausführen können. Python-Code-Dateien tragen üblicherweise die Endung **.py**. Hätten Sie also ein Programm **meinprogramm.py** und wollten dieses ausführen, so könnten Sie im Terminal/in der Konsole Ihres Betriebssystems in das Python-Verzeichnis wechseln und dort eingeben:

```
python meinprogramm.py
```

Bequemer als mit dem **python**-Interpreter direkt geht es natürlich mit einer integrierten Entwicklungsumgebung. Und eine solche bringt Python in der Gestalt von *IDLE* von Haus aus mit.

Die ist allerdings eher rudimentär ausgestattet, auch wenn Sie Features wie Syntax Highlighting unterstützt. Einen Eindruck von *IDLE* bekommen Sie in Abb. 18.2. Wir werden im Weiteren mit *PyCharm* arbeiten, einer ungleich mächtigeren Entwicklungsumgebung, deren Installation wir uns als nächstes zuwenden.



```

taschenrechner.py - C:\taschenrechner.py (3.7.0)
File Edit Format Run Options Window Help
from tkinter import Tk, Button, Label
from tkinter.font import Font
from functools import partial

# Eventhandler-Funktionen für Buttons

def ziffer_operator_press(ziffer_operator):
    display['text'] = display['text'] + ziffer_operator

def loeschen_press():
    display['text'] = ''

def kopieren_press():
    win.clipboard_clear()
    win.clipboard_append(display['text'])

def plusminus_press():
    display['text'] = '-' + display['text']

def gleich_press():
    display['text'] = str(eval(display['text']))

def enter_press(ereignis): # Eventhandler für Enter-Taste
    gleich_press()

# Fenster der Anwendung

win = Tk()

win['background'] = '#000000'
win.title('Taschenrechner')
win.geometry('268x470')
win.resizable(height = False, width = False)

```

Abb. 18.2 Die Python-Entwicklungsumgebung IDLE

## 18.2 Die PyCharm-IDE installieren

PyCharm ist eine populäre integrierte Entwicklungsumgebung (IDE) für Python, die von der tschechisch-russischen Softwareschmiede JetBrains mit Hauptsitz in Prag entwickelt wird. JetBrains ist auf Entwicklungswerzeuge spezialisiert und bietet auch andere bekannte IDEs wie beispielsweise *IntelliJ IDEA* (insbesondere für Java) und *WebStorm* (für Web-Entwicklung, insbesondere JavaScript) an und hat mit *Kotlin* auch selbst eine Programmiersprache entwickelt, die vor allem in der Mobile-App-Entwicklung Anwendung findet.

Von der Webseite ► <https://www.jetbrains.com/PyCharm/> kann man die kostenlose Community-Version von PyCharm herunterladen. Diese ist zwar vom

Funktionsumfang her eingeschränkt, bietet aber bereits erheblich mehr Features, als wir in diesem Buch jemals werden einsetzen können. Klicken Sie den „Download“-Button und wählen Sie dann Ihr Betriebssystem – unterstützt werden Windows, MacOS und Linux.

Nachdem Sie das Setup-Programm heruntergeladen haben, starten Sie den Installationsprozess. Die Standardeinstellungen können Sie auch hier grundsätzlich belassen; empfehlenswert ist jedoch zumindest, im Abschnitt „Create Association“ .py-Dateien mit *PyCharm* zu verbinden, sodass diese ab sofort dann immer automatisch mit der IDE geöffnet werden. Die Installation dauert in der Regel nicht lange, verlangt aber über 600 MB freie Festplattenkapazität. Die Zeiten, als man MS-DOS-Entwicklungsumgebungen von drei oder vier 3,5-Zoll-Disketten mit je 1,44 MB Speicherkapazität installieren konnte, sind definitiv vorbei!

Beim ersten Start von *PyCharm* wird gefragt, ob Sie Einstellungen importieren wollen. Wählen Sie hier die Optionen „Do not import settings“ und bestätigen Sie mit „OK“. Ebenfalls werden Sie beim ersten Start von *PyCharm* danach gefragt, ob Sie das dunkle oder das helle Farbschema verwenden möchten. Alle Abbildungen hier im Buch wurden der besseren Druckbarkeit wegen im hellen Farbschema erzeugt; der Autor arbeitet ansonsten allerdings im Dark Mode, der die Augen etwas weniger belastet. Die nun folgende Installation von Plugins können Sie bequem überspringen. Auf diese Weise gelangen Sie zu einem Willkommensbildschirm, der Sie fragt, ob Sie ein neues Projekt anlegen oder ein vorhandenes öffnen wollen. Am besten beenden Sie *PyCharm* an dieser Stelle erst mal, indem Sie den Dialog schließen. Im folgenden Kapitel werden wir unser erstes Python-Projekt beginnen. Dann geht es an dieser Stelle weiter.

## 18.3 Hilfe zu Python erhalten

---

### ■ Im Internet

Python ist eine der populärsten Programmiersprachen überhaupt. Kein Wunder also, dass Sie im Internet zahllose Ressourcen rund um Python finden. Eine dieser Ressourcen ist die bereits erwähnte Plattform *StackOverflow*, auf der Sie eine Unmenge von Fragen und Antworten zu Python finden, die kaum noch Wünsche offen lassen. Zu dem Zeitpunkt, da diese Zeilen geschrieben werden, sind auf *StackOverflow* 1.327.647 Fragen mit dem Tag **Python** versehen. Heute sind alleine 643 dazugekommen. Die Wahrscheinlichkeit, dass auch Ihre Frage unter den 1,3 Millionen ist, dürfte hoch sein. Daher ist *StackOverflow* für die meisten Fragen eine gute Anlaufstelle. Die *StackOverflow*-Treffer werden bei Google regelmäßig auch unter den Top-Suchergebnissen gelistet. Nachhelfen können Sie natürlich, indem Sie die Google-Suche mit der Zusatzangabe site: ► [StackOverflow.com](http://StackOverflow.com) auf die *StackOverflow*-Foren einschränken oder natürlich auch direkt auf ► [StackOverflow.com](http://StackOverflow.com) suchen. Aber auch über *StackOverflow* hinaus ist das Internet natürlich voll mit Beiträgen, Blogs, Tutorials, Videos und allen erdenklichen anderen Inhaltsformaten rund um das Thema Python.

### ■ „Offizielle“ Python-Hilfe

Python selbst bringt natürlich auch eine offizielle Hilfe mit. Wenn Sie also wissen wollen, wie ein bestimmtes Element der Programmiersprache arbeitet, können Sie auf die Dokumentation der Klassen und Funktionen zurückgreifen, die mit Python automatisch installiert wird.

Das tun Sie, indem Sie in der Python-Konsole die Funktion **help(element)** aufrufen und ihr als Argument den Namen des Moduls, Packages, der Klasse oder der Funktion mitgegeben, zu der Sie Informationen sehen möchten. So liefert etwa der Aufruf von **help(print)** folgenden Hilfe-Informationen:

```
>>> help(print)
print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout,
          flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file: a file-like object (stream); defaults to the current
          sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a
          newline.
    flush: whether to forcibly flush the stream.
```

Starten Sie nochmal die Konsole wie in ► Abschn. 18.1 und probieren Sie es aus! Geben Sie dabei das **>>>** nicht mit ein. Es handelt sich um das *Prompt*, die Eingabeaufforderung, die anzeigt, dass die Konsole bereit ist, Ihre Eingabe entgegenzunehmen. Wenn Sie übrigens einmal **help()** ohne Argument eingeben, wechselt die Konsole in das Python-Hilfe-System. Das veränderte Prompt **help>** zeigt an, dass Sie jetzt Begriffe eingeben können, zu denen dann passende Hilfe-Einträge angezeigt werden. In die Python-Konsole zurück gelangen Sie durch Eingabe von **quit** (ohne Klammern).

Der Output unseres **help()**-Aufrufs liefert uns einige Informationen dazu, was die Argumente der Funktion **print()** sind, und vor allem, was **print()** überhaupt macht. Diese Hilfe ist aber eher rudimentär. Wenn Sie mehr Details haben möchten ist es unumgänglich, sich selbst im Internet auf die Suche zu machen.

Auch die *Python Software Foundation* bietet auf ihrer Website ► [python.org](https://python.org), von der wir eben bereits den Python-Interpreter heruntergeladen haben, unter der Rubrik „Library Reference“ zahlreiche Informationen zur Python-Standardbibliothek (unter anderem auch zur Funktion **print()**: ► <https://docs.python.org/3/library/functions.html#print>). Die „Language Reference“ erläutert den Aufbau und die Syntax der Sprache, ist aber eher technisch gehalten. In der Rubrik „Tutorial“ findet sich eine Einführung in Python, die aber vermutlich ohne vorherige Kenntnisse anderer Programmiersprachen eher schwer verdaulich ist.

## 18.4 Zusammenfassung

---

In diesem Kapitel haben wir uns damit beschäftigt, wie Python sowie die *PyCharm*-IDE installiert werden. Darüber hinaus haben wir gesehen, welches die wichtigsten Möglichkeiten sind, Hilfe zu Python zu erhalten.

Folgende Punkte sollten Sie aus diesem Kapitel unbedingt mitnehmen:

- Die „offizielle“ Python-Implementierung, *CPython*, wird von der *Python Software Foundation* weiterentwickelt und ist diejenige Variante von Python, die man standardmäßig installiert, wenn man Python von ► [python.org](http://python.org) herunterlädt.
- Python kommt mit dem Python-Interpreter-Programm namens **python**, das wahlweise als interaktive Python-Konsole oder zur Interpretation ganzer Python-Skripte verwendet werden kann
- Mit *IDLE* bringt Python auch eine rudimentäre IDE mit.
- *PyCharm* von JetBrains ist eine sehr mächtige Python-Entwicklungsumgebung, von der eine eingeschränkte Community-Version kostenlos verfügbar ist.
- Hilfe zu Python liefert Python selbst mit der Funktion **help(element)**; sie stellt zu Python-Modulen, -Klassen oder -Funktionen Hilfetexte bereit, die aber oft nicht sehr ausführlich sind.
- Weitere „offizielle“ Hilfe erhält man vor allem in Form der Sprach-Referenz sowie der Referenz zu Python-Standardsbibliothek auf ► [python.org](http://python.org).

Daneben sind angesichts der hohen Popularität von Python zahlreiche ergiebige Informationsquellen im Internet verfügbar; eine der wichtigsten ist – wie für die meisten anderen Programmiersprachen auch – das Entwickler-Forum *StackOverflow*.



# Wie bringe ich ein Programm zum Laufen?

## Inhaltsverzeichnis

- 19.1    **Entwickeln und Ausführen von Programmen in Python – 236**
- 19.2    **Die Python-Konsole: Python im interaktiven Modus – 241**
- 19.3    ***PyCharm* kennenlernen – 242**
- 19.4    **Zusammenfassung – 242**

## Übersicht

Jetzt geht es endlich mit dem Programmieren los: Wir schreiben unser erstes kleines Python-Programm und nutzen zugleich die Gelegenheit, uns mit *PyCharm*, der mächtigen Python-IDE, die wir im letzten Kapitel installiert haben, vertraut zu machen.

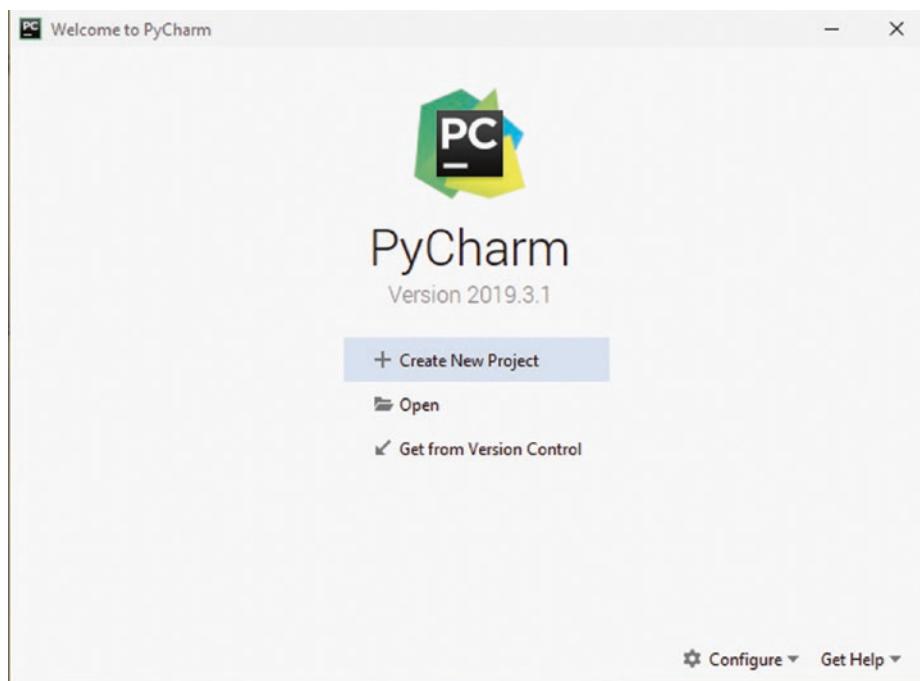
In diesem Kapitel werden Sie lernen:

- wie die Oberfläche der *PyCharm*-IDE aufgebaut ist
- wie Sie mit *PyCharm* Programme schreiben und ausführen können
- wie Sie Python mit *PyCharm* im interaktiven Modus verwenden
- wie ein einfaches „Hallo-Welt“-Programm in Python aussieht.

## 19.1 Entwickeln und Ausführen von Programmen in Python

### ■ Festlegen des Interpreters

Nach dem ersten Start von *PyCharm* erwartet Sie ein Dialog, der ungefähr so aussieht wie in □ Abb. 19.1. Da wir ein neues Programm schreiben wollen, klicken Sie auf die Schaltfläche „Create New Project“. *PyCharm* fragt Sie nun nach dem Python-Interpreter, den Sie verwenden wollen. *PyCharm* ist in der Lage, mit unterschiedlichen Python-Interpretern zu arbeiten. Das ist wichtig, wenn Sie Programme



□ Abb. 19.1 Startdialog von *PyCharm*

pflegen wollen, die nur auf einer älteren Python-Version laufen. Diese Programme für eine neuere Python-Version umzuschreiben, wäre unter Umständen sehr aufwendig. Deshalb können Sie mit *PyCharm* einfach einen älteren Interpreter verwenden, und so ihre in die Jahre gekommenen Programme ohne Probleme weiter betreiben. Die Möglichkeit, die Umgebung zu definieren, in der Ihr Programm laufen soll, geht sogar noch darüber hinaus. Nicht nur können Sie einen anderen als den aktuellsten Interpreter verwenden, Sie können bei Python (sofern Sie das wollen) auch auswählen, auf welche Python-Bibliotheken (sogenannte Module und Packages) Ihr Programm zugreifen, insbesondere, welche Version dieser Bibliotheken es verwenden soll. Auf diese Weise können Sie sich ein maßgeschneidertes Ambiente bauen, in dem Ihr Programm läuft, eine sogenannte *virtuelle Umgebung* (engl. *virtual environment*). Mit Modulen, Packages und virtuellen Umgebungen beschäftigen wir uns ausführlicher in ► Abschn. 23.3. Hier steht erst mal im Vordergrund, den Interpreter festzulegen, den wir verwenden wollen. Und da kann es aus Gründen der Kompatibilität alter Programme eben Sinn machen, mit einem älteren Interpreter zu arbeiten.

Aber keine Sorge: Normalerweise führen Versionssprünge in Python nicht zu so großen Veränderungen an der Sprache, dass zuvor geschriebene Programme plötzlich nicht mehr lauffähig sind. Allerdings hat es mit dem Übergang von Python-Version 2.X zu 3.X tatsächlich einige größere Veränderungen gegeben, die in einigen Fällen eben genau diesen unschönen Effekt haben. Deshalb ist die Fähigkeit von *PyCharm*, mit mehreren Python-Interpretern umzugehen, zunächst mal eine sehr praktische Funktionalität.

Selbst aber, wenn wir mit dem aktuellsten Python-Interpreter arbeiten wollen, müssen wir *PyCharm* mitteilen, wo dieser zu finden ist. Öffnen Sie dazu den Bereich „Project Interpreter: Existing Interpreter“ und klicken wählen Sie aus den beiden zentralen Optionen „Existing Interpreter“. Mit der anderen Option könnten Sie eine virtuelle Umgebung für Ihr Projekt erzeugen. Das müssen wir hier aber nicht. Im „Interpreter“-Auswahlfeld steht möglicherweise derzeit bei Ihnen „<No interpreter>“. Wenn das der Fall ist, klicken Sie auf den nebenstehenden Button mit den drei Punkten und wählen aus dem sich nun öffnenden Dialog die Option „System Interpreter“. Dort sollte bereits der Pfad zu der ausführbaren Datei des Python-Interpreters hinterlegt sein. Wenn das nicht so ist, können Sie jederzeit die ausführbare Datei, die praktischerweise auch selbst **python** heißt, auf Ihrer Festplatte suchen und im Dialog hier manuell auswählen. Jetzt können Sie unter „Location“ (ganz oben im „New Project“-Dialog) noch festlegen, wo genau *PyCharm* Ihr Python-Projekt speichern soll und schon sind Sie startklar für die ersten Schritte mit Python. Klicken Sie auf „Create“ und los geht es!

#### ■ Python-Programme mit *PyCharm* erstellen

Es öffnet sich die *PyCharm*-Oberfläche. Wenn bei Ihnen am oberen Bildschirm-Rand die Symbolleiste mit den Werkzeugen zu Beginn nicht angezeigt wird, klicken Sie einmal im Menü „View“ auf „Appearance“ und markieren Sie dann „Toolbar“.

Als erstes erzeugen wir eine neue Python-Skript-Datei. Klicken Sie dazu aus dem Menü „Datei“ die Option „New...“ und wählen Sie dann aus dem Dropdown-

Menü die Option „Python File“. Jetzt müssen Sie der Datei nur noch einen Namen geben, zum Beispiel **hallowelt.py** (Python-Dateien haben, wie Sie bereits wissen, im Allgemeinen die Dateiendung **.py**). Im rechten Bereich des Fensters öffnet sich nun Ihr neues Python-Skript. Und jetzt fangen wir an zu programmieren!

Geben Sie folgende Code-Zeilen in das Skript ein:

```
print("Hallo Welt")
print("Das ist mein erstes Python-Programm")
```

Sie merken sofort, dass die *PyCharm* Sie bei der Eingabe unterstützt, zum Beispiel dadurch, dass direkt eine schließende Klammer erzeugt wird, wenn Sie eine öffnende Klammer eintippen. Auch werden Klammerpärchen, die zusammengehören, hervorgehoben, wenn Sie den Textcursor vor oder hinter einer der beiden Klammern stellen. Das ist besonders dann hilfreich, wenn Sie viele ineinander verschachtelte Klammern haben und wissen wollen, welche öffnenden und schließenden Klammern eigentlich zusammengehören. Hin und wieder zeigt die IDE auch kleine Glühbirnen-Symbole an. Wenn Sie darauf klicken, erhalten Sie in der Regel Hinweis dazu, wie Sie Ihren Code noch besser formatieren können, und können die Formatierung meist auch direkt mit einem einzigen Klick umsetzen.

#### ■ Python-Programme ausführen – in PyCharm und auf der Kommandozeile

Nun wollen wir unser Programm aber auch starten. Klicken Sie dazu mit der rechten Maustaste in den freien Raum im Editor-Bereich und wählen Sie die Option „Run“ aus dem Kontextmenü (Sie können alternativ auch <CTRL><SHIFT><F10> drücken).

Jetzt tut sich etwas in *PyCharm*! Am unteren Fensterrand öffnet sich ein Bereich, der einer Registerkarte ähnelt und mit „Run“ betitelt ist. Das ist die Run-Konsole, in der Ihre Python-Programme laufen. Und wie Sie sehen, hat Python das kleine Programm, das Sie in den Editor eingegeben haben, tatsächlich ausgeführt und die zwei Textzeilen in die Run-Konsole ausgegeben. Es erscheint (hier am Beispiel von *PyCharm* auf einem Windows-System) folgender Output:

```
C:\[Pfad zu Ihrem Python-Interpreter]\python.exe C:/\[Pfad zu
ihrem Projekt]/hallowelt.py
Hallo Welt
Das ist mein erstes Python-Programm

Process finished with exit code 0
```

Wie das Ganze in der *PyCharm*-Oberfläche aussieht, sehen Sie in □ Abb. 19.2.

Statt Ihr Programm über *PyCharm* auszuführen, hätten Sie es ebenso gut über das Terminal/die Konsole Ihres Betriebssystems mit dem Befehl **python hallowelt.py** starten können (unter Umständen müssen Sie beim Python-Interpreter oder bei Ihrem Python-Skript oder bei beiden noch die Pfadangabe ergänzen, je nachdem

## 19.1 · Entwickeln und Ausführen von Programmen in Python

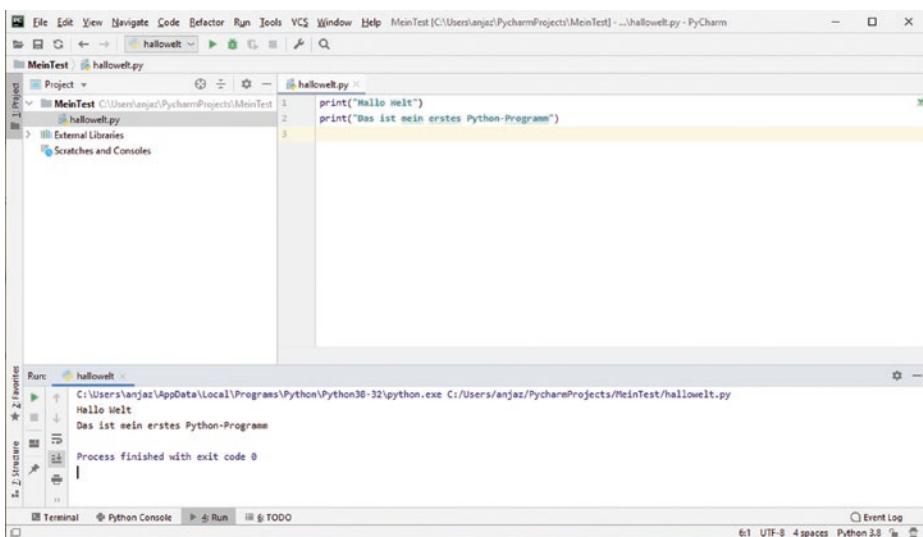


Abb. 19.2 Programm ausführung in der Run-Konsole

wo in der Verzeichnisstruktur sie sich befinden). Das können Sie auch ganz leicht ausprobieren, denn zwei Registerkarten links neben der Run-Konsole befindet sich mit der „Terminal“-Registerkarte ein direkter Zugang zur Kommandozeilenebene Ihres Betriebssystems. Keine Sorge aber, wenn Sie keine Kenntnisse der Befehle haben, die man im Terminal des Betriebssystems verwendet, dank *PyCharm* kommen Sie auch ganz ohne diese auf den ersten Blick archaisch anmutende Bedienung des Betriebssystems aus: Sie können Ihr Programm ja einfach in der Run-Konsole starten (die aber natürlich auch nichts anderes macht, als den Python-Interpreter mit Ihrem Programm aufzurufen und den Output auf der *PyCharm*-Oberfläche darzustellen). Den Befehl, dessen Eingabe *PyCharm* Ihnen dabei abnimmt, sehen Sie übrigens als erste Zeile in der Run-Konsole. Diese Zeile könnten Sie in das Terminal/ die Konsole des Betriebssystems (oder eben den „Terminal“-Reiter in *PyCharm*) kopieren und unmittelbar ausführen.

Neben diesem Befehl und den Ausgaben unseres Programms sehen Sie in der Run-Konsole aber auch noch die seltsame Meldung **Process finished with exit code 0**. Sie bedeutet, dass Ihr Programm fehlerfrei durchgelaufen ist. Das ist natürlich der Optimalfall, den wir möglichst immer haben wollen. Sie werden aber schnell merken, dass Sie mehr Fehler – und damit meist Programmabbrüche – verursachen, als Ihnen lieb sein wird. Die Suche nach und Behebung von Fehlern gehört zum Programmieralltag wie Sägespäne zur Tischlerwerkstatt – nicht schön, aber unvermeidlich. Das gilt für Anfänger wie Profis gleichermaßen.

Deshalb lassen Sie uns doch einmal „künstlich“ einen Fehler verursachen! Wir haben ja derzeit ein lauffähiges Programm im Editorbereich stehen. Löschen Sie einfach mal bei einer der `print()`-Aufrufe eine der beiden schließenden Klammern. Führen Sie dann das Programm nochmal aus. Das können Sie wiederum mit dem

Run-Befehl aus dem Kontextmenü (rechte Maustaste) tun, oder aber – jetzt, da wir das Programm bereits ausgeführt hatten – auch mit dem grünen Play-Pfeil in der Symbolleiste. Wichtig dabei ist lediglich, dass in dem Dropdown-Auswahlfeld neben dem Play-Button der Name Ihres Programms zu lesen ist. Ausgeführt wird nämlich das Programm, das hier ausgewählt ist. Das mag aktuell, da wir nur eine Code-Datei offen haben, noch kein Problem darstellen, wenn Sie aber mehrere Python-Programme auf unterschiedlichen Registerkarten bearbeiten, achten Sie immer darauf, welches Skript Sie gerade ausführen. Dasjenige, das Sie ausführen, muss nicht unbedingt das sein, das in der gerade oben aufliegenden Registerkarte des Editors zu sehen ist!

Wenn Sie nun eine der schließenden Klammern entfernen, weist *PyCharm*, das im Hintergrund automatisch die Syntax Ihres Programms überprüft durch rote „Unterschlängelung“ an der Stelle der fehlenden Klammern sowie eine roter Markierung am Rand des Editor-Bereichs auf ein mögliches Problem hin. Fahren Sie mit der Maus über eine dieser Markierungen, erhalten Sie weitere Informationen in Form einer kleinen Einblendung. Wir werden aber nun ganz bewusst alle diese Warnhinweise ignorieren und stur unser (fehlerhaftes) Programm ausführen. Sie erhalten in der Run-Konsole dann eine rot hervorgehobene Fehlermeldung wie die folgende:

```
File "C:/[Pfad zur Ihrem Projekt]/hallowelt.py", line 2
    print("Hallo Welt")
    ^
SyntaxError: invalid syntax

Process finished with exit code 1
```

Die Fehlermeldungen, die der Python-Interpreter ausgibt, sind bei der Fehler-suche oftmals nur begrenzt hilfreich, wie Sie an diesem Beispiel sehr schön erkennen können. Der von *PyCharm* bereitgestellte Syntax-Check ist das vielfach deutlich nützlicher. Im Output erkennen Sie übrigens auch den **exit code** mit Wert 1. Der signalisiert, dass das Programm mit einem Fehler vorzeitig abgebrochen worden ist.

Obwohl Python eine interpretierte Sprache und als solche normalerweise langsamer ist als compilierte Programmiersprachen, so werden die Programme, die in den nächsten Kapiteln entwickeln werden, normalerweise sehr zügig durchlaufen. Gerade aber, wenn man mit Schleifen-Konstrukten arbeitet, was wir in ► Abschn. 24.4 tun werden, kann es durchaus einmal vorkommen, dass ein Programm länger läuft, insbesondere, wenn Sie ihre Schleife versehentlich so gebaut haben, dass sie niemals (zumindest nicht ohne äußeren Eingriff oder einen Mangel an Systemressourcen) zu einem Ende kommen würde. Dann ist es praktisch, wenn man ein Programm während es läuft abbrechen kann. Genau das lässt sich in *PyCharm* mit dem kleinen Stop-Button in der Symbolleiste bewerkstelligen. Während Ihr Skript läuft, ist dieser Button rot eingefärbt und kann angeklickt werden.

Eine der Registerkarten am unteren Rand der *PyCharm*-Oberfläche haben wir bislang noch gar nicht betrachtet, die *Python-Konsole*. Sie erlaubt es, Python *im interaktiven Modus* zu betreiben. Das bedeutet, wir können eine Python Anweisung eingeben und unmittelbar ausführen. Es wird also normalerweise nicht ein ganzes Skript, eine Folge von Anweisungen, eingegeben und dann en bloc ausgeführt (obwohl auch das möglich wäre), sondern lediglich eine einzelne Anweisung. Auf die Anweisung erfolgt die (wie auch immer genau geartete) Reaktion von Python und man kann eine neue Eingabe machen. Aufgrund dieses Wechselspiels zwischen Eingabe einer Anweisung und der Verarbeitung der Anweisung durch Python spricht man vom interaktiven Modus, manchmal auch von *REPL* (engl. *read-eval-print loop*, also Lese-Auswerten-Ausgeben-Schleife). Die Eingabe dabei erfolgt am sogenannten *Prompt*, das nichts anderes ist als die Aufforderung, eine Eingabe zu machen. In der Python-Konsole wird das Prompt durch drei Größer-Zeichen (**>>>**) markiert.

Klicken Sie am unteren Rand von *PyCharm* auf den Reiter „Python Console“. Geben Sie nun am Prompt einmal eine der `print()`-Anweisungen ein, die oben Bestandteil unseres Skripts sind, und bestätigen Sie mit Enter:

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

Sie sehen, dass unsere Anweisung tatsächlich direkt ausgeführt wird. Danach erscheint direkt ein neues Prompt, an dem wir weitere Anweisungen eingeben könnten.

Abb. 19.3 zeigt die Python-Konsole in *PyCharm* nach der Ausführung unserer Anweisung.

Im vorangegangenen Code-Abschnitt ist, wie generell im weiteren Verlauf dieses Teils des Buchs, das Prompt mit der typischen Zeichenfolge **>>>** symbolisiert. Beachten Sie bitte, dass Sie diese Größer-Zeichen nicht mit eingeben dürfen! Bei allem, vor dem kein Prompt steht, handelt es sich um eine Ausgabe seitens Python.

Die Konsole können Sie benutzen, um Python-Befehl Befehle schnell auszuprobieren. Auch die Hilfe können Sie – wie wir im vorangegangenen Kapitel gese-



Abb. 19.3 Ausführung von Anweisungen in der Python-Konsole in *PyCharm*

hen haben – von hier aufrufen (probieren Sie es aus und geben Sie einmal `help(print)` ein, um Hilfe-Informationen zur Funktion `print()` zu erhalten!). Beachten Sie bitte schon mal an dieser Stelle, dass die Python-Konsole und der Editor für die Python-Skripte zwei vollkommen verschiedene und sauber voneinander getrennte Welten darstellen. Insbesondere können Sie nicht aus der Python-Konsole heraus auf Variablen zugreifen, die Sie in Ihrem Skript verwenden. Dazu aber später mehr in Kapitel ► Kap. 21.

Natürlich können Sie die Python-Konsole auch direkt von der Kommandozeile (Konsole/Terminal) Ihres Betriebssystems aus starten. Führen Sie dazu einfach das Programm `python` ohne weitere Parameter aus. Es öffnet sich dann die Python-Konsole in Ihrem Betriebssystem-Terminal. Verlassen können Sie den interaktiven Modus wieder durch Eingabe der Anweisung `quit()`, die `python` beendet und Sie auf die Betriebssystem-Ebene zurückbringt.

### 19.3 PyCharm kennenlernen

---

Sie haben nun schon einige Funktionen der Oberfläche von *PyCharm* kennengelernt. *PyCharm* kann natürlich noch sehr viel mehr; allerdings werden wir die Mächtigkeit des Funktionsumfangs von *PyCharm* hier nicht vollständig ausschöpfen. Manche Funktionen sind erst dann wirklich relevant, wenn Sie auf professionellem Level Software entwickeln wollen. Und wie mit jeder herkömmlichen Büroanwendung auch, nutzen selbst die professionellsten Anwender nicht alle Möglichkeiten, die ihnen die Software bietet.

Die Oberfläche von *PyCharm* ist, wie die vieler integrierter Entwicklungsumgebungen, durchaus komplex mit ihren unterschiedlichen Fensterbereichen und Registerkarten, die teilweise auch noch in einander verschachtelt sind. Auch wenn wir für unsere Zwecke bereits mit einem relativ bescheidenen Umfang an Funktionen auskommen, so empfiehlt es sich doch, einfach ein wenig mit *PyCharm* herumzuspielen und die Oberfläche genauer kennenzulernen. Seien Sie neugierig und probieren Sie Dinge aus. Kaputt gehen kann bei Ihren Erkundungsreisen glücklicherweise nichts. In ► Abschn. 23.3.3, wo es um die Arbeit mit hinzustallierten Modulen geht, und in ► Abschn. 25.5, wo wir uns mit der Fehlersuche und -behebung beschäftigen, werden wir nochmal zur *PyCharm*-Oberfläche zurückkehren und einige weitere Funktionen kennenlernen.

### 19.4 Zusammenfassung

---

In diesem Kapitel haben wir uns damit beschäftigt, wie Sie mit *PyCharm* Python-Programme entwickeln und ausführen können.

Folgende Punkte sollten Sie aus diesem Kapitel unbedingt mitnehmen:

- *PyCharm* ist eine mächtige integrierte Entwicklungsumgebung mit einer Vielzahl an Funktionen, von denen wir im „Normalbetrieb“ nur einen kleinen Teil tatsächlich einsetzen; insbesondere bietet *PyCharm* durch Syntax-Highlighting

#### 19.4 · Zusammenfassung

und Live-Syntax-Prüfungen praktische Features, die die Entwicklung von Python-Programmen unterstützen.

- *PyCharm* kann mit unterschiedlichen Interpretern umgehen. Es ist daher notwendig, dass Sie, bevor Sie mit der Arbeit beginnen, einen Interpreter auswählen, den Sie verwenden wollen. Das müssen Sie auch dann tun, wenn Sie nur einen einzigen Interpreter zur Verfügung haben (nämlich den, den Sie mit der aktuellsten Python-Version installiert haben).
- Python-Programme laufen in *PyCharm* in der Run-Konsole.
- Daneben gibt es die Python-Konsole, die den Betrieb von Python im interaktiven Modus erlaubt; dabei werden Python-Befehle eingegeben und sofort ausgeführt.
- Python-Programme können auch direkt mit dem Python-Interpreter **python** ausgeführt werden, ohne *PyCharm* zu verwenden; wird der Python-Interpreter ohne eine Python-Skript-Datei als Argument aufgerufen, startet er im interaktiven Modus; dieser kann jederzeit durch Eingabe von **quit()** wieder verlassen werden.
- Mit der Funktion **print()** können Sie Ausgaben auf dem Bildschirm (genauer: der Python-Konsole) erzeugen.



# Wie stelle ich sicher, dass ich (und andere) mein Programm später noch verstehe?

## Inhaltsverzeichnis

- 20.1 Gestaltung des Programmcodes und Namenskonventionen – 246**
  - 20.1.1 Einrückungen und allgemeine Code-Formatierung – 246**
  - 20.1.2 Anweisungsende ohne Semikolon, Anweisungen über mehrere Zeilen – 248**
  - 20.1.3 Case-sensitivity und Wahl von Bezeichnern – 250**
- 20.2 Kommentare – 251**
- 20.3 Dokumentation mit Docstrings – 253**
- 20.4 Zusammenfassung – 255**

## Übersicht

Bevor wir richtig in die Python-Programmierung einsteigen, werden wir uns in diesem Kapitel zunächst damit beschäftigen, wie Python-Code und seine wichtigsten Grundelemente eigentlich aussehen und welche fundamentalen Konventionen es bei der Code-Gestaltung einzuhalten gilt. Python weist im Vergleich zu vielen anderen Programmiersprachen eine Besonderheit in der Code-Gestaltung auf, die uns als Programmierern zwar auf den ersten Blick etwas Gestaltungsfreiheit wegzunehmen scheint, aber unser Leben zugleich auch einfacher macht.

Darüber hinaus werden wir uns ansehen, wie Python-Code *kommentiert* und *dokumentiert* wird. Kommentierung ist wichtig, damit wir als Entwickler unseren eigenen Programmcode später noch verstehen, insbesondere dann, wenn wir ihn weiterentwickeln wollen. Die Dokumentierung hingegen dient dazu, Informationen bereitzustellen, sodass andere Entwickler, die unseren Code in ihren Programmen einsetzen wollen, verstehen, wie genau man das macht.

In diesem Kapitel werden Sie lernen:

- welche Bedeutung Zeileneinrückungen in Python haben
- wie in Python Anweisungen abgeschlossen werden und wie Anweisungen über mehrere Zeilen ausgedehnt werden können
- wie in Python Bezeichner für Variablen, Funktionen/Methoden und Module üblicherweise gewählt werden und welche Restriktionen es dabei gibt
- wie Sie in Python Kommentare formulieren können
- was Docstrings sind und wie Sie sie zur Dokumentierung des Programmcodes einsetzen können

## 20.1 Gestaltung des Programmcodes und Namenskonventionen

### 20.1.1 Einrückungen und allgemeine Code-Formatierung

Wir hatten in ► Abschn. 10.2 gesehen, dass viele Programmiersprachen dem Entwickler sehr viel Spielraum bei der Frage geben, wie der Quelltext formatiert sein soll. Durch Wahl geeigneter Einrückungen zum Beispiel kann man den Programmcode übersichtlicher gestalten. Diese Freiheit in der Gestaltung ist letztlich darauf zurückzuführen, dass viele Programmiersprachen die Formatierung des Programmcodes völlig ignorieren; anders ausgedrückt: Die Formatierung hat keine inhaltliche Bedeutung für das Programm.

Das ist in Python anders. Wo andere Programmiersprachen speziellen Symbole haben, um den Beginn und das Ende von Code-Blöcken zu markieren (zum Beispiel **begin** und **end**, oder geöffnete und geschlossene geschweifte Klammern), da verwendet Python Einrückungen: Anweisungen, die gleich weit eingerückt sind, gehören zum gleichen Code-Block.

Hier das Beispiel einer einfachen Funktion, der man einen Text sowie eine Zahl von Wiederholungen übergeben kann, und die den Text dann einfach entsprechend

oft auf dem Bildschirm ausgibt, allerdings in Großbuchstaben und mit der Ansage der aktuellen Wiederholungsnummer. Machen Sie sich keine Sorgen, dass Sie möglicherweise noch nicht den ganzen Code verstehen, das werden Sie am Ende dieses Teils des Buches, hier geht es zunächst nur um die Einrückungen:

```
def repeat_text(text: str, reps: int):
    text = text.upper()
    for x in range(1, reps+1):
        output = 'Durchlauf Nr.' + str(x) + ': ' + text
        print(output)

repeat_text('Hallo Welt', 3)
```

Nachdem mit **def** die Funktion erzeugt wurde, wird diese in der letzten Zeile dann aufgerufen. Das Programm erzeugt bei diesem Funktionsaufruf folgende Ausgabe:

```
Durchlauf Nr.1: HALLO WELT
Durchlauf Nr.2: HALLO WELT
Durchlauf Nr.3: HALLO WELT
```

Sie sehen, dass der Code-Block, der zur Funktion **repeat\_text()** gehört, eingerückt ist. Gleiches gilt für den Code-Block, der in der **for**-Schleife ausgeführt wird und die Ausgabe bewirkt. Dieser Block ist sogar zweifach eingerückt, er gehört nämlich zur Funktion **repeat\_text()** und darin wiederum zur **for**-Schleife **for x in range(1, reps+1):**. Hinter der Funktionsdefinition geht es mit **repeat\_text('Hallo Welt', 3)** wieder ganz links weiter. Diese Anweisung gehört also weder zum Code-Block der **for**-Schleife, noch sonst wie zum Code-Block der Funktionsdefinition.

Code-Blöcke werden in Python also durch Einrückung begrenzt (und durch einen Doppelpunkt eingeleitet, wie Sie sowohl an der Funktionsdefinition mit Hilfe von **def** als auch an der **for**-Schleife sehen können).

Die Einrückungen erfolgen üblicherweise entweder mit der Tabulatortaste oder durch Eingabe von vier Leerzeichen. Die zweite Art wird allgemein präferiert. Im Sinne eines konsistenten Codes sollten Sie aber vor allem nicht beide Arten der Einrückung in einem Programm zu vermischen.

Durch die erzwungenen Einrückungen ist Python-Code recht gut lesbar, der Verlust an Gestaltungsfreiheit ist daher sicherlich gut verkraftbar. Durch diese Art, Code-Blöcke abzugrenzen, ist es nicht mehr notwendig, dazu Klammern oder andere Schlüsselworte zu verwenden, deren Vergessen – insbesondere am Ende eines Code-Blocks – in anderen Programmiersprachen eine häufige Fehlerquelle ist.

Abgesehen von den Einrückungen sind Sie bezüglich der Gestaltung des Codes vollkommen frei.

In Form von Style Guidelines gibt es aber eine ganze Reihe von Vorschlägen und Empfehlungen, wie man Python-Code schreiben *sollte*. Deren Einhaltung trägt dazu bei, den Code lesbarer und verständlicher zu machen.

```

113
114     def repeat_text(text: str, reps: int):
115         text = text.upper()
116         for x in range(1, reps+1):
117             output='Durchlauf Nr.' + str(x) + ':' + text
118             print(output)
119     repeat_text('Hallo Welt', 3)
120

```

PEP 8: expected 2 blank lines after class or function definition, found 0

Abb. 20.1 Tool-Tip-Einblendung zur Code-Formatierung

Viele der Regeln sind allerdings sehr kleinteilig und finden sicherlich auch bei Python-Profis nicht immer ausnahmslose Beachtung. Der offizielle Style Guide ist als *Python Enhancement Proposal* (PEP) unter der laufenden Nummer PEP 8 auf der ► [python.org](http://python.org)-Website zu finden. Es schadet nicht, einen Blick dort hinein zu werfen.

Wenn Sie mit *PyCharm* arbeiten, bietet Ihnen die praktische Funktion „Reformat Code“ aus dem Menü „Code“ die Möglichkeit, Ihren Code automatisch entsprechend der PEP 8-Regeln formatieren zu lassen.

Auch weist Sie *PyCharm* mit welligen, grauen Unterstreichungen und kleinen Tooltip-Einblendungen, wie Sie sie in Abb. 20.1 sehen, auf „Verstöße“ gegen die PEP 8-Regeln hin.

### 20.1.2 Anweisungsende ohne Semikolon, Anweisungen über mehrere Zeilen

Python verlangt grundsätzlich am Ende einer Anweisung kein Begrenzungszeichen wie etwa ein Semikolon. Jede Anweisung endet einfach automatisch am Zeilenende.

Auch wenn es der geringeren Übersichtlichkeit wegen nicht anzuraten ist, können Sie in Python auch mehrere Anweisungen in eine Zeile schreiben. Dann allerdings müssen Sie die einzelnen Anweisungen tatsächlich mit einem Semikolon von einander trennen.

Umgekehrt gibt es aber auch Möglichkeiten, *eine* Anweisung über mehrere Zeilen zu auszudehnen. Das ist dann ratsam, wenn die Anweisung sehr lang ist. Natürlich könnten man die Anweisung auch einfach in eine lange Zeile schreiben. Allerdings müsste man dann unter Umständen horizontal scrollen, um das Ende der Anweisung sehen zu können. Das ist umständlich und sollte vermieden werden. Nach den Code-Styling-Empfehlungen von PEP 8 sollten Code-Zeilen nicht länger als 79 Zeichen sein. Nun gibt es aber Anweisungen, die eben einfach sehr lang sind, auch deutlich länger als 79 Zeichen. Was also tun?

Anweisungen können innerhalb von runden, eckigen oder geschweiften Klammern einfach umgebrochen werden:

```
repeat_text('Hallo Welt', 3)

repeat_text('Hallo Welt',
3)
```

Den Funktionsaufruf aus unserem Beispiel oben könnten wir also auch in zwei Zeilen darstellen. Dabei darf allerdings nicht innerhalb der Zeichenkette '**Hallo Welt**' umgebrochen werden!

Einrückungen sind hierbei nicht reglementiert, sie können also ganz nach Belieben einrücken, was unter Umständen ganz erheblich zur Lesbarkeit des Programmcodes beiträgt.

Aber auch außerhalb von runden, eckigen und geschweiften Klammern können Sie Anweisungen umbrechen, wenn Sie ans Zeilenende einen Backslash (\) setzen:

```
x = \
'Hello Welt'

print(x)
```

Diese Methode gilt aber als verpönt und sollte nur eingesetzt werden, wenn unbedingt nötig. Meistens sind die Umbrüche ohnehin bei Funktionsaufrufen mit vielen Funktionsargumenten notwendig, und dort befindet man sich dann ohnehin innerhalb von runden Klammern.

Eine letzte Art umgebrochener Anweisungen sind die sogenannten *Docstrings*, mit denen wir uns gleich in ► Abschn. 20.3 beschäftigen, bzw. ganz generell Zeichenketten, die in dreifachen Anführungszeichen eingeschlossen sind. Diese können über mehrere Zeilen laufen:

```
x = """ Das ist ein sehr langer Text, der nicht in eine Zeile
paßt und deshalb auf mehrere Zeilen verteilt werden muss."""
```

Eine „normale“ Zeichenkette kann das nicht. Lassen wir eine solche über das Zeilenende hinauslaufen, vermutet der Python-Interpreter, wir hätten vergessen, die Zeichenkette abzuschließen. Die Zuweisung

```
x = 'Das ist ein sehr langer Text, der nicht in eine Zeile paßt
und deshalb auf mehrere Zeilen verteilt werden muss.'
```

führt dementsprechend zu einer Fehlermeldung:

```
SyntaxError: EOL while scanning string literal
```

### 20.1.3 Case-sensitivity und Wahl von Bezeichnern

Python ist case-sensitive, das heißt, Groß- und Kleinschreibung werden generell unterschieden. Eine Variable namens **alter** ist also eine gänzlich andere als die Variablen **Alter** oder **ALTER**. Die Case-sensitivity bezieht sich natürlich nicht nur auf Variablen, sondern auch auf die Bezeichner etwa von Funktionen und Methoden, Klassen, Modulen und Packages.

Bezeichner all dieser Elemente können aus Buchstaben, Ziffern und dem Unterstrich (engl. *underscore*, `_`) bestehen. Beginnen dürfen sie allerdings niemals mit einer Ziffer. **alter\_kunde** ist also ein zulässiger Variablen-Name, **nimm2** und **\_2mal4** ebenso, nicht aber **11freunde**.

Bezeichner, die mit einem Unterstrich (oder zwei Unterstrichen) beginnen und/oder aufhören, haben in Python eine besondere Bedeutung, mit der wir uns später noch genauer befassen werden. Es empfiehlt sich, auf Bezeichner, die mit Unterstrich beginnen oder aufhören, grundsätzlich zu verzichten und den Unterstrich nur *innerhalb* von Bezeichnern zu verwenden, es sei denn, man beabsichtigt eine der Wirkungen, die mit führenden oder abschließenden Unterstrichen verbunden sind.

Abgesehen von diesen wenigen Regeln sind Sie in der Wahl Ihrer Bezeichner frei. Es gibt aber einige Konventionen, an die sich viele Python-Programmierer halten, obwohl die Verletzung dieser Regeln nicht zu Syntaxfehlern im Programmcode führt. So beginnen Klassen-Namen bzw. ihre Bestandteile normalerweise mit Großbuchstaben (also zum Beispiel **MeineKlasse**), Modulnamen mit Kleinbuchstaben (zum Beispiel **meinModul**), Variablen und Funktionen/Methoden üblicherweise mit Kleinbuchstaben, wobei unterschiedliche Bestandteile regelmäßig mit Unterstrichen getrennt werden (zum Beispiel **meine\_variable**, **meine\_funktion**).

Sie könnten übrigens sogar Umlaute in Ihren Bezeichnern verwenden. Python unterstützt den umfangreichen UTF-8 Zeichensatz, der dies erlaubt. Zu empfehlen sind solche Bezeichner dennoch nicht. Spätestens der Code-Austausch mit Entwicklern, deren Muttersprache (und deren Tastaturen!) diese Zeichen gar nicht kennen, wird so erheblich erschwert. Am besten bleiben Sie bei Bezeichnern, die sich mit einer englischen Standardtastatur gut tippen lassen.

Selbstverständlich müssen Sie sich an all' diese nicht-bindenden Konventionen nicht halten. Wenn Sie es aber tun, sieht Ihr Programmcode „python-iger“ aus und andere Entwickler werden mit Ihrem Code besser zureckkommen, nicht zuletzt deshalb, weil sie sich auf die Bedeutung und Funktionsweise des Codes konzentrie-

ren können und nicht regelmäßig an den für sie ungewohnten Bezeichnern „hängenbleiben“.

## 20.2 Kommentare

Kommentare sind, wie Sie bereits wissen, Teile des Programmcodes, die vom Python-Interpreter einfach ignoriert werden und die Sie verwenden können, um für sich oder für andere, die Ihren Programmcode lesen, Notizen und Erläuterungen zu hinterlassen.

Solche Kommentare werden mit dem Raute-Zeichen (#) eingeleitet. Alles was rechts davon steht, gilt als Kommentar.

Das Kommentar-Symbol muss nicht unbedingt ganz am Anfang einer Zeile stehen. Im folgenden Beispiel sehen Sie beide Varianten, einmal am Anfang der Zeile, einmal weiter rechts:

```
# Unser erstes kleines Python-Programm
print('Hallo Welt') # Ausgabe auf dem Bildschirm
```

Die irgendwo mitten auf der Zeile beginnenden Kommentare nennt man auch *Inline*-Kommentare. Wie in wohl allen Programmiersprachen sind sie auch in Python nicht gerne gesehen, weil sie den Code etwas schwieriger lesbar machen. Wenn man Sie verwendet, sollte man auf jeden Fall darauf achten, dass einige (empfohlen wird offiziell: zwei) Leerzeichen zwischen dem Ende des Programmcodes und dem Kommentar-Symbol stehen.

Kommentare können Sie natürlich auch dazu verwenden, Programmcode vorübergehend „auszuschalten“, indem Sie ihn dem Zugriff des Interpreters entziehen, der ja alles rechts vom Kommentarzeichen ignoriert. Man spricht sinnigerweise in diesem Zusammenhang auch vom *auskommentieren*. Genau das ist mit der zweiten Printanweisung in diesem Beispiel geschehen:

```
# Unser erstes kleines Python-Programm
print('Hallo Welt')
# print('Hello, world')
```

Kommentare reichen immer bis zum Zeilenende, mehrzeilige Kommentare kennt Python grundsätzlich nicht. Wenn Sie also über mehrere Zeilen schreiben wollen – und das werden Sie hin und wieder, denn Ihre Kommentare sollten auf einem normalen Bildschirm auch ohne waagerechtes Scrollen voll lesbar sein (empfohlen sind offiziell höchstens 79 Zeichen pro Zeile) –, dann müssen Sie jede Zeile mit einem Extra-Kommentarsymbol einleiten. Viele Entwicklungsumgebungen nehmen Ihnen hier die Arbeit aber ab. Wenn Sie mit *PyCharm* arbeiten,

können Sie die Zeilen, die Sie auskommentieren wollen, markieren und dann im Menü „Code“ die Option „Comment With Line Comment“ anklicken (oder alternativ die Tastenkombination <CTRL> und </> drücken). Mit der gleichen Option können Sie die Kommentarsymbole von kommentierten Zeilen auch wieder entfernen.

Lassen Sie sich übrigens nicht von der Menü-Option „Comment With Block Comment“ verwirren, Python unterstützt wie gesagt – anders als manche andere Programmiersprache – keine Blockkommentare, also Kommentare über mehr als eine Zeile. Es gibt aber einen Trick, mit dem man trotzdem so etwas wie einen mehrzeiligen Kommentar herstellen kann. Dazu mehr im nächsten Abschnitt.

Über Sinn und Unsinn von Kommentaren kann vortrefflich philosophiert und gestritten werden, insbesondere bzgl. der Frage, wann Kommentare hilfreich sind und wann sie einfach nur den Code umfangreicher und mithin weniger übersichtlich machen.

Blättern Sie noch mal einige Seiten zurück zu ► Abschn. 10.3. Dort haben wir eine Reihe von Best Practices rund um die Kommentierung von Programmcode bereits kennengelernt. Als Daumenregel kann man aber festhalten: Es wird meist eher zu wenig kommentiert, also trauen Sie sich ruhig, Kommentare zu schreiben, Ihr „zukünftiges Ich“, das Ihren alten Programmcode nochmal zu verstehen versucht, wird es Ihnen danken!

Manchmal nutzt man Kommentare auch, um offene Punkte, die noch erledigt werden müssen, direkt an der entsprechenden Stelle im Programmcode zu dokumentieren. Wenn Sie mit *PyCharm* arbeiten, können Sie solche Kommentare mit **# TODO** einleiten:

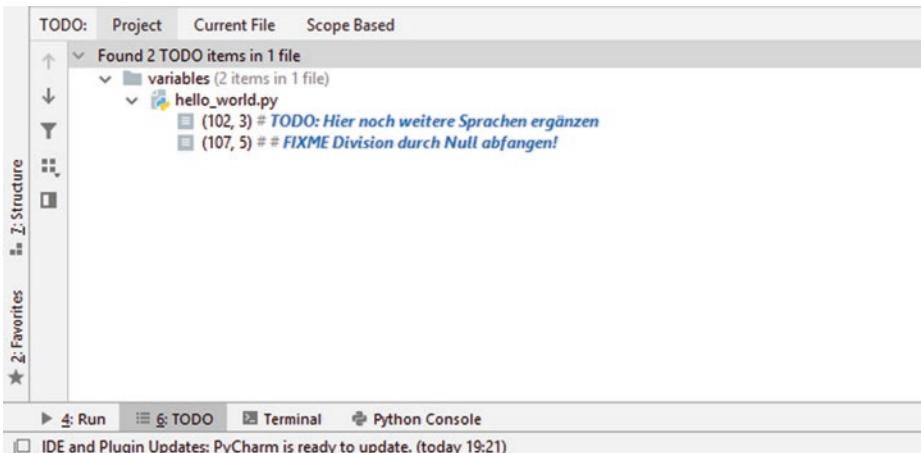
```
# TODO: Hier noch weitere Sprachen ergänzen
print('Hallo Welt')
print('Hello, world')
```

In ähnlicher Weise können Sie kleine Fehler, die noch behoben werden müssen, an der entsprechenden Code-Stelle mit **# FIXME** markieren und erläutern. Kommentare, die so beginnen, werden im Rahmen des Syntax Highlighting extra hervorgehoben und von *PyCharm* besonders behandelt.

*PyCharm* zeigt diese Art von Kommentaren nämlich in einem speziellen Fensterbereich namens „TODO“ an, den Sie auch mit der Tastenkombination <Alt> und <6> öffnen und in □ Abb. 20.2 sehen können. Dort werden alle **TODO**- und **FIXME**-Kommentare angezeigt; mit einem Doppelklick auf einen Eintrag im „TODO“-Bereich gelangen Sie direkt an die entsprechende Stelle im Code.

Wenn Sie möchten, können Sie über die Einstellungen von *PyCharm* sogar eigene Kommentartypen analog zu **# TODO** und **# FIXME** definieren. In den allermeisten Fällen sollten aber die beiden „im Lieferumfang enthaltenen“ besonderen Kommentartypen genügen.

### 20.3 · Dokumentation mit Docstrings



■ Abb. 20.2 Der Fensterbereich TODO in PyCharm

## 20.3 Dokumentation mit Docstrings

Neben den „echten“ Kommentaren, die immer mit dem Kommentarsymbol `#` eingeleitet werden, gibt es noch eine zweite Möglichkeit, Informationen im Code zu hinterlegen, die nicht vom Interpreter ausgeführt werden, und zwar mit Hilfe sogenannter *Docstrings*. Docstrings sind spezielle Zeichenketten, die in dreifachen Anführungszeichen geschrieben werden. Betrachten Sie als Beispiel das folgende Programm:

```
'''  
Das ist ein Docstring für unser Hallo-Welt-Programm, und zwar  
einer, der sogar über mehrere Zeilen reicht.  
''  
  
print('Hallo Welt!')
```

Führen Sie dieses Programm aus, so erhalten Sie die Ausgabe **Hallo Welt!**. Der Docstring dagegen wird nicht angezeigt. Probieren wir etwas anderes:

```
'Das ist ein normaler String in unserem Hallo-Welt-Programm.'  
  
print('Hallo Welt!')
```

Dieses Mal verwenden wir keinen Docstring mit drei Anführungszeichen, sondern eine ganze gewöhnliche Zeichenkette, ähnlich wie '**Hallo Welt!**', das wir unten ausgeben. Anders als die Docstrings können sich normale Strings in Python im Pro-

grammcode nicht über mehrere Zeilen erstrecken, deshalb haben wir unseren „Kommentar“ in einer Zeile untergebracht. Was passiert nun, wenn Sie dieses Programm ausführen? Es ändert sich nichts! Wieder wird nur '**Hallo Welt!**' ausgegeben.

Die Ursache ist ganz einfach: Wann immer Sie in Ihren Python-Programmcode einen Text hineinschreiben, wie wir es zuerst mit dem Docstring und im zweiten Beispiel dann mit einer normalen Zeichenkette getan haben, führt das nicht zu einer Ausgabe auf dem Bildschirm. Der Text wird stattdessen einfach ignoriert (technisch stimmt das zwar nicht ganz, aber zumindest gibt es keinen sichtbaren Effekt). Gleicher gilt für Variablen, wie Sie im folgenden Kapitel sehen werden: Geben wir ohne weitere Anweisung einfach nur den Namen einer Variablen in unseren Programmcode ein, passiert überhaupt nichts. Um den Inhalt der Variable oder eben unseren Text aus dem Beispiel oben anzuzeigen, müssen wir Python explizit anweisen, ihn auszugeben. Das tun wir durch Aufruf der Funktion **print()**.

Anders ist es in der Konsole. Geben Sie dort den Namen einer Variablen ein und drücken <ENTER>, wird Ihnen der Inhalt angezeigt. Geben Sie eine Zeichenkette in die Konsole ein, wird Ihnen die Zeichenkette sofort wieder in der Konsole ausgegeben.

Wie alle anderen Zeichenketten auch, werden also auch die Docstrings, wenn sie ohne weitere Anweisung im Programmcode stehen, nicht auf dem Bildschirm angezeigt. Wenn das so ist, können wir nicht einfach auch normale Zeichenketten zur Erläuterung unseres Codes verwenden?

Ja, das würde funktionieren. Docstrings haben aber zwei spezielle Eigenschaften, derentwegen sie sich, wie ihr Name ja bereits suggeriert, zur *Dokumentation* von Programmcode besonders eignen: Zum einen nämlich können sie mehrere Zeilen umfassen; das haben wir bereits gesehen. Zum anderen behandelt Python diese Docstrings in besonderer Weise. Stehen Sie nämlich am Anfang einer Funktion, einer Klassendefinition oder eines Moduls, so werden sie als Inhalt der Hilfe zu dieser Funktion, dieser Klasse oder diesem Modul verwendet. Rufen Sie also mit der Funktion **help()** die Hilfe auf (zum Beispiel für **print()**: **help(print)**), so sehen Sie den Docstring, der am Anfang des jeweiligen Programmcodes hinterlegt ist.

Es gibt zahlreiche Python-Werkzeuge, die diese Docstrings verarbeiten und in Form einer Dokumentation ausgeben. Die Python-eigene Hilfe verwendet ein Tool namens *pydoc*, das bei Aufruf von **help()** den Docstring aus dem Code extrahiert und anzeigt. Daneben gibt es eine ganze Reihe weiterer Hilfsprogramme, die mit den Docstrings arbeiten, zum Beispiel *autodoc*, *doxygen* und *pydoctor*. Einige dieser Programme sind speziell für Python entwickelt, andere erlauben die automatische Generierung von Code-Dokumentation für unterschiedliche Programmiersprachen. Der Output dieser Dokumentation muss dabei nicht einfach nur Text in der Python-Konsole sein, etliche Tools unterstützen auch die Erzeugung von HTML-, PDF- oder sogar LaTex-Dokumenten. Um die Dokumentation sauber zu strukturieren, setzen einige Werkzeuge dabei einen speziellen Aufbau der Docstrings voraus.

Natürlich haben sich längst kluge Köpfe mit der Frage beschäftigt, wie Docstrings im Allgemeinen aussehen sollten. Die offizielle Antwort auf diese Frage findet sich in den *Python Enhancements Proposals* (PEP) als PEP 257 (*Docstring Conventions*).

Anders als Kommentare werden Docstrings eher dazu verwendet, Code für andere Benutzer zu dokumentieren, also zu erläutern, wie man den Code für eigene Zwecke einsetzt und weniger, um sich Notizen darüber zu machen, wie der Code funktioniert. Dafür werden normalerweise die Kommentare verwendet.

Wir werden an einigen späteren Stellen auf die Docstrings zurückkommen und sie in unseren Programmen einsetzen.

Eine weitere Art der Dokumentation werden wir uns später ebenfalls noch genauer ansehen. Dabei handelt es sich um die sogenannten *Function annotations*, die es erlauben, die erwarteten Datentypen von Funktionsargumenten sowie die Datentypen der Rückgabewerte von Funktionen im Programmcode zu dokumentieren. Auch dies sind Informationen, die von *pydoc* für die Hilfe verwendet werden und natürlich auch von anderen Dokumentationstools verarbeitet werden können.

Die Arbeit mit Docstrings und Function annotations richtet sich häufig an ein anderes Publikum als Sie selbst, nämlich die *Verwender* Ihres Codes. Wenn Sie Python-Code schreiben, der von anderen Entwicklern eingesetzt werden soll, ist es wichtig, zu dokumentieren, was zum Beispiel Funktionen tun, was ihre Argumente bedeuten und welche Rückgabewerte der Entwickler, der Ihren Programmcode nutzt, erwarten kann. Er möchte nicht Ihren Code lesen müssen, um diese Informationen herauszubekommen. Deshalb ist es wichtig, eine Dokumentation bereitzustellen, die es Dritten erlaubt (sofern das vorgesehen ist), Ihren Programmcode „blind“ einzusetzen und seine innere Funktionsweise wie eine Blackbox zu betrachten. Relevant für die Nutzer Ihres Codes ist nur die *Schnittstelle*, also wie man ihn verwendet, nicht wie er im Detail funktioniert. Dazu sind Docstrings und Function annotations geeignete Hilfsmittel.

## 20.4 Zusammenfassung

---

In diesem Kapitel haben wir uns damit befasst, welche Besonderheiten Python bei der Formatierung des Programmcodes aufweist, wie Bezeichner (zum Beispiel für Variablen und Funktionen/Methoden) aufgebaut sein dürfen/müssen, und wie Sie Ihren Programmcode kommentieren und dokumentieren können.

Folgende Punkte sollten Sie aus diesem Kapitel unbedingt mitnehmen:

- Einrückungen markieren in Python einen zusammenhängenden Code-Block und können deshalb nicht beliebig eingesetzt werden; Python „erzwingt“ also gewissermaßen bereits eine gut lesbare Formatierung des Programmcodes.
- Anweisungen enden in Python normalerweise am Zeilenende, ein besonderes Begrenzungszeichen ist nicht notwendig.
- Anweisungen können über mehrere Zeilen gehen, wenn der Zeilenumbruch innerhalb von runden, eckigen oder geschweiften Klammern erfolgt oder durch den Backslash (\) besonders markiert ist, wobei letztere Art von Zeilenumbrüchen eher selten anzutreffen ist.
- Mehrere Anweisungen können durch Semikolons voneinander getrennt auch in einer einzigen Zeile stehen.
- Python ist case-sensitive, unterscheidet also zwischen Groß- und Kleinschreibung.

- Bezeichner dürften aus Buchstaben, Ziffern und dem Unterstrich bestehen.
- Anders als Ziffern dürfen Buchstaben und Unterstriche auch am Anfang eines Bezeichners stehen.
- Da (einfache und doppelte) Unterstriche am Beginn (und auch am Ende) von Bezeichnern eine jeweils spezielle Bedeutung haben, sollte im Allgemeinen auf mit Unterstrichen beginnende Bezeichner verzichtet werden.
- Bezeichner von Variablen und Funktionen/Methoden werden in Python meist kleingeschrieben, ihre Bestandteile, wenn es sich um zusammengesetzte Begriffe handelt, mit Unterstrichen voneinander getrennt.
- Kommentare zur Erläuterung des Codes sind in Python grundsätzlich einzeilig und werden mit dem Kommentarsymbol `#` eingeleitet; alles, was rechts davon steht, wird nicht als Programmcode interpretiert, sondern vom Interpreter ignoriert.
- Docstrings, die in dreifachen Anführungszeichen stehen, werden in Python zur Dokumentation für Anwender des Codes verwendet; sie sind der wesentliche Bestandteil der Python-Hilfe, die über `help()` aufrufbar ist und werden von zahlreichen Entwickertools zu verschiedenen Formaten von Dokumentation weiterverarbeitet.

Docstrings können über mehrere Zeilen reichen.



# Wie speichere ich Daten, um mit ihnen zu arbeiten?

## Inhaltsverzeichnis

- 21.1 Variablen erzeugen und zuweisen – 259**
- 21.2 Variablen löschen – 261**
- 21.3 Grundtypen von Variablen – 261**
  - 21.3.1 Zahlen (int, float) – 262
  - 21.3.2 Zeichenketten (str) – 262
  - 21.3.3 Wahrheitswerte (bool) – 264
  - 21.3.4 None – 265
  - 21.3.5 Weitere Datentypen – 266
- 21.4 Variablen als Objekte – 267**
  - 21.4.1 Attribute und Methoden von Variablen – 267
  - 21.4.2 Erzeugen von Variablen mit der Konstruktor-Methode – 272
- 21.5 Konvertieren von Variablen – 273**
- 21.6 Komplexe Datentypen – 275**
  - 21.6.1 Listen – 275

- 21.6.2 Tupel – 282
  - 21.6.3 Dictionaries – 284
  - 21.6.4 Sets – 287
- 21.7 Selbstdefinierte Klassen – 289**
- 21.7.1 Klassen definieren und verwenden – 289
  - 21.7.2 Klassen aus anderen Klassen ableiten – 290
  - 21.7.3 Doppeldeutigkeit vermeiden: Name mangling – 292
- 21.8 Zusammenfassung – 293**
- 21.9 Lösungen zu den Aufgaben – 296**

## Übersicht

Mit diesem Kapitel wenden wir uns nun den Variablen in Python zu. Python ist eine objektorientierte Sprache und alle Variablen sind zugleich Objekte mit Attributen und Methoden. Deshalb beschäftigen wir uns hier nicht nur mit den Datentypen von Variablen und was man jeweils mit ihnen machen kann, sondern auch damit, wie Objektorientierung in Python umgesetzt ist.

In diesem Kapitel werden Sie lernen:

- wie Sie Variablen in Python erzeugen und ihnen Werte zuweisen
- welche Grundtypen von Variablen es gibt, und wie sie Sie verwenden
- worin der Objektcharakter von Variablen zum Ausdruck kommt und was das für Ihre praktische Arbeit mit Variablen bedeutet
- wie Sie Variablen von einem Datentyp in einen anderem konvertieren können, und wo Python Ihnen die Konvertierung automatisch abnimmt
- welche komplexeren Datentypen (beispielsweise assoziative Felder, sogenannte *Dictionaries*) es gibt, und wie Sie sie einsetzen
- wie Klassendefinitionen in Python funktionieren, und wie Sie selbst Objekt-Klassen definieren und mit ihnen arbeiten.

## 21.1 Variablen erzeugen und zuweisen

---

Anders als in manchen anderen Sprachen ist die Erzeugung von Variablen in Python ganz einfach. Denn Variablen müssen in Python nicht deklariert werden; sie werden einfach bei der ersten Benutzung automatisch angelegt. So erzeugt die Zuweisung

```
>>> x = 5
```

eine (Ganzzahl-)Variable und setzt ihren Wert auf 5 (das **>>>** ist das bekannte Prompt-Zeichen, dass Sie zur Eingabe auffordert, dieses also nicht mit eingeben!).

Wir haben unsere Variable hier der Einfachheit halber **x** genannt. In Kapitel ► Kap. 11 hatten wir gesagt, dass Variablen-Namen idealerweise aussagekräftig sind und den Leser des Codes erahnen lassen, welche Art von Inhalt die Variable haben wird. Auch wenn wir es uns hier in diesen Beispielen sehr einfach machen, Python bietet Ihnen alle Möglichkeiten, aussagekräftige Variablen-Namen zu verwenden. Wie Sie bereits aus ► Abschn. 20.1.3 wissen, können Namen in Python aus Groß- und Kleinbuchstaben, Ziffern sowie dem Unterstrich bestehen. Ziffern dürfen dabei nicht am Anfang des Variablen-Namens stehen, sonst allerdings überall. Darüber hinaus hat der Unterstrich am Beginn (und teilweise auch am Ende) von Namen in Python eine besondere Bedeutung, auf die wir später noch zu sprechen kommen werden. Es ist deshalb anzuraten, Variablen-Namen grundsätzlich nicht mit einem Unterstrich beginnen oder enden zu lassen. Ansonsten sind Sie aber vollkommen frei, wie Sie Ihre Variablen benennen wollen.

Python entscheidet selbst, welchen Typ die Variable haben soll. Über die Lebensdauer der Variable kann sich der Typ durchaus ändern, zum Beispiel, indem wir der Variablen eine andere Art von Daten zuweisen. Mit

```
>>> x = 'Eine Zeichenketten (str)-Variable'
```

wird nicht nur der Wert, sondern auch der Datentyp der Variable geändert, sie ist jetzt eine Zeichenketten-Variable.

Im nächsten Abschnitt, wenn wir uns mit dem Objekt-Charakter von Variablen beschäftigen, werden Sie über die Wertzuweisung hinaus noch eine zweite Art kennenlernen, Variablen in Python anzulegen.

Wenn Sie mit der Python-Konsole arbeiten, können Sie sich den Wert einer Variablen jederzeit anzeigen lassen, in dem Sie ihren Namen eingeben.

```
>>> x
'Eine Zeichenketten (str)-Variable'
```

Wenn Sie Ihren Code dagegen in ein Python-Skript schreiben, müssen die mit der Funktion **print()** arbeiten, die Sie bereits im vorangegangenen Kapitel kennengelernt haben, um sich den Inhalt der Variable ausgeben zu lassen:

```
print18(x)
```

Schreiben Sie dagegen nur den Variablen-Namen in Ihr Python-Programm, so erfolgt keine Ausgabe.

Auf Variablen, die Sie in Ihrem Programm erzeugen, können Sie übrigens nicht in der Konsole zugreifen. Der Namensraum der Konsole und der Ihres Programms sind voneinander getrennt. Verwenden Sie in der Konsole eine Variable, die Sie in im Programm erzeugt haben, so erhalten Sie eine Fehlermeldung (es sei denn natürlich, Sie hätten über die Konsole bereits eine Variable gleichen Namens angelegt – dann aber würden Sie auch mit dieser Variablen arbeiten und nicht mit der, die Sie im Rahmen Ihres Programms verwenden).

Manchmal werden Sie aus Versehen auch eine Variable zugreifen, die es gar nicht gibt, zum Beispiel, weil Sie sich beim Bezeichner vertippt haben. Dann erhalten Sie eine Fehlermeldung wie die folgende:

```
Traceback (most recent call last):
  File "D:\Applications\Anaconda\lib\site-packages\IPython\core\interactiveshell.py", line 2961, in run_code
    exec(code_obj, self.user_global_ns, self.user_ns)
  File "<ipython-input-16-9063a9f0e032>", line 1, in <module>
    y
NameError: name 'y' is not defined
```

Entscheidend dabei ist die letzte Zeile. Sie sagt uns, dass eine Variable namens `y`, auf die wir hier zuzugreifen versucht haben, gar nicht existiert.

## 21.2 Variablen löschen

Einmal angelegte Variablen können mit Hilfe des Befehls `del` auch wieder gelöscht werden. Das macht vor allem dann Sinn, wenn die Variable sehr viel Speicher belegt (zum Beispiel, wenn Sie eine große Datei vollständig eingelesen haben) und Sie den Speicher wieder freigeben wollen, nachdem Sie die Daten nicht mehr benötigen.

Wenn Sie in der Python-Konsole eine Variable löschen und danach versuchen, darauf zuzugreifen, erhalten Sie konsequenterweise eine Fehlermeldung:

```
>>> del x
>>> x
Traceback (most recent call last):
  File "<input>", line 1, in <module>
NameError: name 'x' is not defined
```

In der Fehlermeldung wird davon gesprochen, der *Name x* sei nicht definiert. Wie viele andere Programmiersprachen auch, unterscheidet Python klar zwischen dem Wert der Variablen und ihrem Namen. Der Name ist nur ein Verweis auf den Wert, der in einem bestimmten Bereich des Speichers steht. Name und Wert existieren prinzipiell unabhängig voneinander. Es könnte nun sein, dass mehrere Namen auf exakt denselben Wert verweisen, also auf dieselbe Stelle im Speicher. Ändert sich der in dem betreffenden Speicher abgelegte Wert, so ändern sich auch die Werte aller dieser Variablen entsprechend. In einer solchen Situation mit mehreren Namen, die alle auf denselben Speicherbereich zeigen, bleibt der Wert und die übrigen Namen (und damit Variablen) erhalten, wenn Sie einen Namen löschen. Sie können den Wert dann einfach nur nicht mehr unter dem gelöschten Namen ansprechen, sondern nur noch unter den verbleibenden Namen.

Python zählt die Namen, die auf einen Wert verweisen (sogenannter *reference counter*). Wenn es keinen Namen mehr gibt, der auf einen bestimmten Wert zeigt, wird der Wert selbst von Python gelöscht. Diesen Vorgang nennt man *garbage collection* („Müllabfuhr“). Da es in der Regel aber nur einen einzigen Namen geben wird, der eine Verknüpfung (*binding*) mit dem Wert Ihrer Variable besitzt, wird normalerweise, wenn Sie den `del`-Befehl aufrufen, auch der Wert selbst gelöscht und der betreffende Speicher freigegeben.

## 21.3 Grundtypen von Variablen

In diesem Abschnitt beschäftigen wir uns mit den wichtigsten Arten von Variablen. Dabei konzentrieren wir uns zunächst auf Variablen, die nur einen Wert beinhalten. Im darauffolgenden Abschnitt schauen wir uns dann komplexere Datentypen an, die zugleich mehrere Werte aufnehmen können.

### 21.3.1 Zahlen (int, float)

Zahlen kommen in Python als Ganzzahlen (**int**) oder Fließkommazahlen (**float**) daher. Anders als in vielen anderen Programmiersprachen haben **int**-Variablen in Python keinen festen Wertebereich, bei dessen Überschreitung, ein anderer, mehr Speicherplatz in Anspruch nehmender Datentyp gewählt werden muss. Python reserviert einfach so viel Speicher für die Variable, wie benötigt wird, um den darin enthaltenen Wert aufzunehmen.

Das können Sie sehr schön sehen, wenn Sie sich mit Hilfe der Funktion `sys.getsizeof(objekt)` die Größe der Variablen anschauen:

```
>>> x = 5
>>> sys.sizeof(x)
14

>>> x = 1000000000000
18
```

Wie Sie sehen, ist der Speicherbedarf von ursprünglich 14 Bytes auf 18 Bytes gestiegen, nachdem wir der Variablen statt 5 einen erheblich größeren Wert, nämlich eine Billion, zugewiesen haben.

Vielleicht wundern Sie sich, warum selbst ein kleiner Wert wie 5 immerhin 14 Bytes im Speicher benötigt. In vielen anderen Programmiersprachen hätte eine solche Variable die Größe von nur 2 Bytes. Damit lassen sich immerhin Zahlen zwischen 0 und  $2^{16} = 65536$  darstellen. Warum also ist Python so ein „Speicherschlucker“? Die Antwort hängt mit der Art zusammen, wie Python Variablen ablegt und wird uns im folgenden Abschnitt beschäftigen.

Das Dezimaltrennzeichen bei Fließkommazahlen ist der Punkt, wie es im englischen Sprachraum üblich ist. Das größte Problem damit in Python ist, dass man, wenn man statt des Punktes gewohnheitsmäßig das Komma verwendet, keine Fehlermeldung erhält:

```
>>> pi = 3,1415926535
>>> pi
(3, 1415926535)
```

Python hat unsere Eingabe nämlich missinterpretiert und eine Variable eines ganz anderen Typs erzeugt, nämlich ein *Tupel*. Mit diesen Tupeln werden wir uns etwas später in diesem Kapitel noch eingehender beschäftigen.

### 21.3.2 Zeichenketten (str)

Zeichenketten, Variablen vom Typ **str**, können Sie in Python entweder in einfache oder in doppelte Anführungszeichen setzen:

### 21.3 · Grundtypen von Variablen

```
>>> x = "Ein Text in doppelten Anführungszeichen."  
>>> x  
'Ein Text in doppelten Anführungszeichen.'  
  
>>> y = 'Ein Text in einfache Anführungszeichen.'  
>>> y  
'Ein Text in doppelten Anführungszeichen.'
```

Die Möglichkeit, beide Arten von Anführungszeichen zu benutzen, hat den Vorteil, dass Sie in Python keine Schwierigkeiten haben, Anführungszeichen innerhalb eines Textes darzustellen, denn durch die zwei unterschiedlichen Varianten von Anführungszeichen besteht keinerlei Verwechslungsgefahr zwischen den Anführungszeichen, die Bestandteil des Textes sind und denen, die die Zeichenkette vorne und hinten begrenzen:

```
>>> zitat = 'Hamlet sprach: "Sein oder nicht Sein. Das ist hier  
die Frage!"'  
>>> zitat  
'Hamlet sprach: "Sein oder nicht Sein. Das ist hier die  
Frage!"'
```

Wie Ihnen schon aufgefallen sein wird, setzt Python die Ausgabe des Variableninhalts automatisch in (einfache) Anführungszeichen, um deutlich zu machen, dass es sich hier um eine Zeichenkette handelt. Dass das ein praktisches Feature ist, wird an folgendem Beispiel deutlich:

```
>>> x = '5'  
>>> x  
'5'  
  
>>> x = 5  
>>> x  
5
```

Bei der ersten Zuweisung ist der Inhalt der Variablen eine Zeichenkette, im zweiten Fall tatsächlich eine Zahl, mit der man nun zum Beispiel rechnen könnte.

In Python ist es sehr einfach, Zeichenketten zu erzeugen, die über mehrere Zeilen gehen. Setzen Sie dazu den Text einfach in dreifache Anführungszeichen:

```
z = """Der Text beginnt auf der ersten Zeile  
und wird auf der zweiten Zeile fortgesetzt."""  
  
print(z)
```

Dieses Programm führt zum folgenden Output:

```
Der Text beginnt auf der ersten Zeile  
und wird auf der zweiten Zeile fortgesetzt.
```

Tatsächlich bleibt also der Zeilenumbruch in der Ausgabe erhalten. Dieses Feature können Sie nicht nur dann nutzen, wenn Sie im Skript-Modus arbeiten, also ein Programm schreiben, um es anschließend auszuführen. Auch im interaktiven Modus erkennt Python nach dem Drücken der <ENTER>-Taste, dass Sie hier einen mehrzeiligen String begonnen haben, wartet deshalb mit der Ausführung der Anweisung (die ja normalerweise mit <ENTER> ausgelöst wird) und erlaubt Ihnen stattdessen, auf der nächsten Zeile weiterzuschreiben.

Diese Art von Strings haben Sie bereits im vorangegangenen Kapitel als *Docstrings* kennengelernt. Docstrings stehen als *Dokumentation* im Ihrem Programmcode, nicht aber mit der Absicht, weiterverarbeitet oder für den Endanwender des Programms auf dem Bildschirm ausgegeben zu werden.

Manchmal möchte man Zeichenketten im Programmcode umbrechen, ohne dass dieser Umbruch beim Ausgeben der Zeichenkette sichtbar sein soll; es geht einfach nur darum, dass der Programmcode übersichtlicher sein soll (denken Sie an das empfohlene Limit von 79 Zeichen pro Zeile aus ► Abschn. 20.1.1!). In dieser Situation können Sie mit Backslash (\) arbeiten:

```
meldung = 'Hallo ' \
          'Welt'
print(meldung)
```

Dieser Code erzeugt die Ausgabe:

```
Hallo Welt
```

Es handelt sich also nicht um zwei Zeichenketten, die in zwei unterschiedlichen Zeilen stehen, sondern letztlich nur um eine Zeichenkette, die lediglich aus praktischen Gründen im Code auf zwei Zeilen verteilt ist.

### 21.3.3 Wahrheitswerte (**bool**)

Wahrheitswerte, also die logischen Werte Wahr und Falsch, werden in Python mit dem Datentyp **bool** abgebildet, eine Reverenz an den bereits im ersten Teil des Buches erwähnten englischen Mathematiker und Logiker George Boole, der im 19. Jahrhundert einen wesentlichen Beitrag zur Entwicklung der formalen Logik leistete.

Anders als die anderen Variablen-Typen können Variablen vom Typ **bool** lediglich zwei Ausprägungen annehmen: **True** (wahr) und **False** (falsch). Achten Sie auf

### 21.3 · Grundtypen von Variablen

die Groß- und Kleinschreibung! Die Konstanten **True** und **False** müssen jeweils mit großem Anfangsbuchstaben geschrieben werden. Würden wir stattdessen etwa **false** schreiben, würde Python annehmen, wir wollten auf eine Variable namens **false** zurückgreifen, die es aber natürlich nicht gibt:

```
>>> x = false
Traceback (most recent call last):
  File "<input>", line 1, in <module>
NameError: name 'false' is not defined

>>> x = False
>>> x
False
```

Auch dürfen keine Anführungszeichen verwendet werden, denn diese würden die Variable zu einer **str**-Variable machen:

```
>>> x = 'False'
>>> x
'False'

>>> type(x)
<class 'str'>

>>> x = False
>>> x
False

>>> type(x)
>>> x = 'False'
<class 'bool'>
```

Python speichert die Werte **True** und **False** intern als **0** und **1**. Deshalb können Sie mit ihnen auch rechnen, wie mit normalen Zahlen:

```
>>> x = 5 * True
>>> x
5
```

#### 21.3.4 None

Ein besonderer Datentyp ist **NoneType**. Von diesem Typ können Sie keine eigenen Variablen anlegen. Stattdessen hat Python für Sie bereits ein Objekt vom Typ **NoneType** erzeugt, nämlich **None**.

Das erlaubt es Ihnen, einer Variablen den Wert **None** zuzuweisen, was nichts anderes bedeutet, als dass diese Variable derzeit eben keinen echten, inhaltlich bedeutsamen Wert besitzt:

```
>>> x = None
>>> x
None

>>> type(x)
<class 'NoneType'>
```

Aber ist das nicht etwas umständlich? Könnten wir nicht einfach der Variablen auch den Wert **0** zuweisen, falls die Variable eine Zahl ist, bzw. "", also einen leeren String, falls es sich um eine Zeichenkette handelt? Das könnte man natürlich tatsächlich tun, aber nur dann, wenn die Werte **0** bzw. "" keine inhaltliche Bedeutung haben. Misst man aber zum Beispiel Temperaturen, oder erhebt im Rahmen einer Umfrage die Einstellung einer Person zu einem Thema auf einer Skala von -5 bis +5, kann der Wert 0 eben doch eine eigene, echte Bedeutung haben. Es ist dann sehr wohl ein Unterschied, ob der Befragte die Ausprägung 0 angegeben und damit eine neutrale Haltung zu dem Thema signalisiert, oder aber die Frage gar nicht beantwortet hat (**None**). Um diese Unterscheidung zu realisieren, macht es Sinn, für „kein echter Wert vorhanden“ einen speziellen Anzeiger zu haben, und genau das ist der **None**-Wert.

Rechnen können Sie mit **None** übrigens nicht:

```
>>> None + 1
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'NoneType' and
'int'
```

Wird **None** als logischer Ausdruck ausgewertet, wird es wie **False** behandelt. **None** taugt also wirklich zu nichts anderem, als anzusehen, dass eine Variablen keinen echten Wert beinhaltet.

### 21.3.5 Weitere Datentypen

Neben den bisher besprochenen Datentypen kennt Python von Haus aus noch eine Reihe weiterer Datentypen, wie zum Beispiel **complex**, ein Datentyp, der zur Abbildung der aus der Mathematik bekannten komplexen Zahlen dient, die aus einem Real- und einem Imaginärteil bestehen.

Verschiedene, nicht zum Standardsprachumfang gehörende Packages (Programmbibliotheken) bringen darüber hinaus eigene Datentypen mit.

Ein Beispiel hierfür ist das Package *NumPy*, eine Bibliothek für die effiziente Arbeit mit Vektoren und Matrizen, eine wichtige Grundlage für die Arbeit mit

statistischen und Methoden des maschinellen Lernens, Felder, in denen Python erhebliche Verbreitung gefunden hat.

Aber nicht nur diese komplexen Datentypen bringt *NumPy* mit, auch eine Reihe von grundlegenden Datentypen steht mit *NumPy* zur Verfügung. Für die bereits bekannte Datentypen **int** und **float** zum Beispiel verfügt *NumPy* über eigene Alternativen, die sich dadurch auszeichnen, dass sie sich im ihrem Speicherbedarf nicht dem Variableninhalt anpassen, wie es die Standarddatentypen in Python tun, sondern immer eine feste Zahl von Bytes im Speicher belegen. Das erlaubt es, sehr schnell mit solchen Variablen, insbesondere mit großen Feldern solcher Variablen zu rechnen, und effizientes Rechnen ist gerade im Bereich der Verarbeitung großer Datenmengen, wie eben beim maschinellen Lernen, eine essentielle Fähigkeit.

Da wir aber für die meisten Anwendungsfälle mit den hier diskutierten Datentypen auskommen, werden wir es dabei belassen und uns jetzt im nächsten Abschnitt den Charakter der Variablen in Python noch etwas genauer ansehen.

## 21.4 Variablen als Objekte

### 21.4.1 Attribute und Methoden von Variablen

Python ist stark vom objektorientierten Programmier-Paradigma geprägt. Anders als in manchen anderen Programmiersprachen, die sich ebenfalls diesen Programmieransatz auf die Fahnen geschrieben haben, wie etwa C++, sind in Python selbst die einfachsten Variablen Objekte.

Eine Variable in Python ist letztlich also immer die konkrete Instanz einer *Klasse*, zum Beispiel der Klasse **float** für Fließkommazahlen. Die Klasse **float** besitzt eine ganze Reihe von Methoden und Attributen, auf die Sie über die konkrete Instanz der Klasse, also ihre Variable, zugreifen können. Sollten Ihnen die Begriffe, Klasse, Instanz und Objekt nicht mehr ganz geläufig sein, blättern Sie einfach einige Seiten zum ► Abschn. 11.7 zurück und frischen Sie Ihr Wissen nochmal kurz auf, bevor es weitergeht!

Wenn Sie mit *PyCharm* arbeiten und im Code-Editor den Namen einer Variablen, die Sie in Ihrem Programm bereits verwenden, eingeben, gefolgt von einem Punkt, klappt ein kleines Code-Vervollständigungsmenü auf, in dem Sie die Methoden und Eigenschaften sehen, die die Klasse Ihrer Variable mitbringt. Gleiches erreichen Sie, wenn Sie eine Variable, der Sie bereits über die Python-Konsole einen Wert zugewiesen haben, in die Konsole eingeben, wiederum gefolgt von einem Punkt.

Der Punkt ist in Python der Operator, der Ihnen den Zugriff auf die Methoden und Attribute eines Objekts erlaubt. Daher erwartet *PyCharm* bei Eingabe eines Punktes hinter dem Variablen-Namen, dass Sie eine Methode oder ein Attribut dieser Variable verwenden wollen, und zeigt Ihnen die entsprechende Liste an. Ein Beispiel dafür sehen Sie in □ Abb. 21.1.

The screenshot shows a code editor with the following code:

```

1 x = 2.7
2
3 x.
4     m as_integer_ratio(self)           float
5     m fromhex(cls, s)                float
6     m hex(self)                     float
7     m is_integer(self)              float
8     f __class__                      object
9     m __abs__(self)                  float
10    m __add__(self, x)                float
11    f __annotations__                object
12    m __bool__(self)                 float
13    m __delattr__(self, name)       object
14    f __dict__                      object
15    ...
16    ...
17

```

A completion menu is open at the dot after 'x.', listing various methods and attributes. The menu includes:

- `m as_integer_ratio(self)` float
- `m fromhex(cls, s)` float
- `m hex(self)` float
- `m is_integer(self)` float
- `f __class__` object
- `m __abs__(self)` float
- `m __add__(self, x)` float
- `f __annotations__` object
- `m __bool__(self)` float
- `m __delattr__(self, name)` object
- `f __dict__` object

At the bottom of the menu, a note says: "Press Strg+ to choose the selected (or first) suggestion and insert a dot afterwards ➤ π".

Abb. 21.1 Code-Vervollständigungsmenü für eine float-Variable, aufgerufen aus dem Code-Editor

Die mit einem kleinen „m“ versehenen Menüeinträge sind die Methoden, die die Klasse **float** dem Objekt zur Verfügung stellt, die mit kleinem „f“ versehenen Einträge die Attribute („f“ für engl. *field* = Feld).

Auch erkennt man im Code-Vervollständigungsmenü jeweils rechts, woher das Objekt die Methoden oder Attribute erhalten hat. Wie Sie sehen, kommen manche Methoden und Attribute direkt von der Klasse **float**, manche aber auch von der allgemeineren Klasse **object**, von der die Klasse **float** abgeleitet ist. Die Klasse **object** ist also gewissermaßen die Elternklasse für die Klasse **float** und vererbt ihr Methoden und Attribute.

Erzeugen Sie sich jetzt einmal in der Konsole eine **float**-Variable und rufen Sie dann für dieses Variablen-Objekt, die Methode **is\_integer()** auf:

```

>>> x = 5.3
>>> x.is_integer()
False

```

Die Methode überprüft, ob die Fließkomma-Zahl zugleich auch eine Ganzzahl ist, was in unserem Beispiel hier natürlich nicht der Fall ist.

Die Klassenmethode **is\_integer()** braucht keine Funktionsargumente, denn Sie bezieht sich automatisch auf das Objekt, für das wir sie aufrufen, hier also **x**. Obwohl der Funktion keine Argumente übergeben werden müssen, muss sie dennoch stets mit den (leeren) runden Klammern aufgerufen werden, die sie als Funktion kennzeichnen.

Die Klasse **float** hat aber nicht nur Methoden, sondern auch einige Attribute. Eines dieser Attribute ist **\_\_class\_\_**. Es repräsentiert die Klasse des Objekts:

## 21.4 · Variablen als Objekte

```
>>> x.__class__  
<class 'float'>
```

Alternativ können Sie den Objekt-Typ übrigens auch mit der Funktion **type(objekt)** ermitteln, der das Objekt als Argument übergeben wird:

```
>>> type(x)  
<class 'float'>
```

Mit Hilfe der Funktion **isinstance(Objekt, Klasse)**, die ebenso wie **type(Objekt)** zur Python-Standardbibliothek gehört, die Sie also ohne weitere Vorbereitungen direkt verwenden können, lässt sich ermitteln, ob eine Variable von einem bestimmten Typ ist; mit unserer objektorientierten Terminologie präziser formuliert: Ob eine Variable eine Instanz einer bestimmten Klasse ist. Dazu wird der Funktion die Variable und die Klasse, auf die geprüft werden soll, übergeben:

```
>>> isinstance(x, str)  
False  
  
>>> isinstance(x, float)  
True
```

Lassen Sie uns nochmal einen genaueren Blick auf die Methoden der Objekte richten, also die Funktionen, die die Klasse dem Objekt zur Verfügung stellt. Etwas weiter oben hatten wir ja bereits mit **is\_integer()** eine Methode der Klasse **float** kennengelernt, die das aktuelle Objekt, das Objekt also, dessen Methode wir aufrufen, daraufhin überprüft, ob es eine Ganzzahl ist. Wie Sie sich erinnern, müssen wir dieser speziellen Funktion die Variable, die wir prüfen wollen, gar nicht also Argument übergeben, weil die Methode ja bereits Bestandteil des Objekts ist und sie deshalb gewissermaßen weiß, mit welchem Objekt sie arbeiten soll.

Betrachten wir jetzt einmal String-Variablen:

```
text = 'Strings in Python haben viele interessante Methoden.'
```

Wenn Sie mit *PyCharm* arbeiten und im Skripteditor oder in der Python-Konsole jetzt **text**. (mit dem Punkt-Operator!) eingeben, öffnet sich das mittlerweile bekannte Kontextmenü mit den Attributen und Methoden, die die String-Klasse **str** Ihrem Objekt **text** zur Verfügung stellt. Sie sehen sofort, dass hier vor allem eine reichhaltige Auswahl an unterschiedlichen Methoden verfügbar ist (☞ Abb. 21.2).

Probieren wir einige davon einmal aus:

- **lower()** und **upper()** verwandeln den String in Klein- bzw. Großbuchstaben:

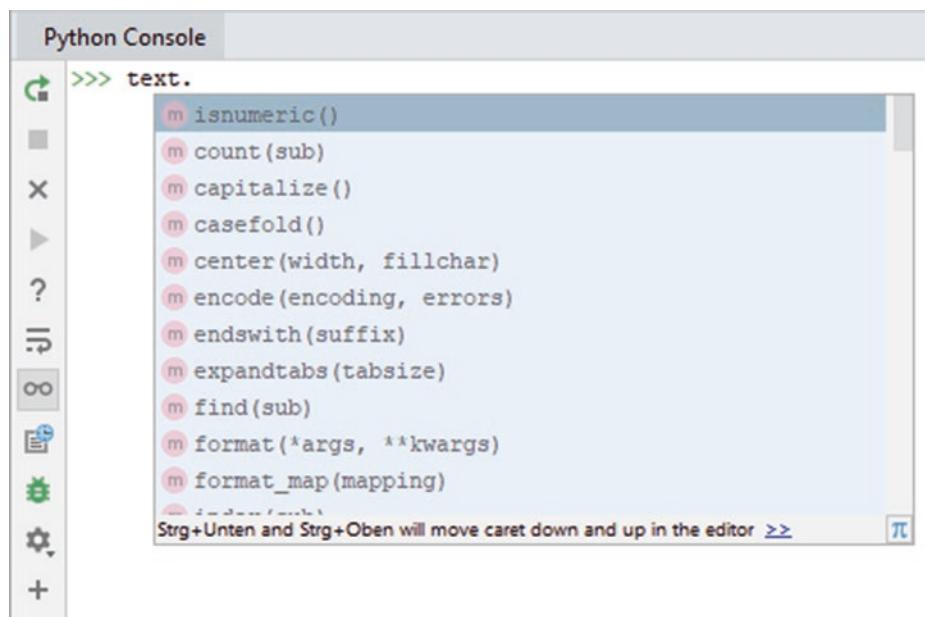


Abb. 21.2 Methoden eines str-Objekts

```
>>> text = 'Ein ganz normaler Text, mit einigen Wörtern und
Satzzeichen.'
>>> text.lower()
'ein ganz normaler text, mit einigen wörtern und satzzeichen.'

>>> text.upper()
'EIN GANZ NORMALER TEXT, MIT EINIGEN WÖRTERN UND SATZZEICHEN.'
```

- **isnumeric()** prüft, ob der String in eine Zahl verwandelt werden könnte:

```
>>> text.isnumeric()
False
```

- **count(substring)** zählt die Vorkommen des (Teil-)Strings **substring**, den man der Methode als Argument übergibt; es arbeitet dabei case-sensitive, unterscheidet also Groß- und Kleinschreibung, wie Sie am folgenden Beispiel sehen können (das „Ei“ von „Ein“ wird nicht mitgezählt):

```
>>> text.count('ei')
2
```

## 21.4 · Variablen als Objekte

- `replace(alt, neu, vorkommnisse)` ersetzt die angegebene Zahl von Vorkommnissen des alten Strings durch den neuen String; die Angabe der Anzahl der Vorkommnisse, die ersetzt werden sollen, ist optional, kann also auch weggelassen werden, was dazu führt, das einfach *alle* Vorkommnisse ersetzt werden:

```
>>> text.replace('Wörtern', 'Begriffen', 1)
'Ein ganz normaler Text, mit einigen Begriffen und Satzzeichen.'
```

- `__len__()` ermittelt die Länge der Zeichenkette:

```
>>> text.__len__()
60
```

Wie Sie an dem beiden führenden und abschließenden Unterstrichen sehen, handelt es sich hierbei um eine spezielle Kernmethode von Python.

### 21.1 [20 min]

Objekte vom Typ `str`, also Zeichenketten-Variablen, haben noch etliche interessante Methoden als die hier exemplarisch vorgestellten. Stellen Sie fest, welche Methoden Ihnen bei `str`-Objekten noch zur Verfügung stehen und probieren Sie sie in der Python-Konsole aus! Wenn Sie eine Methode nicht verstehen oder nicht erfolgreich einsetzen können, halten Sie sich nicht lange mir ihr auf und gehen Sie weiter zur nächsten Methode.

In der Hilfe können Sie sich über die Details der Methoden informieren. Rufen Sie dazu die Hilfe mit `help()` in der Python-Konsole auf und setzen Sie beim Aufruf `str`. vor die Methoden, damit Python weiß, für welche Klasse genau Sie sich die Methode anschauen wollen (es könnte ja mehrere Klassen geben, die eine Methoden diesen Namens besitzen), also beispielsweise `help(str.isnumeric)`.

Vielleicht ist Ihnen bei der Übung aufgefallen, dass Methoden wie etwa `upper()`, `lower()`, und `replace()` nicht das Objekt, für das sie aufgerufen werden, verändern, sondern lediglich eine *veränderte Kopie* des Objekts *zurückgeben*. Wollen Sie das Original-Objekt verändern möchten, müssen Sie ihm die veränderte Version, also den Rückgabewert der Methode zuweisen. Schauen wir uns das am Beispiel von `upper()` nochmal etwas genauer an:>>> text = 'Ein ganz normaler Text, mit einigen Wörtern und Satzzeichen.'

```
>>> text.upper()
'EIN GANZ NORMALER TEXT, MIT EINIGEN WÖRTERN UND SATZZEICHEN.'

>>> text
'Ein ganz normaler Text, mit einigen Wörtern und Satzzeichen.'
```

```
>>> text = text.upper()
>>> text
'EIN GANZ NORMALER TEXT, MIT EINIGEN WÖRTERN UND SATZZEICHEN.'
```

Wie Sie sehen, lässt der Aufruf der Methode `text.upper()` die Variable `text` vollkommen unverändert. Erst die Zuweisung des Rückgabewerts der Methode zu unserer Originalvariable verändert das Objekt `text`.

### 21.4.2 Erzeugen von Variablen mit der Konstruktor-Methode

Im vorangegangenen Abschnitt haben wir gesehen, dass sich Variablen einfach dadurch erzeugen lassen, dass man ihnen erstmalig einen Wert zuweist.

Es gibt aber noch eine andere Art, neue Variablen zu erzeugen. Wie Sie mittlerweile wissen, sind Variablen Objekte, also Instanzen einer Klasse. Wie alle Klassen haben diese Objekte eine Konstruktor-Methode, also eine spezielle Methode, die ein Objekt dieser Art erzeugt. Diese Konstruktoren können wir nutzen, um Variablen anzulegen. Betrachten wir dazu das folgende Beispiel:

```
>>> x = int(3)
>>> x
3

>>> type(x)
<class 'int'>
```

Die Konstruktor-Methode gibt also ein `int`-Objekt zurück mit dem Wert, der ihr als Argument übergeben wird. Das alleine ist vielleicht noch nicht so interessant, schließlich hätten wir denselben Effekt mit der schlichten Zuweisung `x = 3` auch einfacher erreichen können. Interessanter dagegen ist, dass wir dem Konstruktor auch eine Fließkommazahl oder eine Zeichenkette übergeben können und er uns daraus ein `int`-Objekt erzeugt. Im Falle der Fließkommazahl wird der Nachkommateil einfach ignoriert. Wird dem Konstruktor ein String übergeben, muss der Text natürlich auch in eine Zahl umwandelbar sein, ansonsten erhalten wir eine Fehlermeldung:

```
>>> x = int(3.7)
>>> x
3

>>> x = int('3.7')
>>> x
3

>>> x = int('abc')
```

```
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'abcd'
```

## 21.5 Konvertieren von Variablen

Nicht selten ist es nötig, den Datentyp von Variablen zu ändern. Betrachten Sie dazu das folgende Beispiel, indem wir eine Ganzzahl- und eine Fließkommazahlvariable addieren:

```
>>> x = 2
>>> type(x)
<class 'int'>

>>> x = x + 3.7
>>> x
3.7

>>> type(x)
<class 'float'>
```

Als wir die Variable erzeugten, wählte Python also als Datentyp automatisch **int**, weil wir der Variablen eine ganze Zahl zugewiesen hatten. Als wir dann aber **3.7**, also eine Fließkommazahl, hinzufügten, änderte Python den Typ zu **float**, um den neuen Wert aufnehmen zu können. Python konvertiert also *implizit*, ohne dass wir eingreifen müssen.

Probieren wir nun etwas anderes:

```
>>> x = 2
>>> x = x + '2.7'
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Addieren wir also zu unserer Variablen **x** einen String, konvertiert Python nicht mehr implizit. Vielleicht geht es anders herum, in dem wir **x** als String definieren und dazu eine Zahl addieren?

```
>>> x = '2'
>>> x = x + 3.7
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: can only concatenate str (not "float") to str
```

Auch das funktioniert nicht. Python konvertiert also zwischen Zahlen und Zeichenketten nicht implizit. Dennoch müssen wir manchmal Strings in Zahlen umwandeln, um mit ihnen rechnen zu können.

Schauen wir uns das an einem Beispiel an. Erinnern Sie sich noch an die Umrechnung von Kelvin in Celsius aus ► Abschn. 12.2.2? Dort hatten wir ein einfaches Programm besprochen, das als Benutzereingabe eine Temperatur in Kelvin entgegennimmt und sie in Grad Celsius umrechnet. Dieses Programm werden wir jetzt in Python entwickeln.

Dazu bedienen wir uns zur Eingabe der Funktion **input(eingabeaufforderung)**, die den Benutzer zu einer Eingabe auffordert, und diese Eingabe in Form eines Strings zurückgibt. Mit diesem Wissen wäre es jetzt also naheliegend, einen Code wie den folgenden zu schreiben:

```
temp_kelvin = input('Bitte geben Sie eine Temperatur \
in Kelvin ein: ')
temp_celsius = temp_kelvin + 273.15
print(temp_kelvin, 'Kelvin sind', temp_celsius, 'Grad Celsius.')
```

Führen wir dieses Programm aus und geben eine Temperatur in Kelvin ein, erhalten wir eine Ausgabe wie die folgende:

```
Traceback (most recent call last):
  File "C:/Users/MeinUser/Python/var_examples.py", line 54, in
<module>
    temp_celsius = temp_kelvin + 273.15
TypeError: can only concatenate str (not "float") to str
```

Das Problem, das hier auftritt, verstehen Sie mittlerweile: Die Funktion **input()** liefert einen String zurück, mit dem aber nicht gerechnet werden kann, weil Python nicht in implizit in eine Zahl konvertiert.

Also müssen wir *explizit* konvertieren. Genau das machen wir in dieser veränderten Version des Beispiels:

```
temp_kelvin = input('Bitte geben Sie eine Temperatur in \
Kelvin ein: ')
temp_celsius = float(temp_kelvin) + 273.15
print(temp_kelvin, 'Kelvin sind', temp_celsius, 'Grad Celsius.')
```

Dieses Mal konvertieren wir die String-Variable **temp\_celsius** explizit in eine Fließkommazahl, und zwar mit Hilfe der Funktion **float()**. Sie erkennen sofort den Bezug zum letzten Abschnitt: **float()** ist natürlich auch hier die Konstruktormethode der Klasse **float**. Wenn wir die String-Variable **temp\_kelvin** zu einer Fließkommazahl konvertieren, tun wir also letztlich nichts anderes, als einfach ein

neues **float**-Objekt zu erzeugen, indem wir den Konstruktor dieser Klasse aufrufen. Das neue **float**-Objekt initialisieren wir sogleich mit einem Wert; dieser Wert darf auch ein String sein, der **float**-Konstruktor erzeugt daraus dann eine Fließkommazahl.

Die *explizite* Konversion in Python läuft also über die Klassen-Konstruktoren, denen als Argument nicht nur ein Wert von dem Typ, den der Klassen-Konstruktor erzeugt (in unserem Fall **float**) übergeben werden kann, sondern auch verschiedene andere Typen (in unserem Beispiel **str**). Allgemein hat die explizite Konversion damit die Form: **datentyp\_nach(wert\_von)**.

## ?

### 21.2 [10 min]

Schreiben Sie ein Programm, dass das Alter des Benutzers in Jahren entgegennimmt und die Zahl der Minuten ausgibt, die der Benutzer (mindestens) bereits erlebt hat.

## 21.6 Komplexe Datentypen

---

### 21.6.1 Listen

---

#### 21.6.1.1 Listen erzeugen und anzeigen

Anders als viele andere Programmiersprachen kennt Python das Konzept der Variablen-Felder (Arrays) als solches nicht. Stattdessen gibt es in Python aber einen allgemeineren Typ, von dem das Array letztlich ein Spezialfall ist: die Liste.

Listen sind *geordnete* Zusammenfassungen *beliebiger* Objekte. Deshalb kann auch ein Array als eine Liste begriffen werden, denn dieses ist nichts weiter als eine geordnete Zusammenfassung von Objekten desselben Typs.

Listen werden in Python mit eckigen Klammern erzeugt, wie Sie im folgenden Beispiel sehen:

```
>>> zahlen = [1, 2, 3, 4, 5, 6, 7]
>>> vornamen = ['Sophie', 'Thomas', 'Ulrike', 'Tobias',
   'Heike']
```

Durch Eingabe des Listennamens in die Konsole wird uns der Inhalt der Liste angezeigt. Eckige Klammern um die Elemente der Liste erinnern uns daran, dass es sich hier tatsächlich um eine Liste handelt:

```
>>> zahlen
[1, 2, 3, 4, 5, 6, 7]

>>> vornamen
['Sophie', 'Thomas', 'Ulrike', 'Tobias', 'Heike']
```

Wenn Sie nicht in der Konsole, sondern im Skript-Modus arbeiten, also ein ganzes Programm schreiben, dann benutzen Sie zur Ausgabe die bereits bekannte Funktion `print()`, die auch Listen verarbeiten kann:

```
print(zahlen)
print(voramen)
```

Alleine den Bezeichner der Liste zu schreiben, also etwa

```
zahlen
```

führt im Programm – anders als in der Python-Konsole – *nicht* zu einer Ausgabe, wie wir bereits in ► Abschn. 21.1 gesehen haben.

### 21.6.1.2 Einzelne Elemente einer Liste selektieren

Auf einzelne Elemente der Liste kann nun durch Indizierung, die wiederum mit eckigen Klammern erfolgt, zugegriffen werden:

```
>>> zahlen[3]
4

>>> voramen[2]
'Ulrike'
```

Die Indizierung beginnt in Python bei 0, das Element mit dem Index 1 ist also bereits das *zweite* Element in der Liste, weil das erste Element den Index 0 trägt.

Mit *negativen* Indizes können Sie von hinten selektieren. Wollten wir also den zweiten Namen von hinten aus der Liste herausgreifen, könnten wir schreiben:

```
>>> voramen[-2]
'Tobias'
```

Das letzte Element der Liste hat dabei den Index -1, nicht etwa -0, wie man vielleicht vermuten könnte.

Mit Hilfe des Doppelpunkt-Operators kann als Index auch ein Bereich angegeben werden. Wollten wir zum Beispiel den zweiten bis vierten Vornamen selektieren, könnten wir das folgendermaßen bewerkstelligen:

```
>>> vorname[2:5]
['Ulrike', 'Tobias', 'Heike']
```

Selektiert werden dabei die Elemente mit den Indizes 2, 3 und 4, also das dritte, vierte und fünfte Element der Liste. Beachten Sie bitte, dass das Element mit dem

Index 5, also das sechste Element, nicht zur Selektion gehört. Die rechte Grenze der Index-Angabe ist – anders als in anderen Sprachen, wie etwa R – nicht Bestandteil der selektierten Elemente.

Genauso wie die Originalliste ist die Selektion selbst wieder eine Liste, weil wir mehrere Elemente selektiert haben. Selektieren wir dagegen nur ein einziges Element, ist die Selektion selbst keine Liste mehr, sondern hat den Typ des jeweiligen Elements der Liste:

```
>>> type(vorname)
<class 'list'>

>>> sel = vorname[2:5]
>>> type(sel)
<class 'list'>

>>> sel = vorname[2]
>>> type(sel)
<class 'str'>
```

Bei der Indizierung mit einem Bereich von Indizes kann eine Seite auch offenbleiben. Bleibt die linke Seite offen, so wird vom Anfang der Liste selektiert, bleibt die rechte Seite offen, wird bis zum Ende der Liste selektiert:

```
>>> vorname = ['Sophie', 'Thomas', 'Ulrike', 'Tobias',
   'Heike']
>>> vorname[:3]
['Sophie', 'Thomas', 'Ulrike']

>>> vorname[3:]
['Tobias', 'Heike']
```

Sie könnten auch beide Bereichsgrenzen offen lassen (**vorname[:]**), dann würden einfach alle Element der Liste selektiert (und damit letztlich eine Kopie der Liste erzeugt).

Will man in einem Schritt mehrere, nicht zusammenhängende Elemente einer Liste selektieren, zum Beispiel das erste und dritte Element, so ist das in Python nicht ganz so einfach. Hier bietet sich der Einsatz sogenannter Listenkomprehensionsausdrücke an, mit denen wir uns später noch beschäftigen werden, wenn wir die Umsetzung von Schleifen in Python ansehen.

### 21.6.1.3 Listen bearbeiten

Sie können die Indizierung, die wir gerade kennengelernt haben, nicht nur dazu nutzen, Elemente aus einer Liste zur Anzeige oder Weiterverarbeitung zu selektieren, sondern auch, um die Elemente in der Liste direkt zu verändern. In diesem Abschnitt werden wir uns ansehen, wie Sie Elemente einer Liste verändern, einer Liste Elemente hinzufügen, Elemente aus einer Liste löschen, eine Liste sortieren und mehrere Listen zu einer neuen Liste verknüpfen können.

## ■ Listenelemente ändern

Beginnen wir damit, dass wir den Wert eines Listenelements verändern:

```
>>> vornamen = ['Sophie', 'Thomas', 'Ulrike', 'Tobias',
   'Heike']
>>> vornamen[1] = 'Tobias'
>>> vornamen
['Sophie', 'Tobias', 'Ulrike', 'Tobias', 'Heike']
```

Auch ganze Index-Bereiche können damit zugewiesen werden:

```
>>> vornamen = ['Sophie', 'Thomas', 'Ulrike', 'Tobias',
   'Heike']
>>> vornamen[3:5] = ['Sabine', 'Beverly']
>>> vornamen
['Sophie', 'Thomas', 'Ulrike', 'Sabine', 'Beverly']
```

Achten Sie in diesem Fall darauf, dass der Wert, den Sie zuweisen, wiederum eine Liste ist (also in eckigen Klammern steht und auch tatsächlich die Länge der zu ersetzenen Teilliste hat.

### ? 21.3 [5 min]

Was passiert, wenn das zugewiesene Objekt keine Liste ist oder aber nicht die Länge der ersetzenen Teilliste hat? Probieren Sie es aus und versuchen Sie, die Ergebnisse zu erklären!

Listen sind wie alle Variablen in Python Objekte, besitzen also entsprechende Attribute und Methoden.

## ■ Elemente an Listen anhängen

Mit Hilfe der Methode **append(objekt)** kann ein Objekt an die Liste *angehängt* werden. Dabei wird direkt die Listen-Instanz bearbeitet, für die die Methode **append()** aufgerufen wird:

```
>>> vornamen = ['Sophie', 'Thomas', 'Ulrike', 'Tobias',
   'Heike']
>>> vornamen.append('Sabine')
>>> vornamen
['Sophie', 'Thomas', 'Ulrike', 'Tobias', 'Heike', 'Sabine']
```

Wenn Sie *an einer beliebigen Stelle* der Liste ein Element hinzufügen wollen, benutzen Sie die Methode **insert(einfuegen\_vor\_elementindex, objekt)** und übergeben Ihr die Position, die das neue Element haben soll, sowie das Element selbst:

```
>>> vornamen = ['Sophie', 'Thomas', 'Ulrike', 'Tobias',
   'Heike']
>>> vornamen.insert(2, 'Sabine')
>>> vornamen
['Sophie', 'Thomas', 'Sabine', 'Ulrike', 'Tobias', 'Heike']
```

### ■ Elemente aus Listen löschen

Ähnlich einfach können Sie Elemente aus einer Liste *löschen*:

```
>>> vornamen = ['Sophie', 'Thomas', 'Ulrike', 'Tobias', 'Heike']
```

```
>>> vornamen.__delitem__(4)
>>> vornamen
['Sophie', 'Thomas', 'Ulrike', 'Tobias']

>>> del vornamen[2]
['Sophie', 'Thomas', 'Tobias']
```

Dazu stehen Ihnen die Klassenmethode **\_\_delitem\_\_**(*elementindex*) zur Verfügung, der Sie nur den Index des zu löschenenden Elements übergeben müssen wie auch der Operator **del**, der, weil er ein Operator ist, ohne Klammern verwendet wird! Ihnen rufen Sie zusammen mit dem zu löschenen Element auf.

Mit Hilfe des **del**-Operators können Sie sogar mehrere Elemente auf einmal löschen, zum Beispiel durch den Aufruf **del vornamen[2:4]**.

Wie Sie am Beispiel des Löschens sehen, können manche Operationen sowohl mit einer Klassenmethode als auch mit einem Operator umgesetzt werden. Das gilt auch für die Selektionen, die wir im vorangegangenen Abschnitt betrachtet haben. Letztlich ist die Selektion **vornamen[2]** nichts weiter als der Aufruf **vornamen.\_\_getitem\_\_(3)** der Methode **\_\_getitem\_\_(elementindex)** und genau das ist es auch, was Python intern abarbeitet, wenn Sie Ihre Liste über die eckigen Klammern indizieren, die selbst nichts anderes als ein Operator sind.

### ■ Listen sortieren

Ihre Liste können Sie ganz einfach *sortieren*, in dem Sie mit den Klassenmethoden **sort()** und **reverse()** arbeiten, je nachdem, ob Sie aufsteigend oder absteigend sortieren wollen:

```
>>> vornamen = ['Sophie', 'Thomas', 'Ulrike', 'Tobias',
   'Heike']
>>> vornamen.sort()
>>> vornamen
['Heike', 'Sophie', 'Thomas', 'Tobias', 'Ulrike']
```

```
>>> vornamen.reverse()
>>> vornamen
['Ulrike', 'Tobias', 'Thomas', 'Sophie', 'Heike']
```

## ■ Länge von Listen ermitteln

Die *Länge einer Liste* ermitteln Sie ganz einfach mit der Methode `__len__()`.

```
>>> vornamen.__len__()
5
```

Anders als die zuletzt verwendeten Methoden der Klasse `list` verändert `__len__()` die Liste natürlich nicht, sondern gibt lediglich die Länge der Liste zurück.

Wollen Sie zwei *Listen miteinander verknüpfen*, verwenden Sie den Plus-Operator (`+`).

```
>>> vornamen + zahlen
['Ulrike', 'Tobias', 'Thomas', 'Sophie', 'Heike', 1, 2, 3, 4,
 5, 6, 7]
```

Hier sehen Sie, dass wir als Ergebnis eine Liste erhalten, deren Elemente teilweise Strings sind, teilweise aber auch Zahlen. Listen können eben, anders als die spezielleren Arrays, die in vielen Programmiersprachen anzutreffen sind, durchaus unterschiedliche Arten von Elementen aufnehmen. Insbesondere können die Elemente von Listen selbst wieder Listen sein. Genau diese Situation schauen wir uns im folgenden Abschnitt noch etwas genauer an.

### 21.6.1.4 Listen als Elemente von Listen

Betrachten Sie die folgende Liste:

```
>>> liste_mit_listen = [1, 2, 3, ['a', 'b', 'c'], 4]
```

Sie enthält als viertes Element (also als Element mit dem Index 3) wiederum eine Liste. Das sehen wir auch sehr schnell, wenn wir das Element selektieren und genauer inspizieren:

```
>>> liste_mit_listen = [1, 2, 3, ['a', 'b', 'c'], 4]
>>> liste_mit_listen[3]
['a', 'b', 'c']

>>> type(liste_mit_listen[3])
<class 'list'>
```

Um nun auf ein Element unserer „Unterliste“ zuzugreifen, picken wir uns zunächst mit `liste_mit_listen[3]` das vierte Element heraus. Dieses ist nun wiederum eine Liste. Warum also sollten wir aus dieser (Unter-)Liste nicht wieder selektieren können, genau wie wir es mit der „Oberliste“ auch getan haben? Wollten wir also beispielsweise das dritte Element der Liste, also das `c`, herausgreifen, würden wir folgendermaßen „doppelt“ indizieren:

```
>>> liste_mit_listen[3][2]
'c'
```

Auf diese Weise kann man mit Listen auch mehrdimensionale Variablenfelder konstruieren. Angenommen, wir wollten ein rechteckiges Werte-Schema abbilden, dass so aussieht:

1	2	3
4	5	6
7	8	9

Dies lässt sich mit in einander verschachtelten Liste sehr leicht abbilden:

```
>>> drei_mal_drei = [[1,2,3], [4,5,6], [7,8,9]]
>>> drei_mal_drei
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Auf die Koordinaten Zeile 2, Spalte 1 (also der Zahl 4 in unserem Werteschema) können wir dann zugreifen, in dem wir unsere Liste doppelt indizieren (beachten Sie dabei, dass die Indizierung ja bei 0 beginnt!):

```
>>> drei_mal_drei[1][0]
4
```

Der erste Index ist dabei also stets der Zeilenindex, der zweite der Spaltenindex.

Obwohl sich also auch mehrdimensionale Felder sehr gut mit Listen darstellen lassen, verwendet man, zumindest wenn man mit sehr großen Feldern arbeitet und die Geschwindigkeit des Programms ein wichtiger Faktor ist, spezielle Datenstrukturen, wie sie im Modul *NumPy* enthalten sind. NumPy ist eine wichtige Zusatzbibliothek für alle, die im wissenschaftlichen Bereich oder im Data-Science-Umfeld unterwegs sind. Die Bibliothek bietet auch einen speziellen Array-Datentyp, der zwar nicht so flexibel ist wie Listen (weil er tatsächlich nur Elemente desselben Typs aufnimmt), dafür aber kompakter im Speicher und schneller im Zugriff ist. Für unsere Zwecke hier genügen allerdings die zum Standardsprachumfang von Python gehörenden Listen allemal.

## ? 21.4 [20 min]

Probieren Sie ein wenig die Arbeit mit Listen aus! Erzeugen Sie Listen, selektieren Sie Elemente daraus, fügen Sie neue Elemente hinzu, löschen Sie Elemente. Es ist wichtig, dass Sie ein Gefühl dafür entwickeln, wie man mit Listen arbeitet, weil Listen in der praktischen Arbeit mit Python eine große Rolle spielen.

### 21.6.1.5 Strings als Listen

Eine Besonderheit in Hinblick auf Listen sind Strings. Sie haben in Python *listenartige Eigenschaften*. Auf ihre einzelnen Elemente, die Zeichen, kann in Listennotation *lesend* zugegriffen werden:

```
>>> nachricht = 'Hallo Welt'  
>>> nachricht[1]  
'a'
```

Der Versuch, auf diese Weise ein Zeichen eines Strings zu *bearbeiten*, wie etwa mit `nachricht[1] = 'x'`, führt hingegen zur Fehlermeldung **'str' object does not support item assignment**.

### 21.6.2 Tupel

Tupel sind ein Datentyp, der Listen in vielerlei Hinsicht ähnlich ist. Genauso wie Listen, sind auch Tupel geordnete Zusammenstellungen mehrerer Objekte, die nicht unbedingt vom gleichen Typ sein müssen. Der wesentliche Unterschied zu den Listen besteht darin, dass Tupel *unveränderbar* sind. Schauen Sie sich das folgende Beispiel an, in dem wir ein Tupel aus drei Ganzzahlen erzeugen:

```
>>> zahlen = (27, 9, 51)  
>>> zahlen  
(27, 9, 51)  
  
>>> type(zahlen)  
<class 'tuple'>  
  
>>> zahlen[1]  
9  
  
>>> type(zahlen[1])  
<class 'int'>  
  
>>> zahlen[1]=36  
Traceback (most recent call last):  
  File "<input>", line 1, in <module>  
TypeError: 'tuple' object does not support item assignment
```

Beachten Sie, dass – anders als bei den Listen – die Elemente, mit denen das Tupel initialisiert wird, in *runden* Klammern zusammengefasst werden. Der Zugriff auf die Elemente erfolgt allerdings genau wie bei den Listen auch, indem nämlich der Index des betreffenden Elements in *eckigen* Klammern angegeben wird; und auch hier beginnt die Indizierung selbstverständlich – wie immer in Python – bei 0, so dass **zahlen[1]** das *zweite* Element des Tupels abfragt.

Im letzten Schritt versuchen wir, diesem zweiten Element des Tupels einen Wert zuzuweisen. Das schlägt fehl, weil Tupel eben ein unveränderbarer Datentyp sind, dessen Elemente nach Initialisierung nicht mehr ausgetauscht werden können. Auch können keine neuen Elemente mehr hinzugefügt werden. Das Tupel ist und bleibt so, wie es bei seiner Erzeugung angelegt worden ist.

Die Klammern können Sie beim Anlegen des Tupels übrigens auch weglassen. Das obige Tupel **zahlen** könnten wir also auch so herstellen:

```
>>> zahlen = 27, 9, 51  
>>> zahlen  
(27, 9, 51)
```

Vielleicht haben Sie sich schon die Frage gestellt, warum man überhaupt Tupel nutzen sollte, wenn doch Listen alles können, was Tupel auch können, aber oben-drein noch veränderbar sind.

Der Vorteil der Tupel liegt vor allem darin, dass sie von Python schneller verarbeitet werden können als Listen. Sie bieten sich außerdem immer dann an, wenn Sie sichergehen wollen, dass Ihre Daten vor Überschreiben geschützt sind. Sollten Sie das nämlich aus Versehen einmal in Ihrem Programm versuchen, erhalten Sie eine Fehlermeldung, wie Sie im Beispiel oben gesehen haben.

Selbst, wenn Sie Tupel gar nicht so oft bewusst einsetzen, greift Python im Hintergrund vielfach auf Tupel zurück. Ein Beispiel: Anders als in vielen anderen Sprachen können Sie in Python mehrere Zuweisungen von Variablen in nur einer Anweisung unterbringen, zum Beispiel so:

```
>>> a, b = 5, 3  
>>> a  
5  
  
>>> b  
3
```

Was hier intern passiert, ist dass Python zunächst ein Tupel (**5, 3**) erzeugt und dessen Elemente dann den beiden Variablen **a** und **b** zuweist.

Wir werden später noch sehen, wie man in Python – und auch das ist in vielen anderen Programmiersprachen nicht möglich – eine Funktion/Methode mehr als nur einen Wert zurückgegeben lassen kann. Auch dies geschieht „unter der Haube“ über Tupel.

### 21.6.3 Dictionaries

Ein weiterer komplexer Datentyp neben den Listen und Tupeln sind die sogenannten Dictionaries, also „Wörterbücher“. Der Begriff „Wörterbuch“ beschreibt die Funktionsweise dieser Datenstrukturen tatsächlich recht gut. Anders nämlich als bei den Listen, wo wir zum „Nachschlagen“ nach einem Wert den Index des betreffenden Elements in der Liste verwenden, wird hier ein *Schlüssel* verwendet.

Es handelt sich bei den Dictionaries also um assoziative Felder. Wenn Sie mit dem Konzept nicht mehr vertraut sind, blättern Sie am besten nochmal einige Seiten zurück zu ► Abschn. 11.6.

Erzeugen wir als Beispiel ein Dictionary, das zu jedem Vornamen (Schlüssel) das Alter einer Person speichert (Wert). Die verschiedenen Schlüssel-Wert-Pärchen werden in geschweifte Klammern geschrieben, wobei Schlüssel und Wert jeweils durch einen Doppelpunkt, die Pärchen selbst durch Kommata voneinander getrennt werden:

```
>>> d = {'Thomas': 30, 'Sophie': 19, 'Heike': 28}
>>> d
{'Thomas': 30, 'Sophie': 19, 'Heike': 28}
```

In unserem Beispiel sind die Schlüssel Strings, die Werte Zahlen. Das muss aber nicht so sein. Zahlen (zum Beispiel Artikelnummern) können selbst auch Schlüssel sein. Sogar Tupel können Schlüssel sein, Listen allerdings nicht, weil Schlüssel stets unveränderlich sein müssen (Sie erinnern sich, dass Tupel unveränderbar sind, Listen aber durchaus modifiziert werden können).

Als Werte eignen sich alle möglichen Objekt-Typen, unter anderem auch Listen oder wiederum Dictionaries selbst. Auf diese Weise ist es also auch möglich, ein verschachteltes Dictionary zu konstruieren. Das werden wir uns in einer Übungsaufgabe später etwas genauer anschauen.

Der Zugriff auf die einzelnen Elemente erfolgt nun, wie bei assoziativen Feldern üblich, mit Hilfe des Schlüssels:

```
>>> d['Thomas']
30
```

Beachten Sie, dass zwar bei der Erzeugung des Dictionaries die Schlüssel-Wert-Pärchen in geschweifte Klammern geschrieben werden, zum Zugriff auf einzelne Elemente des Feldes aber eckige Klammern verwendet werden.

Anders als Listen sind Dictionaries *ungeordnete* Elementesammlungen. Der Zugriff auf einzelne Elemente mit Hilfe eines numerischen Index, der die Position des Elements innerhalb des Dictionaries angibt, ist nicht möglich, weil in einer ungeordneten Datenstruktur Elemente keine natürliche Position haben, an der sie stehen und abgefragt werden könnten. Deshalb führt der Versuch, über einen numerischen Index auf ein Element zuzugreifen, zu einer Fehlermeldung:

```
>>> d[1]
Traceback (most recent call last):
  File "<input>", line 1, in <module>
KeyError: 1
```

Dieser **KeyError** sagt uns, dass ein Schlüssel **1** im Dictionary nicht vorkommt. Python interpretiert die **1** als einen Schlüssel, versucht dementsprechend, den dazu gehörenden Wert zu finden, muss aber feststellen, dass **1** nicht unter den im Dictionary verwendeten Schlüsseln auffindbar ist. Eine ähnliche Fehlermeldung würden wir erhalten, wenn wir einen Namen als Schlüssel verwenden, der im Dictionary nicht vorkommt, zum Beispiel mit **d['Jakob']**.

Einem Dictionary lassen sich leicht Elemente hinzufügen, indem man für das neue Element eine Zuweisung vornimmt, bei der der neue Schlüssel mit einem Wert verbunden wird:

```
>>> d['Ulrike'] = 36
>>> d
{'Thomas': 31, 'Sophie': 19, 'Heike': 28, 'Ulrike': 36}
```

Natürlich können wir in einer solchen Zuweisung auch einen bereits vorhandenen Schlüssel verwenden, wie im folgenden Beispiel:

```
>>> d['Sophie'] = 22
>>> d
{'Thomas': 31, 'Sophie': 22, 'Heike': 28, 'Ulrike': 36}
```

Da der Schlüssel letztlich der Identifikator ist, den wir zum Zugriff auf ein Element des Dictionaries verwenden, muss er eindeutig sein. Deshalb können wir nicht einfach ein neues Element hinzufügen, das einen Schlüssel besitzt, der bereits im Dictionary existiert, sondern ändern in diesem Fall stets das schon vorhandene Element.

Nicht nur das Hinzufügen, auch das Löschen von Elementen ist mit dem bereits von Listen bekannten **del**-Operator sehr einfach:

```
>>> del d['Thomas']
>>> d
{'Sophie': 22, 'Heike': 28, 'Ulrike': 36}
```

Alternativ wäre, wie bei Listen auch, ein Aufruf der Klassenmethode **\_\_delitem\_\_** möglich: **d.\_\_delitem\_\_('Thomas')**. Anders als bei den geordneten Listen, nimmt diese Methode hier natürlich statt eines numerischen Positionsindex den entsprechenden Schlüssel als Argument.

Manchmal ist man daran interessiert, zu prüfen, ob ein bestimmter Schlüssel im Dictionary vorkommt. Das lässt sich mit dem **in**-Operator einfach bewerkstelligen: Wendet man den **in**-Operator auf einen Schlüssel und ein Dictionary an, so resultiert ein logischer Wert, der anzeigt, ob der Schlüssel im Dictionary verwendet wird, oder nicht.

```
>>> 'Heike' in d
True
```

```
>>> 'Miriam' in d
False
```

Natürlich lassen sich Schlüssel und Werte auch aus dem Dictionary extrahieren. Dazu besitzt die Dictionary-Klasse zwei spezielle Methoden, **keys()** und **values()**. Die Rückgabewerte dieser Methoden sind etwas komplizierter beschaffen, wir werden uns an später Stelle noch genauer mit dieser Art von Objekten, sogenannten iterierbaren Objekten, beschäftigen. Um mit ihnen einfach umgehen zu können, wandeln wir sie mit Hilfe der Methode **list()**, also der Konstruktormethode der Klasse **list**, in eine Liste um:

```
>>> list(d.keys())
['Sophie', 'Heike', 'Ulrike']

>>> list(d.values())
[22, 28, 36]
```

Mit diesen Listen wiederum können wir natürlich alles machen, was uns Listen erlauben, zum Beispiel, auf ein bestimmtes Element zugreifen. Da die Liste der Schlüssel natürlich eine geordnete Datenstruktur ist, können wir auf deren Elemente wiederum mit einem numerischen Index zugreifen, also etwa auf das zweite Element mit dem Index 1 (zur Erinnerung: die Indizierung in Python beginnt bei 0 für das erste Element):

```
>>> list(d.keys())[1]
'Sophie'
```

Auch die vollständigen Elemente des Dictionaries können in eine Liste extrahiert werden:

```
>>> list(d.items())
[('Sophie', 22), ('Heike', 28), ('Ulrike', 36)]
```

Die Elemente der Liste sind nun die einzelnen Elemente des Dictionaries. Sie sind selbst Tupel aus Schlüssel und Wert:

```
>>> type(list(d.items())[0])
<class 'tuple'>
```

### 21.5 [10 min]

Erzeugen Sie ein Dictionary, dessen Schlüssel numerische Artikelnummern und dessen Werte wiederum Dictionaries sind, die für jede Artikelnummer Beschreibung, Hersteller und Preis des jeweiligen Produkts enthalten.

#### 21.6.4 Sets

Als letzten Datentyp wollen wir uns nun noch die sogenannten *Sets* anschauen. Sets und Dictionaries haben gemeinsam, dass beide *ungeordnete* Sammlungen von Objekten sind. Ähnlich wie die Schlüssel eines Dictionaries, die stets eindeutig sein müssen, kann auch jedes Element in einem Set nur genau einmal vorkommen.

Sets unterstützen Mengenoperationen, wie man sie aus der mathematischen Mengenlehre kennt, also beispielsweise die Bestimmung der Schnitt- oder der Vereinigungsmenge zweier Sets.

Um ein Set zu erzeugen, benutzen wir – ähnlich wie beim Dictionary – die geschweiften Klammern; dieses Mal enthalten die geschweiften Klammern aber keine Schlüssel-Wert-Pärchen, sondern einfach die einzelnen Elemente des Sets:

```
>>> freunde_thomas = {'Anna', 'Sophie', 'Peter', 'Mark'}
>>> freunde_julia = {'Peter', 'Sophie', 'Hellen', 'Mike',
'Fatih'}
```

Dass die Reihenfolge der Elemente keine Rolle spielt, können Sie sehr leicht überprüfen, in dem Sie, zwei Mengen, die dieselben Elemente, aber in unterschiedlicher Reihenfolge beinhalten, miteinander vergleichen:

```
>>> {'Sophie', 'Peter'} == {'Peter', 'Sophie'}
True
```

Dabei verwenden wir zum Vergleich das doppelte Gleichheitszeichen, das, wir später noch sehen werden, in Python der Operator für Vergleiche ist (ein einfaches Gleichheitszeichen würde Python als Versuch einer Zuweisung betrachten, die hier natürlich nicht funktioniert würden). Das Ergebnis des Vergleichs, **True**, bestätigt, dass die Reihenfolge der Elemente in den Sets keine Rolle spielt, die beiden Sets sind trotz unterschiedlicher Reihenfolgen ihrer ansonsten gleichen Elemente identisch.

Mit den oben definierten Sets könnten wir nun zum Beispiel prüfen, welche Elemente in beiden Sets vorkommen, welche Personen also Freunde sowohl von Julia als auch von Thomas sind:

```
>>> freunde_thomas.intersection(freunde_julia)
{'Sophie', 'Peter'}
```

Dazu verwenden wir hier die Klassenmethode **intersection(anderes\_set)**, die die Klasse **set** von Haus aus mitbringt. In diesem Fall wäre das Ergebnis, nämlich die Schnittmenge, die uns wiederum als **set** zurückgegeben wird (leicht erkennbar an den geschweiften Klammern), natürlich dasselbe, wenn wir statt **freunde\_thomas.intersection(freunde\_julia)** umgekehrt **freunde\_julia.intersection(freunde\_thomas)** aufgerufen hätten.

Analog hätten wir auch mit dem Schnittmengen-Operator **&** arbeiten können, den die Klasse **set** unterstützt:

```
>>> freunde_thomas & freunde_julia
{'Sophie', 'Peter'}
```

Dass der Schnittmengenoperator ein kaufmännisches Und-Zeichen ist, kommt nicht von ungefähr, suchen wir doch bei der Schnittmenge alle Elemente, die in der einen *und* der anderen Menge enthalten sind.

*Vereinigungsmengen*, also die Menge aller Elemente, die in einer oder in beiden Ausgangsmengen enthalten sind, lassen sich mit der Klassenmethode **union(anderes\_set)** oder dem Pipe-Operator **|** bestimmen:

```
>>> freunde_thomas.union(freunde_julia)
{'Fatih', 'Peter', 'Hellen', 'Mike', 'Anna', 'Mark', 'Sophie'}

>>> freunde_thomas | freunde_julia
{'Fatih', 'Peter', 'Hellen', 'Mike', 'Anna', 'Mark', 'Sophie'}
```

Wie Sie sehen, enthält die Vereinigungsmenge unserer beiden Sets die Namen **Sophie** und **Peter** nur einmal, obwohl sie sowohl in **freunde\_thomas** als auch in **freunde\_julia** vorkommen. Genau das aber ist das Wesen der Sets (wie auch der Mengen in der Mathematik): Alle Elemente einer Menge sind jeweils verschieden voneinander, kein Element kann mehr als einmal auftreten.

In ähnlicher Weise können wir prüfen, ob eine Menge eine *Teilmenge* einer anderen ist:

```
>>> freunde_thomas.issubset(freunde_julia)
False

>>> {'Mark', 'Anna'}.issubset(freunde_thomas)
True
```

Auch hier steht wiederum eine praktische Abkürzung in Form eines Operators zur Verfügung, nämlich des Kleiner-Gleich-Operators:

```
>>> {'Mark', 'Anna'} <= freunde_thomas  
True
```

### 21.6 [20 min]

Finden Sie (zum Beispiel mit der Hilfefunktion `help()`) heraus, welche weiteren interessanten Operationen mit Mengen möglich sind und probieren Sie diese aus. Verwenden Sie dazu die Beispielsets `freunde_thomas` und `freunde_julia` aus diesem Abschnitt oder erzeugen Sie eigene Sets.

## 21.7 Selbstdefinierte Klassen

### 21.7.1 Klassen definieren und verwenden

Alle Datentypen, die wir uns angesehen haben, waren Klassen, ganz gleich, ob es sich um Grundtypen wie `int` und `str`, oder um komplexere Typen wie Listen oder Dictionaries gehandelt hat.

Weil Python als Programmiersprache dem objektorientierten Paradigma folgt, können wir natürlich auch selbst Klassen definieren.

Erinnern Sie sich an das Beispiel der Klasse `Produkt` aus ► Abschn. 11.7.2, die alle wichtigen Basisinformationen zu einem Produkt erfasst? Diese Eigenschaften waren die Bezeichnung, eine detailliertere Beschreibung, die Artikelnummer, der Name des Herstellers und der Preis.

Eine solche Klasse können wir uns in Python sehr einfach bauen, und zwar mit Hilfe des Schlüsselwortes `class`:

```
class Produkt:  
    bezeichnung = ''  
    beschreibung = ''  
    artikelnummer = ''  
    hersteller = ''  
    preis = 0.0
```

Hinter dem Doppelpunkt beginnt der Code-Block (Achtung, Einrückung!) mit den Attributen der Klassen, denen wir jeweils einen initialen Wert zuweisen.

Das war es auch schon! Jetzt können wir unsere Klasse `Produkt` benutzen und eine Variable dieses Typs erzeugen:

```
gartenschaufel = Produkt()
```

**Produkt()** ist die Konstruktor-Methode unserer Klasse, die wir hier so einsetzen, wie wir es in ► Abschn. 21.4.2 bei den Python-Grunddatentypen auch getan haben. Zwar haben wir gar keinen eigenen Konstruktor definiert, aber unsere Klasse bekommt von Python einen Standardkonstruktor, der nicht anderes tut, als ein Objekt der Klasse anzulegen. Wir werden uns später, wenn wir uns Methoden/Funktionen etwas genauer ansehen, damit beschäftigen, wie wir unseren eigenen Konstruktor schreiben und es dem Benutzer unserer Klasse damit erlauben können, zum Beispiel die Werte bestimmter Attribute gleich bei der Erzeugung einer neuen Klasseninstanz nach eigenen Vorgaben zu belegen. Genau das hatten wir ja in ► Abschn. 21.4.2 auch getan, wenn wir etwa den Konstruktor der Klasse **int** mit **int(56)** aufgerufen und ihn so veranlasst haben, ein neues **int**-Objekt zu erzeugen, das den Ganzzahlwert 56 beinhaltet.

Nachdem wir das Objekt vom Typ **Produkt** nun erzeugt haben, können wir seine Eigenschaften beliebig anpassen:

```
gartenschaufel.bezeichnung = 'Gartenschaufel, Edelstahl'
gartenschaufel.preis = 10.99
```

Wenn Sie mit *PyCharm* arbeiten und **gartenschaufel**. eingeben (einschließlich des Punkt-Operators!), dann bekommen Sie in dem sich öffnenden Dropdown-Menü unter anderem die gerade von uns definierten Attribute aufgelistet, die **gartenschaufel**, als Instanz der Klasse **Produkt**, besitzt.

Dass diese Zuweisungen funktioniert haben, können Sie leicht überprüfen, indem Sie sich die Werte der Attribute anzeigen lassen:

```
print(gartenschaufel.bezeichnung)
print(gartenschaufel.preis)
```

Beachten Sie bitte, dass wir hier nun nicht mehr im interaktiven Modus von Python unterwegs sind (Sie erkennen das leicht an dem fehlendem Eingabe-Prompt **>>>** vor den Anweisungen), obwohl wir die Klassendefinition natürlich auch in die Python-Konsole hätten einfüttern können. Daher genügt zur Anzeige des Inhalts einer Variablen nicht mehr die einfache Eingabe ihres Bezeichners (tatsächlich hat diese überhaupt keinen Effekt). Stattdessen müssen wir die Ausgabe explizit durch Aufruf der Funktion **print()** veranlassen.

## 21.7.2 Klassen aus anderen Klassen ableiten

In ► Abschn. 11.7.3 hatten wir das Konzept der Vererbung kennengelernt, dass natürlich auch Python als objektorientierte Sprache anbietet. Wir hatten dort die Klasse **Buch** als abgeleitete Klasse der Klasse **Produkt** definiert, die mit Autor und Seitenzahl über zwei besondere Attribute verfügt, die die Basis-/Elternklasse **Produkt** von Haus aus nicht mitbringt.

## 21.7 · Selbstdefinierte Klassen

Um in Python eine Klasse von einer anderen abzuleiten, wird der Name der Basisklassen in der Klassendefinition in Klammern hinter den Namen der abgeleiteten Klasse gesetzt:

```
class Buch(Produkt):
    seitenzahl = 0
    autor = ''
```

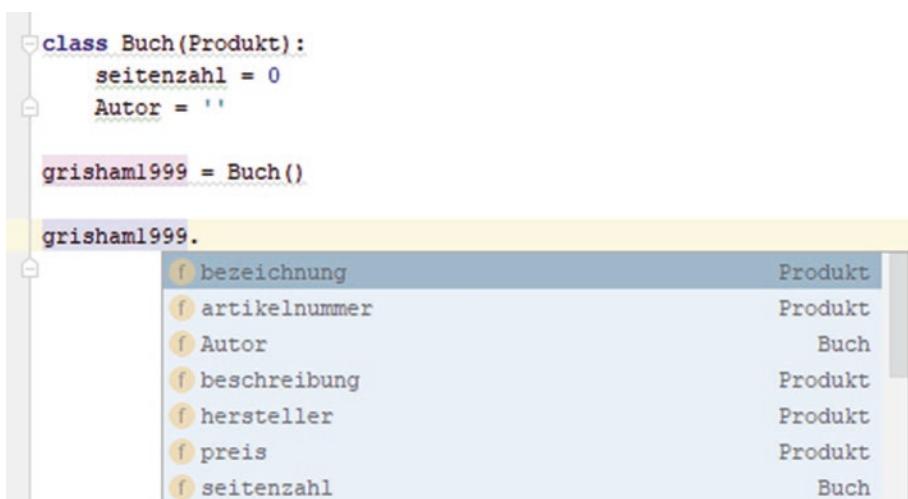
Nun können wir eine Instanz der Klasse **Buch** erzeugen, indem wir ihren Standard-Konstruktor, den Python freundlicherweise bereitstellt, aufrufen:

```
grisham1994 = Buch()
```

Wenn Sie sich nun wiederum in *PyCharm* die Attribute des neuen Objekts anzeigen lassen, indem Sie **grisham1994**. eingeben, so erkennen Sie sofort, dass die Instanz der Klasse **Buch** tatsächlich nicht nur über ihre eigenen Attribute, nämlich **seitenzahl** und **autor** verfügt, sondern auch über die von **Produkt** geerbten Eigenschaften wie etwa **bezeichnung** und **preis**. *PyCharm* zeigt Ihnen, wie Sie an Abbildung □ Abb. 21.3 erkennen können, auch an, von welcher Klasse das jeweilige Attribut stammt.

Mit allen Eigenschaften können wir nun nach Belieben arbeiten:

```
grisham1992.bezeichnung = 'Die Akte'
grisham1992.preis = 8.99
grisham1992.autor = 'John Grisham'
grisham1992.seitenzahl = 478
```



□ Abb. 21.3 Eigene und geerbte Eigenschaften der Klasse „Buch“

Klassen können in Python nicht nur von einer, sondern auch von mehreren Basisklassen abgeleitet sein. Zum Beispiel könnte es eine weitere Klasse **Copyright** geben:

```
class Copyright:
    inhaber = ''
    jahr = 1900
```

Dann könnten wir unsere Klasse **Buch** von beiden Basisklassen, **Produkt** und **Copyright** zugleich ableiten:

```
class Buch(Produkt, Copyright):
    seitenzahl = 0
    autor = ''
```

Indem wir beide Basisklassen in der Definition unserer Klasse **Buch** angeben, erzeugen wir eine Klasse, die die Attribute und Methoden beider Klassen erbt.

Dementsprechend können wir nun auch mit den Attributen arbeiten, die die Klasse Copyright mitbringt:

```
grisham1994 = Buch()
grisham1994.inhaber = 'Heyne Verlag'
grisham1994.jahr = 1994
```

### 21.7 [10 min]

Definieren Sie eine Klasse **Kunde** mit sinnvollen Kundenattributen und leiten Sie aus dieser Klasse eine Klasse **Geschäftskunde** ab, die zusätzliche Attribute beinhaltet, die ausschließlich für Geschäftskunden relevant sind.

### 21.7.3 Doppeldeutigkeit vermeiden: Name mangling

Was wäre aber, wenn nicht nur die Klasse **Produkt**, sondern auch die Klasse **Copyright** ein Attribut namens **bezeichnung** besäße? Die abgeleitete Klasse **Buch** hätte trotzdem nur *ein* Attribut **bezeichnung**. Nur von welchem „Elternteil“ würde dieses Attribut nun stammen? Ist es die **bezeichnung** von **Produkt** oder die von **Copyright**? Die Antwort in diesem Fall wäre: die von **Produkt**, weil Python von links nach rechts vorgeht, also diejenige Klasse, von der „zuerst“ abgeleitet wird, als erstes nach dem Attribut-Namen durchsucht und nur dann in den weiteren Elternklassen sucht, wenn die erste Elternklasse kein Attribut dieses Namens besaß.

Um Namenskonfusionen gänzlich auszuschließen, gibt es die Möglichkeit, Attribute von Klassen mit einem doppelten Unterstrich vor dem Namen zu versehen. Dann sähe die Klassendefinition unserer **Copyright**-Klasse so aus:

```
class Copyright:  
    __inhaber = ''  
    __bezeichnung = ''  
    __jahr = 1900
```

Der Wirkung des doppelten Unterstrichs besteht darin, dass Python die Eigenschaft automatisch unter dem Bezeichner **\_klasse\_attribut** zugreifbar macht, in unserem Beispiel also **\_Copyright\_bezeichnung**:

```
grisham1994._Copyright__bezeichnung = 'Copyright'  
print(grisham1994._Copyright__bezeichnung)
```

Dieser Vorgang, der auch als *name mangling* (zu Deutsch ungefähr *Namensauswahl*) bezeichnet wird, erlaubt es uns, beim Zugriff auf Attribute, deren Namen in der Klassenhierarchie möglicherweise mehrfach vorkommen könnte, Zweideutigkeiten und damit Missverständnisse auszuschließen.

## 21.8 Zusammenfassung

In diesem Kapitel haben wir uns mit Variablen in Python beschäftigt, und wie man mit ihnen arbeitet. Auch haben wir kennengelernt, wie Klassen definiert werden und Objekte als Instanzen von Klassen erzeugt werden.

Folgende Punkte sollten Sie aus diesem Kapitel mitnehmen:

- Python kennt einfache Datentypen, vor allem **int** (Ganzzahlen), **float** (Fließkommazahlen), **str** (Strings/Zeichenketten) und **bool** (Wahrheitswerte) sowie komplexere Datentypen, vor allem **list** (Liste), **dictionary** (assoziatives Feld), **tuple** (Tupel) und **set** (Menge).
- Alle Datentypen sind Klassen, Variablen dementsprechend Objektinstanzen; sie besitzen Attribute (Eigenschaften) und Methoden, um die Objekte zu manipulieren und anderweitig mit ihnen zu arbeiten.
- Bei den Namen von Variablen wird, wie überall in Python, zwischen Groß- und Kleinschreibung unterschieden; die offizielle Empfehlung ist, Variablen kleinzuschreiben und mehrere Begriffe in Variablen-Namen mit einem Unterstrich zu trennen.
- Variablen müssen nicht deklariert werden.
- Variablen können entweder durch Zuweisung eines Objekts zu einem Variablen-Namen generiert werden (wobei Python den Typ automatisch ermittelt) oder mit Hilfe der Konstruktor-Methode der jeweiligen Datentyp-Klasse.

- Die Konstruktoren können häufig auch mit Objekten anderen Typs als Argument aufgerufen werden; auf diese Weise kann zwischen Datentypen explizit konvertiert werden.
- Python konvertiert implizit verhältnismäßig wenig, zum Beispiel aber zwischen **int** und **float**, wo erforderlich.
- Der Dezimalzeichen in **float**-Werten ist der Punkt.
- Strings können in einfachen und doppelten Anführungszeichen stehen.
- Um sich den Inhalt einer Variable anzuzeigen, kann deren Name in die Python-Konsole eingegeben werden; innerhalb eines Python-Programms muss die Ausgabe dagegen immer ausdrücklich herbeigeführt werden (vornehmlich mit Hilfe der Funktion **print()**).
- Die komplexen Datentypen in Python unterscheiden sich danach, ob sie veränderbar sind (veränderbar: **list**, **dictionary**, **set**; unveränderbar: **tuple**) und ob die Elemente darin geordnet oder ungeordnet abgelegt sind (geordnet: **list**, **tuple**; ungeordnet: **dictionary**, **set**).
- Die Elemente komplexer Datentypen (und im Fall der Dictionaries sowohl Schlüssel als auch Werte) können unterschiedlichen Typs sein und selbst sogar wiederum Objekte dieses oder eines anderen komplexen Datentyps sein; so kann es zum Beispiel eine Liste geben, die wiederum Listen als Elemente enthält oder ein Dictionary, dessen Schlüssel teilweise Tupel und dessen Werte Listen und andere Dictionaries sind).
- Die Elemente geordneter Datentypen (Listen, Tupel) lassen sich über Indizes (das heißt, Positionsnummern) ansprechen; das erste Element hat jeweils den Index **0**.
- Mit dem Doppelpunktorperator kann ein Index-Bereich angesprochen werden, **A:B** bedeutet dabei: alle Elemente zwischen den Bereichsgrenzen **A** (einschließlich) und **B-1** (einschließlich).
- Bereichsgrenzen können auch offenbleiben, was gleichbedeutend ist mit „vom Anfang“ (linke Grenze nicht angegeben) bzw. bis zum Ende (rechte Grenze nicht angegeben).
- Negative Indizes bedeuten: Indizierung von hinten statt von vorne.
- Python kennt keinen besonderen Datentyp für Arrays/Felder, sondern den Datentyp **list**, der beliebige Elemente in geordneter Form aufnimmt; ein Array ist damit ein Spezialfall einer Liste (nämlich eine, deren Elemente alle vom gleichen Typ sind).
- Auch Strings verhalten sich im lesenden Zugriff wie Listen, ihre einzelnen Zeichen können in der für Listen üblichen Notation angesprochen werden; ein Schreibzugriff auf die Zeichen ist jedoch auf diese Weise nicht möglich.
- Eigene Klassen können über das Schlüsselwort **class** definiert werden; Klassen können dabei von einer oder mehreren „Elternklassen“ abgeleitet sein, also deren Attribute und Methoden erben.
- Ein doppelter Unterstrich von einem Attribut oder einer Methode in einer Klassendefinition bedeutet, dass das Attribut/die Methode auch unter der Bezeichnung **\_klasse\_attribut** bzw. **\_klasse\_methode()** zugreifbar ist (sog. *name mangling*); auf diese Weise können durch Mehrfachvererbung entstehende Doppeldeutigkeiten bei Attribut-/Methoden-Bezeichnern vermieden werden.

## 21.8 · Zusammenfassung

Die folgende Tabelle gibt eine Übersicht über die wichtigsten Datentypen. Bei den einfachen Datentypen sehen Sie dabei jeweils die Erzeugung per Zuweisung und per Aufruf der Konstruktor-Methode der jeweiligen Klasse. Natürlich können auch die komplexen Datentypen **list**, **tuple**, **dictionary** und **set** per Konstruktor erzeugt werden, auf die Darstellung wurde hier aber der Übersichtlichkeit wegen verzichtet.

■ Übersicht über die wichtigsten Datentypen			
Datentyp	Speichert	Erzeugung einer Variablen	Zugriff auf Elemente
<b>Int</b>	Ganzzahlen	Zuweisung: <b>x = 5</b> Konstruktor: <b>x = int('5')</b>	-
<b>float</b>	Fließkommazahlen	Zuweisung: <b>x = 0.5</b> Konstruktor: <b>x = float('5')</b>	-
<b>Str</b>	Zeichenketten	Zuweisung: <b>x = 'Hallo Welt'</b> Konstruktor: <b>x = str(0.5)</b>	-
<b>Bool</b>	Wahrheitswerte	Zuweisung: <b>x = False</b> Konstruktor: <b>x = bool('True')</b>	-
<b>List</b>	Geordnete, veränderbare Zusammenstellung anderer Objekte (u.U. auch unterschiedlichen Typs)	Elemente kommasepariert in eckigen Klammern: <b>x = [ 'Ulrike', 'Matthias', 'Hellen', 'Jennifer' ]</b>	Index oder Indexbereich in eckigen Klammern: <b>x[1]</b> (liefert: 'Matthias') <b>x[2:4]</b> (liefert: [ 'Hellen', 'Jennifer' ])
<b>Tuple</b>	Geordnete, <i>unveränderbare</i> Zusammenstellung anderer Objekte (ggf. unterschiedlichen Typs)	Elemente kommasepariert in runden Klammern: <b>x = ('abc', 27.5)</b>	Index in runden Klammern: <b>x(1)</b> (liefert: 27.5)
<b>dictionary</b>	Ungeordnete Zusammenstellung von Schlüssel-Wert- Pärchen, Schlüssel und Werte können Objekte unterschiedlicher Typen sein; Schlüssel müssen eindeutig sein	Schlüssel-Wert-Pärchen kommasepariert in geschweiften Klammern, Schlüssel und Wert jeweils durch Doppelpunkt getrennt: <b>x = { 'Ulrike' : 27, 'Matthias' : 41 }</b>	Schlüssel in eckigen Klammern: <b>x = [ 'Ulrike' ]</b> (liefert: 27)

Datentyp	Speichert	Erzeugung einer Variablen	Zugriff auf Elemente
Set	Ungeordnete Zusammenstellung von Objekten (ggf. unterschiedlichen Typs), die eindeutig sein müssen (ein Objekt kann nur einmal in der Liste vorkommen)	Elemente kommasepariert in geschweiften Klammern: <code>x = {'Ulrike', 'Matthias'}</code>	Da keine natürliche Reihenfolge und keine Schlüssel für den Zugriff vorhanden sind, macht Selektion eines einzelnen Elements keinen Sinn (man müsste das Element ja bereits kennen, um es anzusprechen)

## 21.9 Lösungen zu den Aufgaben

---

### ■ Aufgabe 21.1

Einige Beispiele für weitere Methoden des `str`-Objekts sind:

- **`find(sub)`:** Sucht in der Zeichenkette des `str`-Objekts nach `sub` und liefert den Index des ersten Zeichens des gefundenen Substrings zurück, bzw. `-1`, falls der Substring nicht gefunden wurde. Wie immer wird dabei zwischen Groß- und Kleinschreibung unterschieden. Anwendungsbeispiele:

```
>>> x = 'Hallo Welt!'
>>> x.find('We')
6

>>> x.find('WE')
-1

>>> x.upper().find('WE')
6
```

Im letzten Beispiel wird der String zunächst in Großbuchstaben umgewandelt. `x.upper()` gibt den in Großbuchstaben umgewandelten String zurück. Dabei handelt es sich natürlich wieder um ein `str`-Objekt. Dieses wird sodann mit Hilfe seiner `find()`-Methode durchsucht. Dieses Mal ergibt sich dabei ein Treffer.

Mit weiteren Argumenten kann der Bereich, in dem in dem String gesucht werden soll, eingeschränkt werden; die Angabe `S.find(sub[, start[, end]]) -> int` in der Hilfe liest sich so, dass `start` und `end` optionale Argumente von `find()` sind, sie können, müssen aber nicht angegeben werden. Deshalb sind Sie jeweils in eckigen Klammern. Beachten Sie dabei die Feinheiten der Klammerfolge: Die eckigen Klammern um das Argument `end` sind in den eckigen Klammern, die das Argument `start` umschließen, enthalten. Das bedeutet: Nur wenn `start` angegeben worden ist, kann auch `end` angegeben werden, aber `start` kann durchaus auch ohne Angabe von `end` verwendet werden!

## 21.9 · Lösungen zu den Aufgaben

Hinter dem `->` finden Sie schließlich den Typ des Rückgabewerts der Funktion, in unserem Fall hier `int`, weil ja der Index als Zahl zurückgegeben wird.

- **capitalize()**: Versieht den String mit einem großen Anfangsbuchstaben (nur das erste Wort!) und wandelt alle anderen Zeichen in Kleinbuchstaben.  
Anwendungsbeispiel:

```
>>> x = 'hallo Welt'  
>>> x.capitalize()  
'Hallo welt'
```

- **is.lower(), is.upper()**: Prüfen, ob der String ausschließlich aus Klein- bzw. Großbuchstaben besteht. Anwendungsbeispiel:

```
>>> x.isupper()  
False  
  
>>> x = 'HALLO WELT'  
>>> x.isupper()  
True
```

- **center(breite, [fuellzeichen])**: Erzeugt einen String der Länge **breite**, in dessen Mitte die Zeichenkette des **str**-Objekts steht. Links und rechts davon wird mit dem Zeichen **fuellzeichen** „aufgefüllt“ (standardmäßig, wenn nicht angegeben: Leerzeichen). Anwendungsbeispiel:

```
>>> x = 'Hallo Welt'  
>>> x.center(50, '*')  
*****Hallo Welt*****
```

### ■ Aufgabe 21.2

Ein Programm, das das Alter des Benutzers in Jahren einliest und als Minuten ausgibt, könnte so aussehen:

```
alter = input('Bitte geben Sie Ihr Alter in Jahren an:')  
minuten = int(alter) * 365 * 24 * 60  
  
print('Mit', alter, 'Jahren haben Sie bereits mindestens',  
      minuten,  
      'Minuten erlebt.')
```

### ■ Aufgabe 21.3

Weisen wir zunächst den Listenelementen mit den Indizes 3 und 4 ein Objekt zu, das selbst gar keine Liste ist. Wir beginnen mit einer Integer-Zahl:

```
>>> vornamen = ['Sophie', 'Thomas', 'Ulrike', 'Tobias',
   'Heike']
>>> vornamen[3:5] = 23
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: can only assign an iterable
```

Wir erhalten eine Fehlermeldung. Python kann einer (Teil-)Liste nicht etwas zuweisen, was selbst gar keine Liste ist. Anders, als man es vielleicht auch erwarten könnte, ersetzt Python nicht einfach die Elemente 3 und 4, also '**Thomas**' und '**Heike**' (Indizierung startet bei 0!), durch die Zahl 23.

Das Bild ändert sich, wenn wir die Zahl 23 in eine Liste „verpacken“:

```
>>> vornamen = ['Sophie', 'Thomas', 'Ulrike', 'Tobias',
   'Heike']
>>> vornamen[3:5] = [23]
>>> vornamen
['Sophie', 'Thomas', 'Ulrike', 23]
```

Jetzt werden die beiden Elemente tatsächlich durch die „Liste“, die nur die Zahl 23 enthält, ersetzt. Da diese aber kürzer ist als die ersetzte Teil-Liste, *verkürzt* sich unsere Liste **vornamen** entsprechend.

Probieren wir nun noch etwas anderes aus. Dieses Mal ersetzen wir die Teil-Liste durch einen String:

```
>>> vornamen = ['Sophie', 'Thomas', 'Ulrike', 'Tobias',
   'Heike']
>>> vornamen[3:5] = 'Anna'
>>> vornamen
['Sophie', 'Thomas', 'Ulrike', 'A', 'n', 'n', 'a']
```

Anders als oben bei der Zuweisung **vornamen[3:5] = 23** erhalten wir dieses Mal keine Fehlermeldung. Aber es passiert etwas scheinbar Merkwürdiges: Die Teil-Liste **vornamen[3:5]** wird ersetzt durch die Buchstaben des Namens Anna, wobei jeder Buchstabe zu einem neuen Listenelement wird. Die Ursache liegt darin, dass Strings auch als Listen interpretiert werden können. Deshalb ist die Zuweisung **vornamen[3:5] = 'Anna'** letztlich eine Ersetzung durch eine Liste, nämlich durch die Liste **['A', 'n', 'n', 'a']**.

**■ Aufgabe 21.4**

*Keine Lösung.*

**■ Aufgabe 21.5**

Hier haben wir es nun mit einem verschachtelten Dictionary zu tun. Der besseren Übersicht wegen sehen Sie die Dictionary-Definition unten mit Zeilenumbrüchen (Sie erinnern sich an ► Abschn. 20.1.2, dass eine Anweisung innerhalb von geschweiften Klammern umgebrochen werden kann):

```
>>> d= { 12345:
...     {
...         'Beschreibung': 'Plastikgartenstuhl "Garten-
...                     freund"',
...         'Hersteller': 'Omas Gartenparadies GmbH',
...         'Preis': 10.99
...     },
...     56789:
...     {
...         'Beschreibung': 'Gartenschaufel, Edelstahl',
...         'Hersteller': 'Big G Gardening Tools Ltd.',
...         'Preis': 49.90
...     }
... }
```

Der Zugriff erfolgt dann über einen doppelten Schlüssel:

```
>>> d[12345]['Preis']
10.99
```

Der Ausdruck **d[12345]** gibt also ein Dictionary zurück, und aus diesem Dictionary wird über einen Schlüssel, der in diesem Dictionary vorkommt, dann ein Wert selektiert.

**■ Aufgabe 21.6**

Einige Beispiele für weitere **set**-Operationen:

- **difference(*anderes\_set*)**: Bildet die Differenzmenge zu einem anderen Set, also die Menge aller Elemente aus dem Set, für das die **difference()**-Methode aufgerufen wird, ohne die Elemente des anderen Sets. Anwendungsbeispiel:

```
>>> freunde_thomas = {'Anna', 'Sophie', 'Peter', 'Mark'}
>>> freunde_julia = {'Peter', 'Sophie', 'Hellen', 'Mike',
...                     'Fatih'}
>>> freunde_julia.difference(freunde_thomas)
{'Mike', 'Fatih', 'Hellen'}
```

- **remove(element)**: Löscht ein Element aus einem Set. Anwendungsbeispiel:

```
>>> freunde_julia.remove('Hellen')
>>> freunde_julia
{'Mike', 'Sophie', 'Fatih', 'Peter'}
```

- **isdisjoint(anderes\_set)**: Prüft, ob das Set, dessen **isdisjoint()**-Methode aufgerufen wird und das andere Set disjunkt sind, also keine gemeinsamen Elemente haben. Anwendungsbeispiel:

```
>>> freunde_thomas = {'Anna', 'Sophie', 'Peter', 'Mark'}
>>> freunde_julia = {'Peter', 'Sophie', 'Hellen', 'Mike',
'Fatih'}
>>> freunde_thomas.isdisjoint(freunde_julia)
False
```

### ■ Aufgabe 21.7

Die beiden Klassen **Kunde** und **BusinessKunde** könnten so aussehen:

```
class Kunde:
    vorname = ''
    nachname = ''
    strasse = ''
    hausnummer = ''
    plz = ''
    email = ''

class BusinessKunde(Kunde):
    firma = ''
    zahlungsziel_tage = 14
    ustid = ''
```



# Wie lasse ich Daten ein- und ausgeben?

## Inhaltsverzeichnis

- 22.1 Ein- und Ausgabe in der Konsole – 302**
- 22.2 Grafische Benutzeroberflächen mit tkinter – 304**
  - 22.2.1 Überblick – 304
  - 22.2.2 Hallo tkinter! – 305
  - 22.2.3 Grafische Bedienelemente (Widgets) – 307
  - 22.2.4 Anordnen der Bedienelemente (Geometry Managers) – 327
  - 22.2.5 Ereignisse – 333
  - 22.2.6 Beispiel: Taschenrechner-Applikation – 337
- 22.3 Arbeiten mit Dateien – 343**
- 22.4 Übung: Entwicklung eines einfachen Text-Editors – 346**
- 22.5 Zusammenfassung – 347**
- 22.6 Lösungen zu den Aufgaben – 348**

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann [https://doi.org/10.1007/978-3-658-29850-0\\_22](https://doi.org/10.1007/978-3-658-29850-0_22).

## Übersicht

Nachdem wir uns ausgiebig mit der Organisation von Daten im Programm, nämlich den verwendeten Variablen/Objekten, befasst haben, ist es nun an der Zeit, darüber zu sprechen, wie eigentlich Daten vom Benutzer entgegengenommen und wieder an ihn ausgegeben werden können. Dazu beschäftigen wir uns in diesem Kapitel zunächst mit dem einfachsten Weg der Ein- und Ausgabe, nämlich dem über die *Konsole*. Danach werden wir unseren Programmen mit *grafischen Benutzeroberflächen* (GUIs) ein deutlich ansprechenderes Äußeres verleihen. Dabei schauen wir uns die Arbeit mit GUIs nicht nur an einem vollständigen Anwendungsbeispiel genauer an, sondern Sie werden die Gelegenheit haben, im Rahmen einer Übung Ihre eigene erste GUI-Anwendung zu programmieren. Abschließend wenden wir uns der in der Praxis natürlich äußerst wichtigen Arbeit mit *Dateien* zu.

In diesem Kapitel werden Sie lernen:

- wie Sie Informationen in die Konsole ausgeben und in der Konsole vom Benutzer abfragen werden können
- wie Sie mit der Python-Bibliothek **tkinter** Ihr Programm mit einer grafischen Benutzeroberfläche ausstatten können
- welche Steuerelemente Ihnen dabei zur Verfügung stehen, wie diese konfiguriert und auf der Oberfläche platziert und angeordnet werden können
- wie Sie auf Ereignisse, die der Benutzer über die Oberfläche auslöst (zum Beispiel beim Klick auf einen Button), in Ihrem Programmcode reagieren können
- wie Sie Daten aus Dateien auslesen und in Dateien schreiben.

### 22.1 Ein- und Ausgabe in der Konsole

Bereits in den vorangegangenen Kapiteln haben wir die beiden wichtigsten Funktionen verwendet, die der Ein- und Ausgabe von Daten in der Python-Konsole dienen, **input()** und **print()**.

#### ■ Eingabe

**input(*eingabeaufforderung*)** zeigt eine Eingabeaufforderung **eingabeaufforderung** an und lässt den Benutzer über die Tastatur eine Eingabe machen, die er durch Drücken der Tasten <RETURN> oder <ENTER> abschließt. Die Eingabe liefert **input()** dann als Rückgabewert, und zwar stets als *String*. Das ist immer dann wichtig, wenn Sie eigentlich diese Eingabe von Zahlen erwarten, mit denen Sie später weiterrechnen wollen. In diesem Fall müssen Sie den Rückgabewert von **input()** zunächst explizit in eine Zahl umwandeln, wie wir es auch in Abschn. 21.5 getan haben.

#### ■ Ausgabe

Das wichtigste Hilfsmittel zu Ausgabe von Informationen ist die Funktion **print()**. Sie kann dazu verwendet werden, eines oder mehrere Objekte auszugeben. Wenn

## 22.1 · Ein- und Ausgabe in der Konsole

man mehr als nur ein Objekt ausgeben will, bestimmt das optionale String-Argument **sep** dabei, wie die einzelnen Objekte in der Ausgabe voneinander getrennt werden; standardmäßig geschieht das mit einem Leerzeichen. Das ebenfalls optionale Argument **end** steuert, was am Ende der Ausgabe stehen soll; sofern mit Hilfe von **end** nichts anderes angegeben ist, wird ein Zeilenumbruch ans Ende der Ausgabe gesetzt. Der Zeilenumbruch wird dabei mit Hilfe einer Escape-Sequenz, nämlich **\n** (für *new line*) dargestellt, die wir bereits in Abschn. 11.2.2 im Zusammenhang mit Strings kennengelernt haben. Diese Escape-Sequenzen können wir natürlich auch direkt in Strings einbauen, die wir ausgeben wollen: Betrachten Sie als Beispiel folgendes kleine Programm:

```
user = input('Benutzername: ')
pwd = input('Passwort: ')

print('Willkommen, ' user, '!\\nIhr Passwort lautet:', pwd)
```

Hier lesen wir vom Benutzer einen Benutzernamen und ein Passwort ein und geben dann insgesamt vier Objekte aus:

- den String **'Willkommen,'**
- die String-Variable **user**
- den String **'!\\n Ihr Passwort lautet:'** (Achtung: Dieser String enthält einen Zeilenumbruch hinter dem Ausrufezeichen!)
- die String-Variable **pwd**.

Ruft man das Programm nun auf und macht als Eingaben für den Benutzernamen und das Passwort **peter** und **889X!z5**, dann erhält man folgende Ausgabe:

```
Willkommen, peter !
Ihr Passwort lautet: 889X!z5
```

Etwas unschön ist dabei das Leerzeichen zwischen dem Benutzernamen **peter** und dem Ausrufezeichen. Es ist zurückzuführen auf den Standardwert des Separator-Arguments **sep**, der eben ein Leerzeichen ist. Aufgrund dessen werden die beiden auszugebenden Objekte, die Variable **user** und der mit dem Ausrufezeichen beginnende String durch ein Leerzeichen voneinander getrennt. Um solcherlei Probleme zu vermeiden, empfiehlt es sich, die Ausgabe von Leerzeichen selbst zu steuern und das Argument **sep** auf „leeren String“ zu setzen, sodass also gar kein Separator durch die **print**-Funktion selbst ausgegeben wird. Der Aufruf von **print()** könnte dann so lauten:

```
print('Willkommen, ' user, '!\\nIhr Passwort lautet: ', pwd, sep
= '')
```

Wenn Sie diesen mit dem obigen Aufruf vergleichen, sehen Sie, dass wir überall dort, wo ein Leerzeichen ausgegeben werden soll, eben eines eingefügt haben und dafür das **sep**-Argument „leeren“. Beachten Sie dabei bitte, dass das **sep**-Argument immer mit seinem Namen aufgerufen werden muss, da sonst die **print()**-Funktion nicht weiß, ob der letzte String noch zu den auszugebenden Objekten gehört oder aber eine spezielle Bedeutung hat, also bereits das *nächste Argument* der Funktion darstellt, wie in unserem Fall. Wir rufen **sep** also als sogenanntes *Schlüsselwortargument* auf. Mehr dazu in ► Abschn. 23.1.2.

Die Objekte, die mit **print()** ausgegeben werden, müssen natürlich keineswegs nur Zeichenketten sein. Tatsächlich können Sie mit **print()** praktisch beliebige Objekte anzeigen und dabei in ein- und demselben **print()**-Aufruf auch Objekte unterschiedlicher Typen, das heißt, unterschiedlicher Klassen unterbringen. Das gilt sogar für Klassen, die Sie selbst definiert haben. Aber wie kann das funktionieren? Woher weiß **print()** denn, wie es beispielsweise ein Objekt vom Typ **Produkt** aus Abschn. 21.7 darstellen soll? Die Antwort ist ganz einfach: Klassen in Python können eine spezielle Funktion **\_\_str\_\_()** besitzen. Sie liefert eine String-Darstellung des Objekts und wird von **print()** aufgerufen, wenn ein Objekt dieser Klasse angezeigt soll. Als Entwickler der Klasse, können Sie also selbst festlegen, wie Objekte Ihrer Klasse dargestellt werden sollen. Sie müssen lediglich eine **\_\_str\_\_()**-Methode definieren. Ein Beispiel hierfür werden wir uns in Abschn. 23.2 genauer ansehen.

### 22.1 [5min]

Geben Sie drei unterschiedliche Arten an, wie die drei String-Ausdrücke '**Erste Zeile**', '**Zweite Zeile**' und '**Dritte Zeile**' in drei aufeinanderfolgenden Zeilen angezeigt werden können.

## 22.2 Grafische Benutzeroberflächen mit tkinter

### 22.2.1 Überblick

Programme auf der Kommandozeile bzw. in der Python-Konsole sind nicht jedermanns Sache. Insbesondere, wenn Sie Software für nicht-technikaffine Endkunden entwickeln, kommen Sie natürlich an grafischen Benutzeroberflächen nicht vorbei.

Deshalb werden wir uns in diesem Abschnitt damit beschäftigen, wie man in Python mit sehr überschaubarem Aufwand grafische Benutzeroberflächen gestalten und mit Programmfunktionalität hinterlegen kann.

Am Beispiel eines grafischen Taschenrechners werden Sie sehen, dass man bereits mit recht wenigen Zeilen Programmcode ein nützliches, voll funktionsfähiges Programm mit einer attraktiven Oberfläche schreiben kann, das seinen Benutzer nicht zwingt, vor einer schwarzen Konsole sitzend stur dem vorgegebenen Programmablauf zu folgen.

Das Kapitel schließt mit einer Übung, bei der Sie selbst einen einfachen Texteditor entwickeln werden, der es erlaubt, Textdateien zu öffnen, zu bearbeiten und zu speichern.

Es gibt zahlreiche unterschiedliche Bibliotheken und Frameworks, um grafische Oberflächen zu entwickeln. Viele davon sind plattform-übergreifend, das heißt, die Programme, die Sie damit entwickeln, laufen auf unterschiedlichen Computer- (und manchmal auch Mobil-)Betriebssystemen.

Eine häufig verwendete Bibliothek, mit der auch wir arbeiten werden, ist **tkinter**. Sie gehört praktischerweise zum Standardlieferumfang von Python, sodass wir nichts zusätzlich installieren müssen.

**tkinter** basiert auf **Tk**, einer plattform-übergreifenden Bibliothek für grafische Benutzeroberflächen, die ursprünglich Anfang der Neunziger Jahres des letzten Jahrhunderts in einer Programmiersprache namens Tcl entwickelt wurde. Populär geworden ist Tcl vor allem durch eben diese Bibliothek **Tk**. Denn die ist nicht nur für Tcl selbst verfügbar, sondern mittlerweile für eine Vielzahl anderer Programmiersprachen, darunter Python.

Python bietet ein Package namens **tkinter**, das letztlich eine Art „Verbindung“ zu **Tk** darstellt: Um die **Tk**-Bibliothek, die ja in Tcl geschrieben ist, zu verwenden, ruft Python einen Tcl-Interpreter auf, der ebenfalls zur Standardinstallation von Python gehört. Im Endeffekt arbeiten Sie also indirekt mit einer anderen Programmiersprache, nämlich Tcl, müssen dazu aber weder die Tcl-Syntax verstehen, noch den Tcl-Interpreter selbst aufrufen; stattdessen können Sie wie gewohnt in Python arbeiten und die übliche Python-Syntax verwenden. Da, wo nötig, „übersetzt“ Python dann Ihre Anweisungen in Tcl-Code und ruft den Tcl-Interpreter auf.

Das Praktische an **Tk** ist, dass Sie, wenn Sie verstanden haben, wie diese Bibliothek funktioniert, auch in anderen Programmiersprachen, die **Tk** unterstützen – und davon gibt es einige – ohne große Umstellungsschwierigkeiten rasch eigene grafische Benutzeroberflächen entwickeln können.

Im nächsten Abschnitt werden wir ein einfaches Hallo-Welt-Programm schreiben, das ein Fenster auf dem Bildschirm öffnet. Im Anschluss werden wir uns etwas intensiver mit den unterschiedlichen grafischen Steuerelementen der Benutzeroberfläche, den sogenannten *Widgets*, wie etwa Buttons, Eingabefelder und Checkboxen beschäftigen. Danach fehlen uns letztlich nur noch zwei Komponenten: Wir müssen diese Elemente auf der Oberfläche so anordnen, wie wir das haben möchten, um die gewünschte Oberflächendarstellung tatsächlich zu realisieren. Und schließlich müssen wir die Bedienelemente noch mit dem dahinterstehenden Programmcode „verdrahten“, damit sie auch auf die Aktionen des Benutzers entsprechend reagieren.

Damit haben wir alles zusammen, um Python-Programme mit einer richtigen grafischen Oberfläche zu schreiben, wie etwa den Taschenrechner, den wir zum Abschluss unserer Einführung in **tkinter** entwickeln werden.

Aber fangen wir erst mal ganz klein an.

## 22.2.2 Hallo tkinter!

---

Legen Sie eine neue Python-Datei mit folgendem Programmcode an und führen Sie das Programm aus:

```
from tkinter import Tk

win = Tk()
win.title('Hallo-Welt-Programm')
win.geometry('900x500')
win.mainloop()
```

Es öffnet sich ein Fenster, das ungefähr so aussieht, wie in Abb. 22.1.

Das Fenster ist noch sehr leer, aber das wird sich in den folgenden Abschnitten rasch ändern.

Wenn Sie sich den Code genauer anschauen, werden Sie feststellen, dass dieser mit einer **import**-Anweisung beginnt. Diese ist notwendig, um das Package **tkinter** verfügbar zu machen. Über den genauen Aufbau der **import**-Anweisung sollten Sie sich an dieser Stelle noch keine Gedanken machen, denn mit dem Import aus Packages und Modulen beschäftigen wir uns in Abschn. 23.3 genauer. An dieser Stelle genügt es, zu wissen, dass die **import**-Anweisung die Klassen des Moduls **tkinter**, insbesondere die Klasse **Tk** für unser Programm verwendbar macht.

Von dieser Klasse **Tk** erzeugen wir in unserem Programm eine Instanz, nämlich das Objekt **win**, das Hauptfenster unserer Applikation. Mit den Methoden **title(titletext)** und **geometry(dimensionen)** setzen wir zwei wichtige Eigenschaften, den Titel des Fensters und dessen Größe in Pixeln. Danach zeigen wir mit der Methode **mainloop()** das Fenster auf dem Bildschirm an und starten die Ereignisverarbeitung; unser Programm kann nun auf Aktionen des Benutzers reagieren.

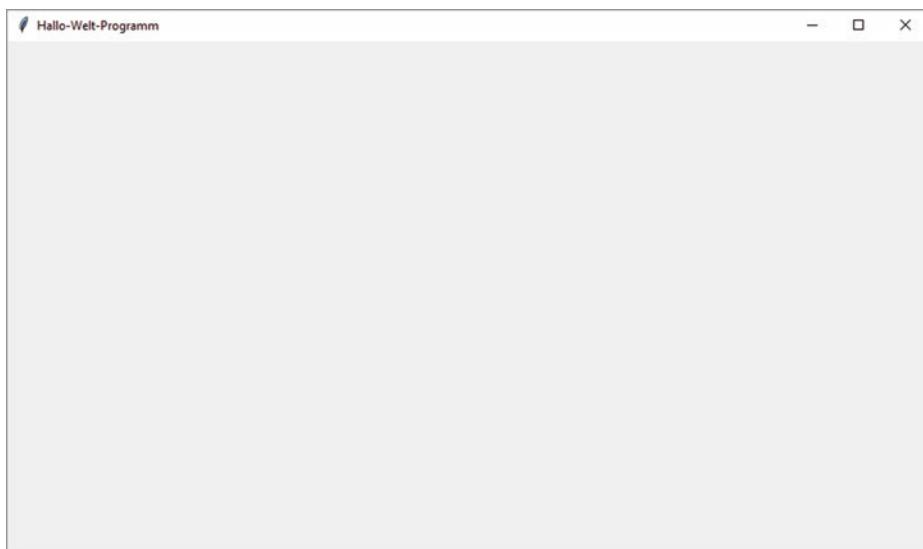


Abb. 22.1 Ein Hallo-Welt-Programm mit tkinter

Schon ist unser erstes **tkinter**-Programm fertig! Ganz einfach, oder?

Noch kann der Benutzer mit unserer grafischen Oberfläche aber wenig anfangen. Wir brauchen also Bedienelemente, die es dem Benutzer erlauben, Eingaben zu machen und Aktionen auszulösen. Diese Bedienelemente werden in **Tk/tkinter** als *Widgets* bezeichnet. Mit ihnen befassen wir uns im nächsten Abschnitt.

### 22.2.3 Grafische Bedienelemente (Widgets)

---

In diesem Abschnitt werden wir uns eine Reihe wichtiger Widgets anschauen. Am Beispiel des ersten Widgets, des Buttons, werden Sie sehen, wie Widgets erzeugt und ihre Eigenschaften bei der Erzeugung (oder auch später noch) angepasst werden.

#### 22.2.3.1 Schaltflächen (Button)

Schaltflächen/Buttons sind in der **tkinter**-Klasse **Button** abgebildet. Sie dienen dazu, den Benutzer des Programms Aktionen auslösen zu lassen.

Wie alle Widgets lassen sie sich mit ihrer Konstruktormethode leicht erzeugen. Das um diesen Konstruktoraufruf erweiterte Programm aus dem vorangegangenen Abschnitt sieht dann so aus:

```
from tkinter import Tk, Button

win = Tk()

win.title('Hallo-Welt-Programm')
win.geometry('900x500')

schalter = Button(win, text = 'Drück mich')
schalter.pack()

win.mainloop()
```

Die Konstruktormethode **Button()** bekommt als Funktionsargumente zum einen das Fenster übergeben, in dem wir den Button platzieren wollen, in unserem Fall also **win**. Zum anderen können Optionen zur Gestaltung des Buttons mit angegeben werden; in unserem Beispiel beschriften wir den Button mit Hilfe der Option **text**.

Natürlich können wir die Eigenschaften des Buttons auch noch ändern, *nachdem* wir den Button bereits erzeugt haben. Die Eigenschaften sind allerdings keine Klassen-Attribute der Klasse **Button**, sondern werden mit einer speziellen Methode namens **config()** verändert. Wie bereits der Konstruktor erwartet die Methode **config()** jeweils Schlüssel-Wert-Pärchen, bestehend aus dem Namen der anzupassenden Option als Schlüssel und dem Wert, der ihr zugewiesen werden soll.

Wollten wir also zum Beispiel die Breite unseres Buttons ändern, müssten wir die Option **width** folgendermaßen anpassen:

```
schalter.config(width = 50)
```

Genauso kann mit der Option **text**, also der Beschriftung des Buttons, verfahren werden.

Eine zweite Möglichkeit, Optionen zu verändern, besteht darin, auf die Optionen wie auf ein Dictionary zuzugreifen, dessen Schlüssel die Optionsnamen sind:

```
schalter['width'] = 50
```

Die Frage ist nun natürlich, welche Einstellungsmöglichkeiten es überhaupt gibt. Um das herauszufinden, sollten Sie als erstes einen Blick in die Hilfe der Klasse **tkinter** werfen.

Dazu müssen Sie zunächst die Klasse **Button** aus dem **tkinter**-Modul in die Konsole importieren (Sie erinnern sich: Ihre Python-Programme und die Konsole verwenden nicht denselben Namensraum; dass Sie bereits ein Python-Programm haben, welches das Modul importiert führt nicht dazu, dass Sie das Modul auch in der Konsole nutzen können. Vor dem Aufruf der Hilfe muss also in der Konsole noch eine **import**-Anweisung ausgeführt werden):

```
>>> from tkinter import Button
>>> help(Button)
```

Recht weit oben in dem dann erscheinenden Hilfetext steht folgende Information:

```
|      STANDARD OPTIONS
|
|          activebackground, activeforeground, anchor,
|          background, bitmap, borderwidth, cursor,
|          disabledforeground, font, foreground
|          highlightbackground, highlightcolor,
|          highlightthickness, image, justify,
|          padx, pady, relief, repeatdelay,
|          repeatinterval, takefocus, text,
|          textvariable, underline, wraplength
|
|      WIDGET-SPECIFIC OPTIONS
|
|          command, compound, default, height,
|          overrelief, state, width
```

Wie Sie sehen, gibt es also einerseits *Standardoptionen*, die den meisten Widgets gemein sind, andererseits spezielle Optionen, die nur für Buttons verfügbar sind. Leider fehlt in der Hilfe eine Beschreibung, welche Einstellung die jeweiligen Optionen steuern, und wie sie verwendet werden. Auch die „offizielle“ Dokumentation des **tkinter**-Packages (zum Zeitpunkt, zu dem diese Zeilen geschrieben werden: ► <https://docs.python.org/3/library/tk.html>) ist, insbesondere für Einsteiger, von eher überschaubarem Nutzen. Allerdings gibt es zahlreiche Seiten im Internet, auf denen die Eigenschaften verständlich erläutert werden, aktuell zum Beispiel ► [https://www.tutorialspoint.com/python/tk\\_button.htm](https://www.tutorialspoint.com/python/tk_button.htm).

■ Tab. 22.1 zeigt eine Übersicht über einige der Optionen, über die *alle*, oder doch zumindest die meisten Widgets verfügen. In dieser Tabelle werden zugleich im Zusammenhang mit der Codierung von Farben sowie der Arbeit mit Schriftformatierungen Vorgehensweisen erläutert, die an vielen Stellen in **tkinter** sinnvoll eingesetzt werden können.

■ Tab. 22.2 listet dann einige der *speziellen* Button-Optionen. In den folgenden Abschnitten zu den anderen Widgets finden Sie dann jeweils eine solche Tabelle mit den wichtigsten spezifischen Eigenschaften für genau dieses Widget.

■ **Tab. 22.1** Allgemeine Eigenschaften von tkinter-Widgets

Option	Typ	Bedeutung
<b>activebackground/activeforeground</b>	str	Farbe des Steuerelements (background) bzw. des Textes darauf (foreground), wenn das Steuerelement aktiviert wird (zum Beispiel beim Klicken auf den Button). <i>Wie alle Farben</i> in <b>tkinter</b> können Sie hier entweder eine der vielen vordefinierten Farbkonstanten angeben (zum Beispiel 'green' oder 'purple'; im Internet finden Sie schnell Listen dieser Farbkonstanten), oder aber einen Rot-Grün-Blau-(RGB-) codierten Wert der Form '#RRGGBB', wobei <b>RR</b> , <b>GG</b> und <b>BB</b> jeweils den hexadezimal (!) codierten Rot-, Grün- bzw. Blau-Anteil der Farbe darstellt. Verwenden Sie am besten einen der zahlreichen Umrechner im Internet, um die Dezimalwerte auf das hexadezimale System umzurechnen. Der Wert für Rot (R=255, G=0, B=0) würde damit zu '#FF0000', weil FF im hexadezimalen Zahlensystem die Zahl 255 darstellt.
<b>background/foreground</b>	str	Standard-Farbe des Steuerelements (background) bzw. des Textes darauf (foreground).
<b>border</b>	int	Stärke der Umrandung des Steuerelements in Pixeln ( <b>border=0</b> bedeutet keine Umrandung).
<b>cursor</b>	str	Form des Mausursors, wenn sich der Mauszeiger über dem Steuerelement befindet; Beispiele sind ' <b>hand2</b> ' (Hand), ' <b>watch</b> ' (Sanduhr), ' <b>cross</b> ' (Kreuz), ' <b>left_ptr</b> ' („normaler“ Mauszeiger mit Spitze links oben). Auch hierfür finden sich im Internet Listen, die die möglichen Zeiger-Ausprägungen aufführen.

(fortsetzung)

Tab. 22.1 (fortsetzung)

Option	Typ	Bedeutung
<b>font</b>	Font	<p>Die Schriftformatierung des Steuerelements; wollen Sie von der Standardschrift abweichen, müssen Sie eine zusätzliche Import-Anweisung einfügen und dann mit Hilfe des Konstruktors <b>Font()</b> ein neues Font-Objekt erzeugen:</p> <pre>from tkinter.font import Font</pre> <p><b>schrift = Font(family = 'Times', size = 36, weight = 'bold', underline = 1)</b></p> <p>Danach kann der <b>font</b>-Option des Steuerelements das neue <b>Font</b>-Objekt zugewiesen werden:</p> <pre>schalter['font'] = schrift</pre> <p>Die <b>family</b> ist dabei ein Schriftartenbezeichner (z.B. <b>Helvetica</b>, <b>Courier</b>). Beim <b>weight</b> wird zwischen '<b>bold</b>' (fett) und '<b>normal</b>' unterschieden. Darüber hinaus können Sie mit Hilfe der im obigen Beispiel nicht verwendeten Option <b>slant</b> den Text kursiv ('<b>italic</b>') oder nicht kursiv setzen ('<b>normal</b>'). <b>overstrike</b>, das ebenso wie das zur Unterstreichung verwendete <b>underline</b> die Werte <b>1</b> und <b>0</b> (oder <b>True</b> und <b>False</b>) annehmen kann, dient dazu, den Text durchzustreichen.</p> <p>Änderungen an Ihrem <b>Font</b>-Objekt können Sie, wie bei Steuerelementen auch, mit der Methode <b>config()</b> vornehmen:</p> <pre>schrift.config(weight = 'normal')</pre> <p>Änderungen, die Sie auf diese Weise vornehmen, wirken sich automatisch auf <i>alle</i> Steuerelemente aus, deren <b>font</b>-Option Sie das jetzt geänderte <b>Font</b>-Objekt ursprünglich zugewiesen hatten.</p>
<b>padx, pady</b>	int	Einrückung des Texts (bzw. eines Bildes) auf dem Steuerelement links/rechts ( <b>padx</b> ) bzw. oben/unten ( <b>pady</b> ).
<b>relief</b>	str	3D-Darstellung des Steuerelements. Mögliche Ausprägungen hier sind: ' <b>raised</b> ' (hervortretend, der Standardwert), ' <b>sunken</b> ' (vertieft), ' <b>flat</b> ' (flach), ' <b>groove</b> ' (vertiefte Umrandung) und ' <b>ridge</b> ' (einfacher Rand, ansonsten flach).
<b>text</b>	str	Beschriftung des Steuerelements.

Wir haben eben beim Ändern von Optionen gesehen, dass sich ein Widget teilweise wie ein Dictionary verhält. Deshalb können Sie auch mit der Methode **keys()** die Schlüssel und damit die Namen der Optionen auslesen. Wenn Sie in Ihren Programmcode nach der Erzeugung der Widget-Instanz die Anweisung

```
print(schalter.keys())
```

einbauen, werden Ihnen bei der Ausführung des Programms die Namen der verfügbaren Optionen *in der (Run-)Konsole* angezeigt. Wenn Sie die Methode **config()** ohne Argumente aufrufen, erhalten Sie das gesamte Dictionary mit allen Optionsnamen-Optionswert-Pärchen zurück. Auch dieses können Sie sich ausgeben lassen:

**Tab. 22.2** Spezielle Eigenschaften des Button-Widgets

Option	Typ	Bedeutung
<b>command</b>	<b>function</b>	Funktion, die ausgeführt wird, wenn der Benutzer auf den Button klickt. Dieser Ereignisbehandlung werden wir uns in ► Abschn. 22.2.5 genauer ansehen.
<b>Default</b>	<b>int</b>	Wenn <b>default</b> = 1, dann ist der Button der Default-Button (wird ausgelöst, wenn der Benutzer die <ENTER>-Taste drückt).
<b>height/width</b>	<b>int</b>	Höhe/Breite des Buttons. Angabe in Buchstaben, wenn Text auf dem Button dargestellt wird, in Pixeln, wenn ein Bild dargestellt wird. Wenn gar nicht angegeben, werden Breite und Höhe automatisch berechnet.
<b>state</b>	<b>str</b>	Button kann entweder auf ' <b>normal</b> ' (klickbar) oder ' <b>disabled</b> ' (ausgegraut) gesetzt werden. Wird der Button gerade geklickt, nimmt <b>state</b> den Wert ' <b>active</b> ' an.

```
optionen = schalter.config()
print(optionen)
```

Häufig können die kleingeschriebenen Zeichenketten-Optionswerte wie '**sunken**' für die Option **relief** auch mit Hilfe einer vordefinierten, dann jeweils großgeschriebenen Konstanten angesteuert werden, in unserem Beispiel also **SUNKEN**. Wir werden im Folgenden regelmäßig aber mit den Zeichenketten statt mit den Konstanten arbeiten. Wenn Sie mit den Konstanten arbeiten wollen, suchen Sie auf Ihrer Festplatte nach der Datei **constants.py**, die in Ihrem **tkinter**-Verzeichnis liegt. Dort sind die Konstanten definiert. Um diese nutzen zu können, fügen Sie Ihrem Programm folgende Importanweisung hinzu:

```
from tkinter.constants import *
```

Bisher haben wir noch gar nicht über den Aufruf der Methode **pack()** gesprochen, den wir in unser Programm hineingeschmuggelt haben. Dieser Anweisung macht den Button überhaupt erst auf dem Bildschirm sichtbar (kommentieren Sie die Zeile aus und schauen Sie, was passiert!). Das Sichtbarmachen hängt eng mit der Anordnung der Widgets auf der grafischen Oberfläche zusammen. Damit werden wir uns in ► Abschn. 22.2.4 genauer befassen. An dieser Stelle genügt es, dass wir **pack()** aufrufen, um unser Steuerelement auf der Oberfläche anzuzeigen.

### 22.2.3.2 Menüs (Menu)

Menüs lassen sich mit **tkinter** sehr einfach gestalten.

Dazu erzeugen wir zunächst für unser Fenster, im Beispiel von oben also dem Tk-Objekt **win**, eine neue Menüleiste, die in **tkinter** durch die Klasse **Menu** repräsentiert wird:

tiert ist. Dazu müssen wir zunächst (zusätzlich zu **Tk**) die Klasse **Menu** aus dem Modul **tkinter** importieren, indem wir die Importanweisung folgendermaßen erweitern:

```
from tkinter import Tk, Menu
```

Dem Aufruf des Konstruktors dieser Klasse wird dabei als Argument bereits das Fenster übergeben, zu dem die Menüleiste gehören soll. Umgekehrt weisen wir dem Fenster mit Hilfe der Option **menu** des **Tk**-Objekts die neue Menüleiste explizit zu, damit sie später auch tatsächlich sichtbar ist:

```
menuleiste_oben = Menu(win)
win.config(menu = menuleiste_oben)
```

Wie Sie bereits wissen, hätten wir, statt in der letzten Anweisung die Methode **config()** des **Tk**-Objekts aufzurufen, uns auch den praktischen Umstand zu Nutze machen können, dass **tkinter**-Widgets teilweise wie Dictionaries funktionieren und auf ihre Optionen deshalb auch zugegriffen werden kann, wie auf die Schlüssel-Wert-Pärchen eines Dictionaries. Dementsprechend hätten wir also auch schreiben können:

```
win['menu'] = menuleiste_oben
```

Damit besitzt Ihre Anwendung nun bereits eine Menüleiste, die aber noch nicht angezeigt wird, wenn Sie das Programm in diesem Zustand starten.

Zunächst müssen wir nämlich noch die einzelnen Pulldown-Menüs hinzufügen, die dann als Bestandteile der Menüleiste zu sehen und für den Benutzer auf- und zuklappbar sein werden. Auch diese Pulldown-Menüs wiederum sind Objekte der Klasse **Menu**, die wir durch Aufruf des Konstruktors erzeugen können. Dieses Mal allerdings hängen wir die neuen (Pulldown-)Menüs nicht direkt an das Fenster **win**, sondern an unsere bestehende Menüleiste **menuleiste\_oben**:

```
dateimenu = Menu(menuleiste_oben, tearoff = 0)
```

Die Option **tearoff=0** bewirkt, dass das Menü fest an das Fenster „angeschraubt“ wird. Lassen Sie diese Option weg oder stellen Sie sie auf **tearoff=1**, so erscheint in Ihrem Menü als erster Eintrag eine gestrichelte Linie. Klicken Sie auf diesen Eintrag, löst sich das Menü aus seiner Verankerung und wird in einem eigenen Fenster dargestellt, das beliebig auf dem Bildschirm bewegt werden kann (probieren Sie es aus!); ein Verhalten, das man normalerweise abschalten möchte.

Jetzt können wir damit beginnen, dem Menü neue Befehlseinträge mit **add\_command(label = *beschriftung*)** hinzuzufügen:

```
dateimenu.add_command(label = 'Öffnen...')  
dateimenu.add_command(label = 'Speichern')  
dateimenu.add_separator()  
dateimenu.add_command(label = 'Schließen')
```

Mit Hilfe der Funktion **add\_separator()** des **Menu**-Objekts erzeugen einen horizontalen Trennstrich im Menü, um unsere Menübefehle deutlicher zu strukturieren. Neben **add\_command()** und **add\_separator()** stehen mit **add\_checkbutton()** und **add\_radiobutton()** noch zwei weitere Arten von Menüeinträgen zur Verfügung. Beide erlauben es, den Benutzer eine Einstellung vorzunehmen zu lassen, wie im folgenden Beispiel. Die aktuelle Einstellung (ob die Menü-Option gerade angewählt ist oder nicht), lässt sich dabei jederzeit anhand der Variablen ablesen, die als Option **variable** der Funktion **add\_checkbutton()** übergeben wird.

```
dateimenu.add_checkbutton(label = 'Automatisch speichern',  
variable = auto_save)
```

Bis jetzt hat unser Menü – anders als die einzelnen Menüeinträge – noch gar keinen Anzeigenamen. Das ändern wir durch Aufruf der Funktion **add\_cascade()** des Menüleisten-Objekts, die außerdem dafür sorgt, dass das Menü tatsächlich auch als Pulldown-Menü auf unserer Menüleiste angezeigt wird:

```
Menuleiste_oben.add_cascade(label = 'Datei', menu = dateimenu)
```

Wenn Sie das Programm nun so starten, öffnet sich ein Programmfenster, das tatsächlich eine Menüleiste mit einem Menü „Datei“ aufweist. Analog könnten wir nun natürlich weitere Pulldown-Menüs auf der Menüleiste platzieren.

Die **add ...()**-Funktionen wie **add\_command()** können übrigens mit zahlreichen Optionen aufgerufen werden, von denen viele in □ Tab. 22.1 zu finden sind. So erzeugt beispielweise

```
dateimenu.add_command(label = 'Speichern', back-  
ground='#FF0000')
```

einen Menüeintrag „Speichern“ mit rotem Hintergrund, die Anweisung

```
dateimenu.add_command(label = 'Speichern', state='disabled')
```

dagegen einen deaktivierten (ausgegraute) Menüeintrag.

Natürlich können Sie Ihr Menü auch noch verändern, *nachdem* Sie es angelegt haben. Mit **delete(index\_von, index\_bis)** löschen Sie Menüeinträge anhand ihrer numerischen Indizes, mit **entryconfig(index, option=wert)** ändern Sie eine Option **option** (zum Beispiel **state** zum Aktivieren/Deaktivieren) für den durch **index** definierten Menüeintrag. Abfragen können Sie die Optionen mit **entrycget(index, option)**. Um sich also die Beschriftung (Option **label**) des zweiten Menüintrags anzeigen zu lassen, können Sie die Anweisung

```
print(dateimenu.entrycget(1, 'label'))
```

ausführen (die Indizes starten, wie immer in Python, bei 0!).

Wollen Sie nachträglich noch einen Menüeintrag einfügen, so benutzen Sie die Funktionen **insert(...(index, option))**, wobei ... für das jeweilige Element steht (also zum Beispiel **command**, **separator**, **checkbox**), ganz so wie bei **add...()**. Der Eintrag wird dann *hinter* dem durch **index** spezifizierten Eintrag eingefügt.

Unser Menü ist nun zwar recht hübsch anzusehen, hat aber keinerlei Funktionalität: Ein Klick auf einen Menübefehl löst keine Aktion aus. Das liegt daran, dass wir bei unseren Aufrufen von **add\_command()** des Pulldown-Menü-Objekts **dateimenu** auf die wichtige Option **command** verzichtet haben, die es erlaubt, eine Funktion anzugeben, die immer dann aufgerufen wird, wenn der Benutzer den Menüeintrag anklickt. In dieser Funktion würden wir dann denjenigen Teil des Programmcodes unterbringen, der hinter dem Menüeintrag stehen soll. Mit diesem sogenannten Event Handlern (Ereignis-Verarbeitern) werden wir uns in ► Abschn. 22.2.5 genauer beschäftigen, hier konzentrieren wir uns zunächst mal ganz und gar darauf, eine optisch ansprechende Oberfläche zu gestalten.

Der Umgang mit Menü in **tkinter** ist, wie Sie sehen, keineswegs kompliziert, erfordert aber mehrere unterschiedliche Arbeitsschritte. Deshalb hier noch einmal in der Übersicht das, was Sie tun müssen, um Ihr Programm mit einem Menü zu versorgen:

1. Erzeugen Sie die Menüleiste als solche mit Hilfe des Konstruktors **menuleiste = Menu(fenster)**.
2. Fügen Sie dem Fenster die Menüleiste hinzu: **fenster['menu'] = menuleiste**.
3. Erzeugen Sie ein neues Pulldown-Menü, dass Sie auf der Menüleiste platzieren wollen: **pd\_menu = Menu(menuleiste)**.
4. Fügen Sie dem neuen Pulldown-Menü Menüeinträge, also Befehle (mit **add\_command()**), Einstellungsoptionen (mit **add\_checkbox()** oder **add\_radiobutton()**) oder Trennstriche (**add\_separator()**) hinzu, zum Beispiel: **pd\_menu.add\_command(label = 'Titel des Befehls')**. Beachten Sie, dass wir später, um dem Menü Funktionalität zu verleihen, an dieser Stelle noch die Option **command = ereignis\_funktion** mit einbauen werden.
5. Hängen Sie Ihr neues Pulldown-Menü in die Menüzeile ein und vergeben Sie einen Titel, der in der Menüzeile zu sehen sein wird: **menuleiste.add\_cascade(label = 'Titel des Menüs', menu = pd\_menu)**.

Wir haben uns in diesem Abschnitt mit der Anlage eines Anwendungsmenüs beschäftigt, als mit dem Hauptmenü in Ihrem Programm. Mit dem **Menu**-Objekt lassen sich dank der Funktion **post(x, y)** Pop-up/Kontext-Menüs an beliebigen Stellen in Ihrem Fenster öffnen, was aber an dieser Stelle über den kurzen Einstieg in **tkinter** hinausgeht.

### 22.2.3.3 Eingabefelder (Entry und ScrolledText)

#### ■ Einzelige Eingabe: Entry

Eingabefelder dienen dazu, vom Benutzer Texteingaben entgegenzunehmen. In **tkinter** wird zwischen einzeiligen (Widget-Klasse **Entry**) und mehrzeiligen Texteingaben (Widget-Klasse **ScrolledText**) unterschieden. Beide schauen wir uns in diesem Abschnitt genauer an.

Das Widget vom Typ **Entry** lässt sich mit dem Konstruktor sehr einfach anlegen und dem Fenster, in unserem Beispiel also nach wie vor dem **Tk**-Objekt **win**, zuordnen.

```
eingabe = Entry(win)
eingabe.pack()
```

Wie schon beim Button zuvor, rufen wir auch dieses Mal wieder die Methode **pack()** auf, um das Steuerelement in unserem Fenster anzuzeigen. In ► Abschn. 22.2.4 beschäftigen wir uns dann eingehender damit, wie wir die Anordnung der Steuerelemente im Fenster beeinflussen können.

In □ Tab. 22.3 finden Sie eine Übersicht über die wichtigsten Eigenschaften des Widgets, auf die Sie in gewohnter Weise entweder durch die Dictionary-Indizierung **eingabe['option']** oder (dann allerdings nur schreibend) durch **eingabe.configure(option=wert)** zugreifen können.

□ Tab. 22.3 Spezielle Eigenschaften des Entry-Widgets

Option	Typ	Bedeutung
<b>justify</b>	<b>str</b>	Ausrichtung des Texts im <b>Entry</b> -Feld, ' <b>left</b> ' (linksbündig, der Standard), ' <b>center</b> ' (zentriert) oder ' <b>right</b> ' (rechtsbündig).
<b>selectbackground</b>	<b>str</b>	Hintergrundfarbe von Textmarkierungen.
<b>selectforeground</b>	<b>str</b>	Schriftfarbe von Textmarkierungen.
<b>show</b>	<b>str</b>	Zeichen, das (statt des eingegebenen Zeichens) angezeigt wird; kann zum Beispiel für Passworteingaben verwendet werden.
<b>width</b>	<b>int</b>	Breite des <b>Entry</b> -Feldes (in Zeichen, nicht in Pixeln).

Mit Hilfe der Methode **get()** können Sie jederzeit den Text abfragen, der aktuell im Entry-Feld vorhanden ist, beispielsweise:

```
mein_text = eingabe.get()
```

Natürlich können Sie den Text auch bearbeiten. Dies geschieht mit Hilfe der Methode **insert(index, string)**. **index** gibt dabei die Textposition (beginnend bei 0) an, an der Sie einfügen möchten, **string** den einzufügenden Text. Zu Beginn können Sie also mit

```
eingabe.insert(0, 'Ein erster Text im Eingabefeld.')
```

das Eingabefeld mit einem Text vorbelegen.

Statt eines echten Positionsindex kann in der Methode **insert()** auch einfach '**end**', '**insert**' oder '**anchor**' angegeben werden. Dann wird der Text am Ende ('**end**'), an der aktuellen Cursorposition ('**insert**'), oder am Beginn der aktuellen Markierung eingefügt (wenn derzeit eine Markierung besteht, ansonsten einfach am Anfang des Textes). Auf diese Weise können Sie mit der Anweisung

```
eingabe.insert('end', 'Ende des Textes.')
```

zum Beispiel Text am Ende anhängen.

In ähnlicher Weise können Sie Text löschen, indem Sie mit der Methode **delete(von\_index, bis\_index)** arbeiten. Lassen Sie dabei das optionale Argument **bis\_index** weg, wird nur das Zeichen mit dem Index **von\_index** entfernt.

Mit **icursor(index)** können Sie den Cursor an eine bestimmte Stelle im Text stellen, genauer gesagt, *hinter* das durch **index** angegebene Zeichen. Achtung aber: Die Zählung der Zeichen beginnt bei **tkinter** Python-untypisch bei 1. Geben sie als **index** die Zahl 0 an, stellen Sie den Cursor *vor* das erste Zeichen. Diese Zählweise gilt überall, wo Zeichenindizes zum Einsatz kommen.

Damit nun der Cursor im Text tatsächlich sichtbar ist, müssen Sie dem Steuer-element noch den Fokus geben, es also zum derzeit aktiven Steuerelement in Ihrem Fenster machen. Dies erreichen Sie, wie im übrigen bei allen anderen Widgets auch, mit der Methode **focus()**.

Die Methode **selection\_range(von\_index, bis\_index)** schließlich erlaubt es Ihnen, Text zu markieren. Auch hier können Sie wieder mit den von **insert()** bekannten speziellen Konstanten '**end**', '**insert**' und '**anchor**' arbeiten. Den derzeit markierten Text können sie mit **selection\_get()** abfragen, die Methode **selection\_present()** gibt **True** zurück, wenn ein Text markiert ist.

**Entry** erlaubt Ihnen auch, mit der Zwischenablage zu arbeiten. Der Zwischenablage können Sie Text mit Hilfe der Methode **clipboard\_append(text)** hinzufügen, den Inhalt der Zwischenablage mit **clipboard\_get()** abfragen und die Zwischenablage leeren mit **clipboard\_clear()**.

### ■ Mehrzeilige Eingabe: ScrolledText

Wenn längere Texte bearbeiten werden sollen, bietet sich das Widget **ScrolledText** an. Um es zu verwenden, muss die Klasse **ScrolledText** aus dem **scrolledtext**-Modul des **tkinter**-Packages zunächst importiert werden:

```
from tkinter.scrolledtext import ScrolledText
```

Das **ScrolledText**-Widget verhält sich in vielerlei Hinsicht genauso wie das **Entry**-Widget. Das meiste von dem, was Sie über **Entry** gelernt haben, können Sie also unmittelbar auf **ScrolledText** übertragen (Unterschiede bestehen v.a. darin, dass die von **Entry** bekannten **justify**- und **show**-Optionen, die Methode **icursor()** und die Einfügeposition '**anchor**' nicht zur Verfügung stehen).

Da **ScrolledText** eine mehrzeilige Eingabe erlaubt, ist es nur logisch, dass Textpositionen auch in gemäß einem Zeilen-/Spalten-Schema angesprochen werden können. Wollten wir etwa den gesamten Text ab dem fünften Zeichen in der zweiten Zeile löschen, ließe sich das mit folgendem Methodenaufruf bewerkstelligen, wenn das zuvor mit der Konstruktormethode **ScrolledText()** erzeugte Widget auf den Namen **st** hört:

```
st.delete(2.4, 'end')
```

Beachten Sie, dass dort, wo bei **Entry** noch ein einzelner Index stand, jetzt scheinbar eine gebrochene Zahl zu finden ist: **2.4**. Diese Zahl ist allerdings tatsächlich eine codierte Zeilen-/Spaltenangabe. Sie besagt: Zweite Zeile, fünftes Zeichen. Während also innerhalb einer Zeile die Zeichen beginnend von 0 gezählt werden (und 4 damit auf das fünfte Zeichen verweist), wie wir es bereits von **Entry** gewohnt sind, beginnt die Zeilenzählung in **tkinter** bei **1!** 2.4 bezieht sich damit auf die *zweite* Zeile, aber das *fünfte* Zeichen.

Das **ScrolledText**-Widget ist ungleich mächtiger als das **Entry**-Widget. Es erlaubt beispielsweise von Haus aus das Rückgängigmachen bzw. Wiederholen von Editieroperationen mit Hilfe der Methoden **edit\_undo()** und **edit\_redo()** (zuvor muss die Option **undo** des Widgets auf **True** gesetzt werden). Auch erlaubt **ScrolledText** Textbereichen Namen zuzuweisen (sogenannte *tags*) und die Textbereiche dann über die *tags* anzusprechen und zu bearbeiten. Auf diese Weise können Sie zum Beispiel unterschiedliche Textbereiche unterschiedlich einfärben. Wenn Sie also einen Texteditor mit Syntax-Highlighting entwickeln wollen, ist das **ScrolledText**-Widget kein schlechter Startpunkt.

#### 22.2.3.4 Textanzeigen (Label)

Labels sind statische Texte, die im Anwendungsfenster dargestellt werden. Sie bieten dem Benutzer normalerweise keine Interaktionsmöglichkeit, sondern dienen dazu, Informationen darzustellen oder die Funktion von Steuerelementen zu beschreiben, sofern diese keine eigene Beschriftung mitbringen (wie etwa Buttons).

Tab. 22.4 Spezielle Eigenschaften des Label-Widget

Option	Typ	Bedeutung
<b>anchor</b>	<b>str</b>	Position und Ausrichtung des Textes. Wird anhand der englischen Abkürzungen der Himmelsrichtungen beschrieben, zum Beispiel 'ne' für <i>north east</i> , das heißt, rechts oben; mögliche Ausprägungen sind demnach: 'n' (oben mittig), 'ne' (oben rechts), 'e' (rechts mittig), 'se' (unten rechts), 's' (unten), 'sw' (unten links), 'w' (links mittig), 'nw' (oben links) sowie zusätzlich ' <b>center</b> ' (horizontal und vertikal mittig).
<b>width</b>	<b>int</b>	Breite des Labels in Zeichen. Wenn nicht angegeben, wird das Label so breit gemacht, dass der dazustellende Text darauf passt.
<b>wraplength</b>	<b>int</b>	Zahl der Zeichen, nach der der Text auf dem Label umgebrochen werden soll. Wenn nicht angegeben, erfolgt kein Umbruch.

Erzeugen und anzeigen können Sie ein Label-Widget folgendermaßen:

```
beschriftung = Label(win, text = 'Hier ein fester Text')
beschriftung.pack()
```

- Tab. 22.4 zeigt einige wichtige Eigenschaften von Labels. Darüber hinaus stehen natürlich auch die von den meisten Widgets bereitgestellten Standardeigenschaften hinaus, die in Tab. 22.1 gelistet sind und deren bedeutsamste für Labels sicherlich die **text**-Option ist, also der Text, der auf dem Label angezeigt wird. Im Beispiel haben wir den Text direkt beim Aufruf des Konstruktors gesetzt, aber natürlich lässt sich dieser, wie alle anderen Eigenschaften auch, später noch mühelos mit **beschriftung.configure(text = neuer\_text)** oder mit **beschriftung['text'] = neuer\_text** ändern.

### 22.2.3.5 Checkbuttons und Radiobuttons

Checkbuttons (oder Checkboxen) und Radiobuttons dienen dazu, dem Benutzer eine Auswahl aus mehreren Einstellungen/Optionen zu geben. Während der Benutzer bei Checkboxen mehrere unterschiedliche Einstellungen/Optionen markieren kann, ist bei Radiobuttons immer nur eine Einstellung/Option auswählbar.

Checkbuttons und Radiobuttons werden in **tkinter** durch die gleichnamigen Klassen repräsentiert. Erzeugen wir zunächst zwei Radiobuttons:

```
auswahl_var = IntVar()

auswahl_rechts = Radiobutton(win, text = 'Rechts', variable =
auswahl_var, value = 1)
```

```
auswahl_links = Radiobutton(win, text = 'Links', variable =
auswahl_var, value = 2)

auswahl_rechts.pack()
auswahl_links.pack()
```

Betrachten Sie zunächst die Konstruktoraufrufe der Radiobuttons selbst: Hier übergeben wir wie üblich zunächst unser Fensterobjekt **win** (vom Typ **Tk**), zu dem der Radiobutton gehören soll, sowie einen Text, der als Auswahloption dargestellt wird. Darüber hinaus gibt es aber mit **variable** und **value** noch zwei weitere Argumente.

Dem Argument **variable** weisen wir eine Variable **auswahl\_var**, die wir zu Beginn des Codeabschnitts erzeugt haben, und zwar als Instanz der Klasse **IntVar**. **IntVar** ist eine von mehreren speziellen Variablenklassen, die **tkinter** mitbringt. Es handelt sich dabei nicht einfach um eine gewöhnliche **int**-Variable. Das Besondere an dieser Variablen, nachdem wir sie dem Argument **variable** des Radiobutton-Konstruktors zugewiesen haben, ist, dass sie stets den aktuellen Zustand unseres Radiobutton-Sets wiedergibt, also anzeigt, welche Radiobutton-Option gerade ausgewählt ist. Den Wert, den **auswahl\_var** annimmt, wenn ein bestimmter Radiobutton ausgewählt ist, definieren wir über das **value**-Argument von **Radiobutton()**. Klickt der Benutzer also den Radiobutton „Links“ an, besitzt **auswahl\_var** den Wert **2**, klickt er auf „Rechts“, nimmt **auswahl\_var** den Wert **1** an. Die **IntVar** **auswahl\_var** erfährt also automatisch ein Werte-Update, sobald sich beim Radiobutton an der Auswahlsituation etwas ändert.

Das ist sehr praktisch; allerdings muss man beim Abfragen des Werts eine Besonderheit beachten: Führt man die Anweisung

```
print(auswahl_var)
```

aus, wird folgendes in der Run-Konsole ausgegeben:

```
PY_VAR0
```

Das ist aber nicht der Wert der Variablen. **auswahl\_var** ist eben keine normale **int**-Variable. Ihren Wert müssen wir – auch wenn das etwas gewöhnungsbedürftig ist – mit ihrer **get()**-Methode abfragen. Zum Erfolg führt also folgende Anweisung:

```
print(auswahl_var.get())
```

Analog, gibt es eine Methode `set()`, mit der der Wert der **IntVar**-Variable verändert werden kann. Da wir über das Argument **variable** des Konstruktors **Radiobutton()** die Variable **auswahl\_var** mit dem Radiobutton verknüpft haben, ändert sich damit auch die Auswahl unter den Radiobuttons. Mit

22

```
auswahl_var.set(2)
```

selektieren wir automatisch das Radiobutton, dessen zugewiesener Wert 2 ist, also den „Links“-Radiobutton.

Gleiches ließe sich auch erreichen, indem die Methode `select()` der betreffenden **Radiobutton**-Instanz aufgerufen würde:

```
auswahl_links.select()
```

Dadurch, dass wir beiden Radiobuttons als **variable**-Option unsere **IntVar**-Variable **auswahl\_var** zugewiesen haben, weiß **tkinter**, dass diese beiden Radiobuttons zusammengehören und daher immer nur *einer von beiden* ausgewählt sein darf. Achten Sie also stets darauf, Radiobuttons, die zur selben „Auswahlgruppe“ gehören, auch dieselbe Variable als **variable**-Option zuzuweisen!

**Checkbuttons** funktionieren ganz ähnlich wie Radiobuttons, mit dem Unterschied natürlich, dass hier mehrere Checkboxen zugleich ausgewählt sein können, denn die beiden Checkboxen lassen sich ja unabhängig voneinander markieren. War beim Radiobutton noch klar, dass, wenn Button 1 ausgewählt ist, Button 2 nicht ebenfalls ausgewählt sein kann, gibt es bei zwei Checkbuttons nicht mehr nur zwei, sondern gleich vier verschiedene Zustände (beide ausgewählt, keiner ausgewählt, nur erster Button ausgewählt, nur zweiter Button ausgewählt); der Zustand des einen hängt also nicht mehr vom Zustand des anderen Buttons ab. Dementsprechend benötigen Sie auch zwei unterschiedliche **IntVar**-Variablen als **variable**-Argumente des Konstruktors **Checkbutton()**:

```
auswahl_rechts = IntVar()
auswahl_links = IntVar()

einfach_rechts = Checkbutton(win, text = 'Rechts', variable =
auswahl_rechts)
einfach_links = Checkbutton(win, text = 'Links', variable =
auswahl_links)

einfach_rechts.pack()
einfach_links.pack()
```

Vielleicht ist Ihnen aufgefallen, dass wir beim **Checkbutton()**-Konstruktor hier kein **value**-Argument mehr haben. Standardmäßig nimmt die Status-Variablen **va-**

■ Tab. 22.5 Spezielle Eigenschaften der Checkbutton- und Radiobutton Widgets

Option	Typ	Bedeutung
indicator	bool	Wenn <b>False</b> , dann wird statt des kreisförmigen (bei Radiobuttons) oder quadratischen (bei Checkbuttons) Auswahlelements ein richtiger Button angezeigt, der heruntergedrückt aussieht, wenn der Radiobutton/Checkbutton markiert ist.
selectcolor	str	Hintergrundfarbe des kreisförmigen (bei Radiobuttons) oder quadratischen (bei Checkbuttons) Auswahlelements.

riable den Wert **True** an, wenn der Checkbutton ausgewählt ist, und **False**, wenn nicht. Mit den optionalen Argumenten bzw. Widget-Optionen **onvalue** und **offvalue** lässt sich das bei Bedarf aber ändern.

Tabelle ■ Tab. 22.5 gibt eine Übersicht über die wichtigsten speziellen Eigenschaften von Widgets der Typen **Radiobutton** und **Checkbutton** an, über die in ■ Tab. 22.1 aufgeführten Standardoptionen hinaus.

### 22.2.3.6 Auswahl-Listen (Listbox)

Auswahl-Listen, repräsentiert durch die **tkinter**-Klasse **Listbox**, erlauben eine (einfache oder mehrfache) Auswahl aus einer Aufzählung von Texteinträgen.

Nach dem Erzeugen einer **Listbox** mit Hilfe des Klassenkonstruktors **Listbox()** und dem Anzeigen der **Listbox** im Fenster mittels der Methode **pack()**

```
namen = Listbox(win)
namen.pack()
```

können wir damit beginnen, der **Listbox** Einträge hinzufügen. Dazu verwenden wir die Methode **insert(index, eintrag, ...)**:

```
namen.insert('end', 'Sophie', 'Thomas', 'Ulrike', 'Tobias',
'Heike')
```

**insert()** weist einige Ähnlichkeiten mit der gleichnamigen Methode des **Entry**-Widgets auf: **index** gibt zunächst den numerischen Index des Elements an, *nach* dem eingefügt werden soll; wie bei der **insert()-Methode** von **Entry** kann dabei auch der Wert '**end**' angegeben werden, der dafür sorgt, dass die Einträge am Ende der Liste angefügt werden. Die Liste dabei auch um mehrere Einträge gleichzeitig ergänzt werden. Beachten Sie bitte, dass die Indizierung der Listeneinträge Python-typisch bei **0** beginnt (anders als die Indizierung der Zeichen bei den **Entry**-Widgets aber genauso wie die „Spaltenindizierung“ bei den mehrzeiligen **ScrolledText**-Widgets).

Ebenfalls analog zu **Entry** können Sie Listeneinträge mit der Methode **delete(index\_von, index\_bis)** wieder aus der **Listbox** löschen. Beim **Entry**-Widget hatte diese Methode einzelne Zeichen aus dem Inhalt des Entry-Feldes, also einem String und damit einer „Zeichen-Liste“, entfernt.

Durch Aufruf der Methode **selection\_set(index\_von, index\_bis)** markieren Sie Einträge in der **Listbox** so, als hätte der Benutzer sie selektiert.

Sowohl bei **delete()** als auch bei **selection\_set()** ist **index\_bis** jeweils ein optionales Argument, ein Argument also, das beim Aufruf der Methode auch weggelassen werden kann.

Ebenso, wie Sie mit **selection\_set()** einen (oder mehrere) Einträge markieren können, können Sie mit **selection\_includes(index)** prüfen, ob das mit **index** bezeichnete Element der Liste aktuell selektiert ist.

Mit **selection\_clear()** schließlich können Sie die aktuelle Markierung in der **Listbox** aufheben.

Die Anzahl der Listeneinträge ermitteln Sie mit Hilfe der Methode **length()**, während Ihnen **get(index\_von, index\_bis)** erlaubt, den Texteintrag eines oder mehrerer Einträge auszulesen. Wenn Sie dabei das optionale Argument **index\_bis** verwenden und damit einen Bereich angeben, so erhalten Sie als Rückgabewert ein *Tupel* mit den Texteinträgen der angegebenen Listeneinträge.

Tabelle □ Tab. 22.6 listet einige wichtige Optionen, die Sie auf die mittlerweile gewohnten Weisen für das **Listbox**-Widget anpassen können.

□ Tab. 22.6 Spezielle Eigenschaften des Listbox-Widgets

Option	Typ	Bedeutung
<b>activestyle</b>	<b>str</b>	Gibt an, wie der aktive Eintrag, das heißt, der markierte Eintrag, der aktuell den Fokus hat, visuell hervorgehoben werden soll; mögliche Ausprägungen sind ' <b>underline</b> ' (Text unterstrichen), ' <b>dotbox</b> ' (Umrandung mit gepunkteter Linie) und ' <b>none</b> ' (keine Hervorhebung), wobei ' <b>underline</b> ' der Standardwert ist.
<b>height</b>	<b>int</b>	Die Höhe der <b>Listbox</b> , gemessen aber nicht in Pixeln, sondern in Einträgen; der Standardwert ist 10. Wenn Sie mehr als die in <b>height</b> eingestellte Anzahl von Einträgen haben und sicherstellen wollen, dass der Eintrag an der Position <b>index</b> angezeigt wird, können Sie die Methode <b>see(index)</b> aufrufen; sie scrollt die <b>Listbox</b> so, dass der Eintrag <b>index</b> auf jeden Fall sichtbar ist.
<b>selectmode</b>	<b>str</b>	Gibt an, wie viele Einträge gleichzeitig selektiert sein können und wie diese zusammenhängen müssen; mögliche Ausprägungen sind: ' <b>single</b> ' (ein Eintrag auswählbar durch Klick auf den Eintrag), ' <b>browse</b> ' (ein Eintrag auswählbar durch Klick oder Bewegen der Maus mit gedrückter Maustaste), ' <b>multiple</b> ' (mehrere Einträge auswählbar, ein Klick auf einen Eintrag setzt die Markierung, wenn zuvor keine vorhanden war, bzw. entfernt sie, wenn bereits eine existierte) und ' <b>extended</b> ' (mehrere Einträge auswählbar durch Klick bei gedrückter <b>Ctrl</b> bzw. <b>Shift</b> -Taste); Standardwert ist ' <b>multiple</b> ', was für Windows-Benutzer etwas ungewöhnlt ist.

Daneben können Sie natürlich mit den Standard-Optionen arbeiten, die wir uns in Tabelle □ Tab. 22.1 angeschaut hatten. Viele von deren Eigenschaften können Sie übrigens auch auf *einzelne Einträge* in der **Listbox** anwenden. Dazu verwenden Sie die Methode **itemconfig(index, option=wert)**. Um zum Beispiel dem dritten Eintrag einen grünlichen Hintergrund zu geben, können Sie folgende Anweisung ausführen:

```
namen.itemconfig(2, background="#ED5036")
```

In Tabelle □ Tab. 22.6 finden Sie nun wieder einmal Fälle von Optionen, die eine von mehreren Ausprägungen annehmen können, zum Beispiel **activestyle**. Ange-sichts der eher schwierigen Dokumentationslage bei **tkinter** stellt sich natürlich die Frage, welche Ausprägungen überhaupt zulässig sind. Zwar weiß man dann noch immer nicht, was genau die einzelnen Ausprägungen bewirken, aber das lässt sich im Zweifel durch Ausprobieren recht schnell herausfinden. Wichtig wäre also im ersten Schritt, zu verstehen, welche Möglichkeiten Sie überhaupt haben. Dazu gibt es einen einfachen Trick: Sie provozieren einen Fehler. Betrachten Sie dazu den folgenden Code:

```
namen['activestyle'] = 'xxx'
```

Hier versuchen wir der Option **activestyle** den Wert 'xxx' zuzuweisen, der mit hoher Wahrscheinlichkeit (und so ist es in der Tat) keine zulässige Ausprägung für diese Option darstellt. Führen wir ein Programm mit dieser Anweisung aus, erhalten wir eine Fehlermeldung, aber genau das ist es, was wir in diesem Fall erreichen wollen. Es ergibt sich im Beispiel der folgende Fehler:

```
Traceback (most recent call last):
  File "C:/Users/MeinUser/Desktop/Dokumente/tkinter_hallo_welt.py", line 138, in <module>
    namen['activestyle']='xxx'
  File "C:/Users/MeinUser/AppData/Local/Programs/Python/Python37-32/lib/tkinter/__init__.py", line 1489, in __setitem__
    self.configure({key: value})
  File "C:/Users/MeinUser/AppData/Local/Programs/Python/Python37-32/lib/tkinter/__init__.py", line 1482, in configure
    return self._configure('configure', cnf, kw)
  File "C:/Users/MeinUser/AppData/Local/Programs/Python/Python37-32/lib/tkinter/__init__.py", line 1473, in _configure
    self.tk.call(_flatten((self._w, cmd)) + self._options(cnf))
_tkinter.TclError: bad activestyle "xxx": must be dotbox, none, or underline
```

Das Entscheidende steht in der letzten Zeile. Dort listet die Fehlermeldung nämlich die möglichen Ausprägungen, die sich jetzt leicht ausprobieren lassen.

### 22.2.3.7 Nachrichten-/Entscheidungsdialoge (MessageBox)

Manchmal möchten Sie den Benutzer auf etwas hinweisen, und zwar auf eine Weise, die ihn dazu zwingt, Ihre Nachricht zur Kenntnis zu nehmen. Oder aber, Sie wollen den Benutzer zu einer sofortigen Entscheidung zwingen, beispielsweise, ob eine Datei beim Speichern überschrieben werden soll, oder nicht. In all diesen Fällen ist eine *MessageBox* das Mittel der Wahl.

Eine MessageBox erzeugen Sie mit Hilfe des **tkinter**-Moduls **messagebox**. Weil die MessageBox, anders als die meisten der anderen Steuerelemente, die wir bisher kennengelernt haben, in also in einem eigenen Modul „verpackt“ ist (ebenso wie **ScrolledText**), müssen wir die darin enthaltenen Klasse zunächst importieren. Wir machen es uns hier einfach und importieren mit der Stern-Wildcard einfach alle Klassen, die im Modul **messagebox** enthalten sind:

```
from tkinter.messagebox import *
```

Danach können wir mit Hilfe der Funktionen **showinfo(*titel, nachricht*)**, **showwarning(*titel, nachricht*)** und **showerror(*titel, nachricht*)** eine MessageBox anzeigen lassen, je nach Art der Nachricht (Information, Warnung oder Fehler) mit einem jeweils anderen Icon. Hier das Beispiel einer Warnmeldung:

```
showwarning('Achtung', 'Das eingegebene Alter muss größer als 0 sein.')
```

Sie können die Gestaltung der Messageboxen aber auch noch anpassen, in dem Sie beim Aufruf der jeweiligen Funktion weitere Optionen mitgeben. Mögliche Optionen sind hier **icon** (welches Icon soll angezeigt werden?), **type** (welche Buttons sollen angezeigt werden?) und **default** (welcher Button soll vorausgewählt sein, sodass er ausgelöst wird, wenn der Benutzer die <ENTER>-Taste drückt).

Die Option **icon** kann dabei die Werte '**info**', '**warning**', '**error**' und '**question**' annehmen. Die ersten drei sind die Icons, die auch von den Funktionen **showinfo()**, **showwarning()** und **showerror()** verwendet werden, '**question**' zeigt ein Fragezeichen-Icon an. Ebenso wie die Icons lassen sich auch die dem Benutzer zur Verfügung stehende Buttonkombination mit Hilfe von Konstanten festlegen, in diesem Fall mit '**ok**' (*Okay*-Button), '**okcancel**' (*Okay* und *Abbrechen*), '**yesno**' (*Ja* und *Nein*), '**yesnocancel**' (*Ja*, *Nein* und *Abbrechen*), '**retrycancel**' (*Wiederholen* und *Abbrechen*) und '**abortretryignore**' (*Abbrechen*, *Wiederholen* und *Ignorieren*).

Wollten wir also beispielsweise einen kleinen Dialog anzeigen, die den Benutzer fragt, ob er eine Datei überschreiben will, und ihm die Buttons *Ja*, *Nein* und *Abbrechen* anbieten, so könnten wir eine passende MessageBox so erzeugen:

```
feedback = showwarning('Bestätigung',
                      'Möchten Sie die Datei tatsächlich überschreiben?',
                      icon = 'question',
                      type = 'yesnocancel',
                      default = 'yes')
print(feedback)
```

Sie sehen an diesem Beispiel, dass wir, obwohl wir die Funktion **showwarning()** verwenden, die ja standardmäßig ein Ausrufezeichen als Icon anzeigt, das Standard-Verhalten durch Angabe der **icon**-Option überschreiben können.

Die Funktion liefert als Wert einen kleingeschriebenen String zurück, der die Beschriftung des Buttons enthält, das angeklickt wurde, in unserem Beispiel also '**'yes'**', '**'no'**' oder '**'cancel'**'. Dieser Funktionswert ist wichtig, um auf die Eingabe des Benutzers dann entsprechend reagieren zu können.

Anders als die Steuerelemente, die wir in den vorangegangenen Abschnitten betrachtet haben, ist die Messagebox keine Klasse, von der wir eine Instanz erzeugen müssen. Es ist vielmehr einfach eine Funktion, die im Modul **messagebox** enthalten ist. Ganz ähnlich ist es auch bei den Dateidialogen, mit denen wir uns im folgenden Abschnitt beschäftigen.

### 22.2.3.8 Datei-Öffnen-/Datei-Speichern-Dialoge

Wenn Sie den Benutzer eine Datei auswählen lassen wollen, die geöffnet werden soll, oder in der etwas gespeichert werden soll, erlaubt Ihnen das **tkinter**-Modul **filedialog**, auf einfache Weise, die bekannten Standard-Dialoge „Datei öffnen“ und „Datei speichern unter“ auch in Ihren eigenen Programmen zu verwenden.

Zunächst muss das Modul importiert werden. Der Einfachheit halber importieren wir alle im Modul enthaltenen Klassen und Funktionen, genau so, wie wir es im vorangegangenen Abschnitt mit dem Modul **messagebox** auch getan haben:

```
from tkinter.filedialog import *
```

Danach können mit Hilfe der Funktionen **askopenfilename()** bzw. **asksaveasfilename()** die Dialoge für Öffnen bzw. Speichern von Dateien aufgerufen werden. Im folgenden Beispiel lassen wir eine Datei zum Öffnen auswählen und geben das Ergebnis in der (Run-)Konsole aus:

```
datei = askopenfilename(defaultextension = 'txt',
                        filetypes = [
                            ('Textdateien', '*.txt'),
                            ('Alle Dateien', '*.*')],
                        title = 'Datei öffnen...',
                        initialdir = 'C:\\Windows')
print(datei)
```

Die Funktionen geben den Namen (inklusive Pfad) der ausgewählten Datei zurück. Bricht der Benutzer den Dialog ab, ohne eine Datei auszuwählen bzw. einen

Dateinamen einzugeben, wird ein leerer String zurückgegeben. Wie Sie am Beispiel oben sehen, können Sie das Verhalten des Dialogs mit einigen Optionen steuern. **filetypes** ist eine Liste (beachten Sie die umschließenden eckigen Klammern!) von Tupeln, bestehend jeweils aus einer Beschreibung und einer Dateiendung; die so definierten Dateitypen kann der Benutzer dann vorauswählen. Mit der **title**-Option wird der Dialogfenster-Titel gesteuert, mit **initialdir** das Verzeichnis, dessen Inhalt beim Öffnen des Dialogs standardmäßig angezeigt werden soll; der Backslash (\) als Bestandteil des Pfadnamens muss dabei doppelt geschrieben werden, einen einfachen Backslash würde Python als den Versuch interpretieren, das dahinterstehende Zeichen zu escapen; wenn Ihnen das Escapen nicht mehr geläufig ist, Blättern Sie nochmal einige Seiten zurück zu Abschn. 11.2.2.

Übrigens: Vielleicht haben Sie sich schon im letzten Abschnitt, beim Modul **messagebox** gefragt, warum wir hier eigentlich nicht mit Klassen arbeiten, wie bei den übrigen Widgets, sondern stattdessen Funktionen wie **showwarning()** oder **askopenfilename()** aufrufen. Die Antwort lautet: Weil es so für uns am einfachsten ist! Diese Funktionen erzeugen im Hintergrund die notwendigen Klasseninstanzen; da wir aber nur an den Ergebnissen interessiert sind, hier also an den Dateinamen, und nicht wirklich mit den Dialog(klasseninstanzen) selbst arbeiten wollen, genügt es vollkommen, dass uns eine entsprechende Funktion einfach den Dateinamen liefert und uns die Mühe erspart, selbst Instanzen der erforderlichen Klassen anzulegen und mit ihnen zu arbeiten. Anders ist die Situation natürlich bei den übrigen Widgets, die wir typischerweise nicht nur kurzfristig nutzen, sondern die in unserem Anwendungsfenster ja dauerhaft vorhanden sind und mit denen wir auch später noch arbeiten wollen.

### 22.2.3.9 Weitere Widgets

Neben den Widgets, die wir uns in den vorangegangenen Abschnitten angesehen haben, gibt es noch eine ganze Reihe weiterer Steuerelemente, die Sie bei der Gestaltung Ihrer Programm-Oberfläche verwenden können, zum Beispiel **Canvas** (Zeichenfläche), **Spinbox** (Eingabefeld mit Hoch-Runter-Pfeilen zum Verstellen des Wertes), **Tree View** (hierarchische Darstellung von Elementen; Import aus Modul **tkinter.ttk** notwendig), **Notebook** (Darstellung von Steuerelementen auf Registerkarten/Reitern; in Modul **tkinter.ttk**), **Progressbar** (Fortschrittsanzeige; in Modul **tkinter.ttk**), um nur einige zu nennen.

Für die Arbeit mit Bildern, die vielen Steuerelementen (etwa Menüeinträgen, Buttons als Icons) hinzugefügt oder auch als Hintergrund (zum Beispiel des Hauptanwendungsfensters) verwendet werden können, existiert darüber hinaus die Klasse **PhotoImage**.

Wie Sie sehen, gibt es bei **tkinter** noch eine Menge zu entdecken! Einziger Wermutstropfen ist die bisweilen verhältnismäßig dürftige Dokumentationslage; zumindest für die oben genannten Widgets aus dem **tkinter.ttk**-Modul existiert aber eine einigermaßen ausreichende „offizielle“ Dokumentation (aktuell unter ► <https://docs.python.org/3/library/tkinter.ttk.html>). Darüber hinaus ist dann aber oftmals eigene Internet-Recherche die beste Option. Meist wird man hierbei zum Glück schnell fündig, haben doch viele Programmierer vor Ihnen bereits ganz ähnliche Fragen gehabt!

## 22.2.4 Anordnen der Bedienelemente (Geometry Managers)

In den vorangegangenen Abschnitten haben wir eine Reihe von Bedienelementen kennengelernt, die es dem Benutzer erlauben, das Programm über die Oberfläche zu steuern. Um nun eine Oberfläche zu konstruieren, die auch optisch ansprechend und gut zu bedienen ist, müssen die verschiedenen Bedienelemente auf der Oberfläche noch dort platziert werden, wo sie stehen sollen.

Um die Bedienelemente auf der Oberfläche anzuordnen, bietet **tkinter** drei so genannte *Geometry Managers* an, mit denen wir uns in den folgenden Abschnitten näher befassen. Jeder der drei folgt einem anderen Grundprinzip, wie die Position der Bedienelemente auf der Oberfläche definiert wird. Je nachdem, welcher Ansatz der Positionierung für die programmierte Anwendung am besten geeignet erscheint, entscheidet man sich für den jeweiligen Geometry Manager.

Die drei Geometry Manager sind:

- **Pack:** **Pack** platziert die Bedienelemente einfach eines neben dem anderen, bzw. eines unter dem anderen. In den vorangegangenen Abschnitten haben wir mit diesem Geometry Manager bereits gearbeitet, indem wir die Methode **pack()** der Widgets aufgerufen haben. Diese hat bewirkt, dass die Widgets untereinander platziert und dabei im zur Verfügung stehenden Raum zentriert wurden.
- **Grid:** **Grid** teilt den zur Verfügung stehenden Raum auf der Oberfläche gedanklich in ein Raster aus Zeilen und Spalten ein und erlaubt es, Steuerelemente in jeder „Zelle“ dieses Rasters zu platzieren. **Grid** eignet sich gut, um komplexe Oberflächen zu konstruieren und sollte für die meisten Zwecke genügen.
- **Place:** **Place** positioniert Elemente anhand ihrer *absoluten* „Koordinaten“, die in Bildpunkten gemessen werden, oder *relativ* (gemessen in Anteilen der Fensterbreite/-höhe) zu linken, oberen Ecke des Fensters.

### 22.2.4.1 Pack

Als wir uns die verschiedenen Widgets genauer angesehen haben, haben wir mit Hilfe der Methode **pack()** dafür gesorgt, dass diese Widgets auch tatsächlich auf unserer Programmoberfläche dargestellt wurden. Allerdings wurden die Widgets nicht nur einfach angezeigt, sondern natürlich auch zugleich im Fenster positioniert, und zwar untereinander.

Und genau das ist es, was der Geometry Manager **Pack** leistet: Wie der Name schon suggeriert, „packt“ er die verschiedenen Steuerelemente zusammen, entweder vertikal als „Stapel“ (das ist die standardmäßige Ausrichtung) oder auch horizontal als „Reihe“. Wo ein Element genau auf der Oberfläche zu sehen ist, hängt dann vor allem davon ab, wie groß seine Vorgänger im Stapel (bei vertikaler Ausrichtung) bzw. der Reihe (bei horizontaler Ausrichtung) sind.

Als Beispiel betrachten wir eine Programmoberfläche, die zur Eingabe eines Passworts dient:

```
from tkinter import Tk, Button

win = Tk()

win.title('Anwendung der Geometry Managers')
win.geometry('500x100')

prompt = Label(win, text = 'Bitte geben Sie Ihr Passwort ein:')
pwd = Entry(win)
login = Button(win, text = 'Login')

pwd['show'] = '*'
pwd['width'] = 20
pwd.focus()
```

Wir erzeugen drei Widgets: ein **Label** namens **prompt** zur Anzeige einer Eingabeaufforderung, ein **Entry**-Feld **pwd**, das das Passwort aufnimmt und einen **Button** **login**, der zum Bestätigen der Passworteingabe dient. Das **Entry**-Feld **pwd** wird dann noch so konfiguriert, dass die Passworteingabe verdeckt ist und nur Sternchen für die eingegeben Zeichen angezeigt werden. Außerdem setzen wir seine Breite auf 20 Zeichen und weisen ihm den Fokus zu, sodass der Benutzer direkt mit der Eingabe beginnen kann.

Als nächstes rufen wir jetzt, wie bereits zuvor, die Methode **pack()** auf, um den Pack-Geometry-Manager die Widgets auf der Oberfläche platzieren zu lassen. Damit die Bedienelemente nebeneinander und nicht – wie es standardmäßig geschehen würde – untereinander dargestellt werden, rufen wir **pack()** mit der Option **side** auf, die neben dem Standardwert '**top**' die Konstanten '**left**', '**right**' und '**bottom**' als Werte für die Richtung der Widget-Positionierung akzeptiert:

```
prompt.pack(side = 'left')
pwd.pack(side = 'left')
login.pack(side = 'left')

win.mainloop()
```

Das Ergebnis sehen Sie in Abb. 22.2. Die Widgets stehen nun unmittelbar nebeneinander. Mit der Option **padx** (und vertikal **pady**) können wir etwas Abstand (*padding*) jeweils links und rechts von jedem Widgets einfügen, zum Beispiel mit:

```
pwd.pack(side = 'left', padx = 5)
```

Mit zwei weiteren wichtigen Optionen lässt sich das Verhalten von **pack()** noch detaillierter steuern: **expand**, das die Werte **1** und **0** bzw. **True** und **False** annehmen kann, legt fest, ob der **Pack** Geometry Manager die volle Breite, die er zur Verfü-

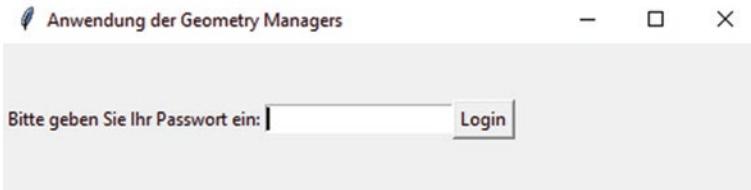


Abb. 22.2 Anordnung der Widgets mit pack(side = LEFT)



Abb. 22.3 Anordnung der Widgets mit expand-Option

gung hat, nutzen soll. In unserem Beispiel ist das die gesamt Fensterbreite. Würden wir für alle drei Widgets die Standardeinstellung ändern und **pack()** mit **expand=1** anwisen, die gesamte Breite zu nutzen, also

```
prompt.pack(side = 'left', expand = 1)
pwd.pack(side = 'left', expand = 1)
login.pack(side = 'left', expand = 1)
```

erhielten wir ein Ergebnis wie in Abb. 22.3

Damit werden die Widgets immer noch „gepackt“, aber jedes Widget hat mehr Platz. Innerhalb des jedem Widgets zur Verfügung stehenden Raums wiederum wird das Bedienelement standardmäßig zentriert, was sich aber mit den nach den englischen Abkürzungen der Himmelsrichtungen benannten Ausprägungen '**n**', '**ne**', '**e**', '**se**', '**s**', '**sw**', '**w**', '**nw**', und '**center**' (Standard) der zusätzlichen Option **anchor** leicht anpassen lässt. Blättern Sie nochmal einige Seiten zurück, dieselbe Art der Ausrichtungsangabe hatten wir bereits bei der **anchor**-Option für das **Label**-Widget kennengelernt.

Auch können wir **pack()** anwisen, die Widgets in dem ihnen zur Verfügung stehenden Raum maximal auszudehnen, was für manche Zwecke nützlich ist, oftmals aber eher seltsam aussieht. Dazu setzen wir die Option **fill** auf eine der Konstanten '**x**' (ausdehnen in horizontaler Richtung), '**y**' (ausdehnen in vertikaler Richtung) oder '**both**' (ausdehnen sowohl horizontal als auch vertikal). Genau das geschieht im folgenden Beispiel mit dem Eingabefeld, das wir in horizontaler Richtung, und dem Button, den wir in beiden Richtungen den zur Verfügung stehenden Platz ausnutzen lassen, was dann die in Abb. 22.4 zu sehende Oberfläche ergibt:



Abb. 22.4 Anordnung der Widgets mit verschiedenen Optionen

```
prompt.pack(side = 'left', fill = 'x', anchor = 'w')
pwd.pack(side = 'left', expand = 1, fill = 'x', padx = 5,
         anchor = 'w')
login.pack(side = 'left', expand = 1, fill = 'both',
           anchor = 'w')
```

In der Regel wird man, wie an diesem Beispiel deutlich wird, zunächst ein wenig mit den Pack-Optionen **side**, **fill**, **expand**, **anchor** und **padx/pady** herumspielen müssen, bis man ein gutes Layout gefunden hat. Mit etwas Erfahrung kann man dann natürlich auch eine Oberflächenstruktur, die man sich überlegt oder bestenfalls sogar auf Papier oder auch digital skizziert hat, fast beim ersten Anlauf umsetzen können.

#### 22.2.4.2 Grid

Während Pack versucht, die Steuerelemente Seite an Seite „packt“, versteht der Geometry Manager **Grid** das Anwendungsfenster als ein Raster (*grid*) aus Zeilen (*rows*) und Spalten (*columns*). Steuerelemente können innerhalb des Grids frei positioniert werden. Dabei funktioniert die Ansprache der einzelnen „Zellen“ im Grid mit den Python-typischen, bei 0 beginnenden Indizes.

Betrachten Sie das bereits aus dem letzten Abschnitt bekannte Passwort-Eingabe-Beispiel:

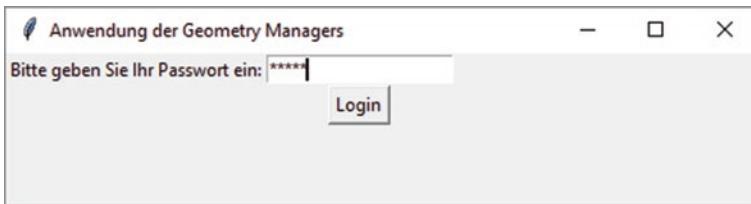
```
from tkinter import Tk, Button

win = Tk()

win.title('Anwendung der Geometry Managers')
win.geometry('500x100')

prompt = Label(win, text = 'Bitte geben Sie Ihr Passwort ein:')
pwd = Entry(win)
login = Button(win, text = 'Login')

pwd['show'] = '*'
pwd['width'] = 20
pwd.focus()
```



■ Abb. 22.5 Anordnung der Widgets mit grid()

Im nächsten Schritt positionieren wir mit Hilfe der Methode **grid(row = zeile, column = spalte)** die Eingabeaufforderung und das Eingabefeld nebeneinander, den Login-Button unter dem Eingabefeld:

```
prompt.grid(row = 0, column = 0)
pwd.grid(row = 0, column = 1)
login.grid(row = 1, column = 1)

win.mainloop()
```

Das Ergebnis ist in ■ Abb. 22.5 dargestellt.

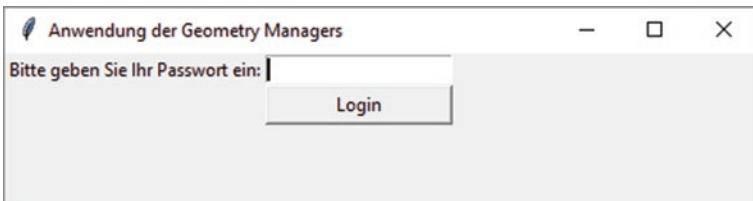
Wie Sie sehen, werden Eingabefeld und Button in derselben Spalte, nämlich Spalte 1 (also der zweiten Spalte) dargestellt. Die Zellen wählt der Geometry Manager dabei so, dass die Widgets gerade hineinpassen. Innerhalb der Zellen werden die Steuerelemente standardmäßig zentriert (vertikal, und – wie man am Beispiel des Buttons gut sehen kann – horizontal).

Die Ausrichtung der Widgets innerhalb ihrer Zellen lässt sich aber mit Hilfe der Option **sticky** leicht beeinflussen: Wie bereits mehrere Male zuvor gesehen, ist **sticky** eine Himmelsrichtungsangabe. **sticky = 'w'** bedeutet daher, dass das Widget „im Westen“, also am linken Zellenrand ausgerichtet werden soll. Zugleich lässt sich mit den Himmelsrichtungen auch die Breite des Widgets so regulieren, dass es, wenn es eigentlich kleiner als die Zelle wäre, diese trotzdem voll ausfüllt. So führt etwa **sticky = 'we'** dazu, dass sich das Widget in seiner Zelle, deren Breite von dem breiteren Eingabefeld in der Zeile darüber bestimmt ist, „von Westen nach Osten“ erstreckt, also über die gesamte Zellenbreite.

Das Resultat sehen Sie in ■ Abb. 22.6.

Manchmal reicht das aber nicht und Sie wollen, dass sich ein Widget über mehrere Spalten oder mehrere Zeilen erstreckt. In diesem Fall helfen die Optionen **rowspan** und **columnspan**, die jeweils die Zahl der überspannten Zeilen bzw. Spalten angeben. Mit

```
login.grid(row = 1, column = 0, sticky = 'we', columnspan = 3)
```



■ Abb. 22.6 Anordnung der Widgets mit grid() und der Option sticky = 'we' für das Button



■ Abb. 22.7 Anordnung der Widgets mit grid() und den Optionen padx und pady

würde unser Button vom linken Rand der Eingabeaufforderung bis zum rechten Rand des Eingabefeldes reichen.

Mit den bereits von **pack()** bekannten Optionen **padx** und **pady** können Sie einen linken/rechten bzw. oberen/unteren Abstand festlegen, was die Darstellung im allgemeinen etwas entzerrt, wie am Beispiel in ■ Abb. 22.7 zu sehen ist, wo mit **padx**- und **pady**-Werten von 5 gearbeitet wurde.

Wie groß das Grid in Ihrem Anwendungsfenster insgesamt ist, richtet sich übrigens danach, wohin Sie die am weitesten außen positionierten Widgets platziert haben. In unserem Beispiel verfügt das Grid über zwei Spalten und zwei Zeilen; würden wir mit **widget.grid(row = 0, column = 2)** ein Widget rechts neben das Eingabefeld platzieren, käme eine dritte Spalte hinzu.

**Grid** ist ein sehr populärer Geometry Manager, weil er es mit geringem Aufwand erlaubt, effektvolle Oberflächen zu gestalten und dabei eine intuitive Positionierung vorzunehmen.

Das Beispiel in ► Abschn. 22.2.6, wo wir eine vollständige **tkinter**-Anwendung entwickeln, wird ebenfalls mit **Grid** arbeiten.

Vielleicht haben Sie sich schon gefragt, ob es auch mit Pack möglich ist, ein Steuerelemente-Layout wie in den vorangegangenen Beispielen zu realisieren, also ein Layout mit mehr als einer Zeile *und* mehr als einer Spalte. Die Antwort darauf lautet zwar ja, aber es ist ungleich schwieriger als mit Grid, ein solches Layout zu erreichen, weil man mit unterschiedlichen *Frames* arbeiten muss. Ein **Frame** ist ein spezielles Widget, das wir hier nicht weiter behandeln und das – ebenso wie ein **Tk**-Fenster – andere Widgets aufnehmen kann. Um ein Layout wie in unseren Beispielen oben zu erhalten, könnte man dann zum Beispiel zwei Frames erzeugen, die

jeweils eine (mit „horizontalem“ **pack()** zusammengeschobene) „Zeile“ unserer Darstellung repräsentieren, und diese dann wiederum mit „vertikalem“ **pack()** untereinander setzen. Wie Sie sehen, ist hier etwas mehr Kreativität gefragt, wenn es um die technische Realisierung der angedachten Programmoberfläche geht.

#### 22.2.4.3 Place

Der dritte, letzte und wahrscheinlich in der Praxis am wenigsten häufig eingesetzte Geometry Manager ist **Place**. **Place** dient dazu, Widgets entweder mit Hilfe der Argumente **x** und **y** an eine durch x- und y-Koordinate angegebene *absolute* Position zu setzen, oder aber an eine *relative* Position. Relativ bedeutet dabei, dass die Position relativ zur linken oberen Ecke des Fensters *in Prozent der Fensterbreite* bzw. *-höhe* gemessen wird. Die Anweisung

```
login.place(relx = 0.5, rely = 0.5)
```

positioniert den Button **login** also exakt in der Fenstermitte, nämlich nach 50 % der Fensterbreite (**relx**) und 50 % der Fensterhöhe (**rely**) von der linken oberen Ecke des Fensters entfernt. Genauer gesagt: Die *obere linke Ecke* des Buttons wird dort platziert. Mit der bereits von **pack()** bekannten Option **anchor** kann auch eine andere Ecke als „Anker“ der Positionierung festgelegt werden. Dabei wird die Ecke wieder durch Himmelsrichtungen angegeben, zum Beispiel 'se' für die südöstliche, also die rechte untere Ecke.

Auch die Größe der Widgets kann spezifiziert werden, entweder absolut mit Hilfe der Optionen **width height** für Breite und Höhe, oder durch eine relative Größenangabe. Dabei kann mit **relwidth** und **relheight** eine Breite bzw. Höhe in Prozent der Fensterbreite bzw. -höhe angegeben werden.

Die Angabe relativer Positionierungen und Größen hat den Vorteil, dass sich Position und Größe der Widgets bei Veränderung der Fenstergröße mit anpassen, ein Effekt, der auch bei den Geometry Managers **Pack** und **Grid** erzielbar ist (bei letzterem allerdings etwas mühsamer), der bei Verwendung von Place mit absoluten Positionsangaben aber nicht gegeben ist.

#### 22.2.5 Ereignisse

##### ■ Die command-Option der tkinter-Widgets

Bisher haben wir zwar ansprechende Oberflächen zusammengebaut, diese sind aber weitestgehend ohne Funktion. Nichts passiert, wenn man auf einen unserer Buttons oder Menüeinträge klickt. Das soll sich jetzt ändern.

Manche Widgets, wie etwa **Button** oder **Menu**, bringen die Möglichkeit mit, eine Funktion anzugeben, die immer dann aufgerufen wird, wenn das Steuerelement durch den Benutzer „ausgelöst“ wird. Betrachten Sie dazu das folgende einfache Beispiel der mittlerweile gut bekannten Umrechnung zwischen Kelvin und Grad Celsius. Ein kleiner Umrechner mit grafischer Oberfläche könnte so aussehen:

```

from tkinter import Tk, Menu, Entry, Button, Label

def umrechnen():
    lb_erg['text'] = 'Umrechnungsergebnis: ' + str(
        round(float(en_kelvin.get()) - 273.15, 2)) + ' °C.'

def beenden():
    quit()

win = Tk()
win.title('Kelvin-Celsius-Umrechnung')
win.geometry('400x150')
menuleiste_oben = Menu(win)
win.config(menu = menuleiste_oben)
aktionsmenu = Menu(menuleiste_oben, tearoff = 0)
aktionsmenu.add_command(label = 'Umrechnen',
                        command = umrechnen)
aktionsmenu.add_separator()
aktionsmenu.add_command(label = 'Beenden',
                        command = beenden)
menuleiste_oben.add_cascade(label = 'Aktion',
                             menu = aktionsmenu)
lb_eingabe = Label(text = 'Temperatur in Kelvin:')
en_kelvin = Entry()
bt_umr = Button(win, text = 'Umrechnen',
                 command = umrechnen)
lb_erg = Label(width = 30)
lb_eingabe.pack()
en_kelvin.pack()
bt_umr.pack(pady = 10)
lb_erg.pack(pady = 10)
win.mainloop()

```

Die Oberfläche des Programms ist in Abb. 22.8 dargestellt.

Im Unterschied zu den vorangegangenen Abschnitten haben wir dieses Mal die **command**-Option beim Aufruf der Button-Konstruktor-Methode **Button()** um beim Hinzufügen von Menüeinträgen mit **add\_command()** verwendet. Der **command**-Option wird jeweils eine Funktion zugewiesen, eine sogenannte *Event-Handler*- oder auch *Callback*-Funktion, die immer dann automatisch aufgerufen wird, wenn das jeweilige Steuerelement vom Benutzer „ausgelöst“ wird. Wird also beispielsweise unser „Umrechnen“-Button angeklickt, wird automatisch die Funktion **umrechnen()** aufgerufen. Die Funktion **umrechnen()** haben wir weiter oben im Programm definiert (die Funktionsdefinition muss im Programmquelltext vor der Zuweisung an die **command**-Option stehen); sie nimmt einfach die Umrechnung vor und gibt das Ergebnis auf dem Label **lb\_erg** aus. Beachten Sie bitte, dass beim Zuweisen der Event-Handler-Funktion an die **command**-Option keine runden



Abb. 22.8 Kelvin-zu-Celsius-Umrechner

Klammern hinter dem Funktionsnamen angegeben werden. Das liegt daran, dass die **command**-Option einfach das Funktionsobjekt übergeben bekommt; Sie erinnern sich, dass in Python auch Funktionen Objekte sind – nur bei einer Funktionsdefinition (wie Sie weiter unten noch sehen werden) oder beim Aufruf einer Funktion müssten wir die runden Klammern mit angeben, nicht aber, wenn wir das Funktionsobjekt selbst meinen.

#### ■ Event-Handler an Ereignisse binden

**tkinter** kennt noch eine weitere, ungleich mächtigere Art, auf Ereignisse zu reagieren. Dies geschieht mit Hilfe der Widget-Methode **bind()**. Mit der Anweisung

```
bt_umr.bind('<Button-1>', umrechnen)
```

hätten wir den gleichen Effekt erreichen können wie mit der **command**-Option im **Button()**-Konstruktor oben. Die Methode **bind(ereignis, eventhandler\_funktion)** bindet eine Event-Handler-Funktion an ein Ereignis. Fortan wacht unsere Ereignisverarbeitung **mainloop()** mit Argusaugen darüber, ob das Ereignis ausgelöst wird und ruft die Event-Handler-Funktion auf, wenn das der Fall sein sollte. Die Zeichenkette '**<Button-1>**' repräsentiert das Ereignis, dass die linke/primäre Maustaste („Taste 1“) gedrückt wird (die rechte Maustaste wäre übrigens '**<Button-3>**', '**<Button-2>**' die mittlere Maustaste). Neben diesen Button-Events gibt es eine Vielzahl solcher Ereignisse, an die wir Event Handler binden können; hier einige Beispiele:

- **<DoubleButton-1>**: Doppelklick mit der linken Maustaste.
- **<Enter>** und **<Leave>**: Der Benutzer ist mit dem Mauszeiger in den Bereich des Steuerelements eingetreten bzw. der Mauszeiger hat den Bereich des Steuer-elements verlassen.
- **a, b, c, ...**: Der jeweilige Buchstabe wurde gedrückt.
- **<Key>**: Es wurde *irgendein* Buchstabe gedrückt.
- **<F1>, ...**: Die jeweilige Funktionstaste wurde ausgelöst.

- <Escape>, <BackSpace> (Löschen), <Delete> (Entfernen), <Tab> (Tabulator), <Return> (Return/Enter), <Shift\_L> (Umschalt-/Shift), <Control\_L> (Strg), <Alt\_L> (Alt), <End> (Ende), <Home> (Pos 1), <Left> (Pfeil links), <Up> (Pfeil hoch), <Right> (Pfeil rechts), <Down> (Pfeil runter), <Print> (Drucken), <Insert> (Einfügen): Die jeweilige Spezialtaste wurde gedrückt.

## 22

Auch Tastenkombinationen sind damit darstellbar: Wollen Sie eine Funktion zum Beispiel an das Ereignis binden, dass <CTRL> und <S> gleichzeitig gedrückt wurden, so können Sie als Ereignis einfach als '<Control\_L>S' angeben.

Werfen wir nochmal einen etwas genaueren Blick auf die Event Handler, die wir mit **bind()** an ein Ereignis binden: Diese Funktionen bekommen nämlich automatisch ein Argument vom Typ **Event** übergeben. Dementsprechend müssen wir unsere bisherigen Event Handler, die wir zum Beispiel der der **command**-Option unseres Buttons zugewiesen haben, anpassen, denn diese Event Handler benötigten ja keinerlei Argument. Die Änderung ist marginal, vermeidet aber einen Laufzeitfehler:

```
def umrechnen(ev = None):
    lb_erg['text'] = 'Umrechnungsergebnis: ' + str(
        round(float(en_kelvin.get()) - 273.15, 2)) + ' °C.'
```

Wir brauchen mit dem **Event**-Objekt **ev**, mit dem unser Event Handler aufgerufen wird, gar nichts tun, der Event Handler muss aber das Argument vorsehen. Indem wir dem Argument einen Standardwert (nämlich **None**, also „nichts“) mitgeben, machen wir die Funktion auch noch aufrufbar für die **command**-Option unseres Menüeintrags „Umrechnen“, denn die ruft den Event Handler ja *ohne* Argument auf. Der Event Handler muss also sowohl damit zurechtkommen, dass er mit einem Argument aufgerufen wird, als auch damit, dass das Argument entfällt.

Was aber hat es nun inhaltlich mit diesem Event-Objekt auf sich? Das Event-Objekt liefert einige Informationen zum Ereignis, vor allem:

- **x,y**: Die Mausposition relativ (relativ zu linken oberen Ecke des Fensters), an der das Ereignis ausgelöst wurde (interessant natürlich für Klick-Events)
- **widget**: Das Widget, das das Ereignis ausgelöst hat.
- **char**: Die gedrückte Zeichentaste (interessant vor allem für das Ereignis <Key>).

Übrigens: Sie können Ereignisse natürlich auch direkt an das Anwendungsfenster binden, in unserem Fall also an das **Tk**-Objekt **win**. Wird ein Ereignis für ein Widget ausgelöst, wird zunächst automatisch geprüft, ob für das jeweilige Widget ein Event Handler an dieses Ereignis gebunden ist. Ist das nicht der Fall, wird geprüft, ob das „nächsthöhere“ Objekt, in unserem Fall eben das Anwendungsfenster, eine

Ereignisbehandlungsroutine für dieses Ereignis aufweist. In diesem Sinne wird also vom „Speziellen zum Allgemeinen“ auf das Vorhandensein von Event Handlern geprüft, die Event Handler bilden gewissermaßen eine Hierarchie.

### 22.2.6 Beispiel: Taschenrechner-Applikation

In diesem Abschnitt werden wir einen einfachen Taschenrechner mit **tkinter** entwickeln. Der Taschenrechner soll die vier Grundrechenarten beherrschen und es erlauben, das Ergebnis der Berechnungen in die Zwischenablage zu kopieren.

Das Ergebnis sehen Sie in □ Abb. 22.9.



□ Abb. 22.9 Die Oberfläche der Taschenrechner-Anwendung

Schauen wir uns den Code nun Schritt für Schritt an:

```

1  from tkinter import Tk, Button, Label
2  from tkinter.font import Font
3  from functools import partial
4
5
6  # Eventhandler-Funktionen für Buttons
7  def ziffer_operator_press(ziffer_operator):
8      display['text'] = display['text'] + ziffer_operator
9
10
11 def loeschen_press():
12     display['text'] = ''
13
14
15 def kopieren_press():
16     win.clipboard_clear()
17     win.clipboard_append(display['text'])
18
19
20 def plusminus_press():
21     display['text'] = '-' + display['text']
22
23
24 def gleich_press():
25     display['text'] = str(eval(display['text']))
26
27
28 # Eventhandler für Enter-Taste
29 def enter_press(ereignis):
30     gleich_press()
31
32
33 # Fenster der Anwendung
34 win = Tk()
35 win['background'] = '#000000'
36 win.title('Taschenrechner')
37 win.geometry('268x470')
38 win.resizable(height = False, width = False)
39
40 # Schriftarten für Buttons und Display
41 ziffern_schrift = Font(family = 'Arial', size = 18)
42 display_schrift = Font(family = 'Arial', size = 24, weight =
43 'bold')
44
45 # Erzeugung des Displays
46 display = Label(text = '', background = '#000000',
47                 foreground =
48                 '#00FF00')
```

## 22.2 · Grafische Benutzeroberflächen mit tkinter

```
49 display['width'] = 13
50 display['font'] = display_schrift
51 display['height'] = 2
52 display['anchor'] = 'e'
53
54 # Erzeugen der Buttons
55 loeschen = Button(win, text = 'Löschen', width = 9,
56                     height = 1,
57                     font = ziffern_schrift, foreground =
58                     '#FFFFFF',
59                     background = '#4C4E4F', command =
60                     loeschen_press)
61 plusminus = Button(win, text = '+/-', width = 4, height = 1,
62                     font = ziffern_schrift, foreground =
63                     '#FFFFFF',
64                     background = '#4C4E4F', command =
65                     plusminus_press)
66 kopieren = Button(win, text = 'Kop.', width = 4, height = 1,
67                     font = ziffern_schrift, foreground =
68                     '#FFFFFF',
69                     background = '#4C4E4F', command =
70                     kopieren_press)
71 ziffer1 = Button(win, text = '1', width = 4, height = 2,
72                     font = ziffern_schrift,
73                     command = partial(ziffer_operator_press,
74                     '1'))
75 ziffer2 = Button(win, text = '2', width = 4, height = 2,
76                     font = ziffern_schrift,
77                     command = partial(ziffer_operator_press,
78                     '2'))
79 ziffer3 = Button(win, text = '3', width = 4, height = 2,
80                     font = ziffern_schrift,
81                     command = partial(ziffer_operator_press,
82                     '3'))
83 ziffer4 = Button(win, text = '4', width = 4, height = 2,
84                     font = ziffern_schrift,
85                     command = partial(ziffer_operator_press,
86                     '4'))
87 ziffer5 = Button(win, text = '5', width = 4, height = 2,
88                     font = ziffern_schrift,
89                     command = partial(ziffer_operator_press,
90                     '5'))
91 ziffer6 = Button(win, text = '6', width = 4, height = 2,
92                     font = ziffern_schrift,
93                     command = partial(ziffer_operator_press,
94                     '6'))
95 ziffer7 = Button(win, text = '7', width = 4, height = 2,
96                     font = ziffern_schrift,
97                     command = partial(ziffer_operator_press,
98                     '7'))
```

```

99 ziffer8 = Button(win, text = '8', width = 4, height = 2,
100                 font = ziffern_schrift,
101                 command = partial(ziffer_operator_press,
102                               '8'))
103 ziffer9 = Button(win, text = '9', width = 4, height = 2,
104                 font = ziffern_schrift,
105                 command = partial(ziffer_operator_press,
106                               '9'))
107 ziffer0 = Button(win, text = '0', width = 9, height = 2,
108                 font = ziffern_schrift,
109                 command = partial(ziffer_operator_press,
110                               '0'))
111 geteilt = Button(win, text = '/', width = 4, height = 2,
112                 font = ziffern_schrift,
113                 foreground = '#FFFFFF',
114                 background = '#10a605',
115                 command = partial(ziffer_operator_press,
116                               '/'))
117 mal = Button(win, text = '*', width = 4, height = 2,
118                 font = ziffern_schrift, foreground = '#FFFFFF',
119                 background = '#10a605',
120                 command = partial(ziffer_operator_press,
121                               '*'))
122 minus = Button(win, text = '-', width = 4, height = 2,
123                 font = ziffern_schrift,
124                 foreground = '#FFFFFF',
125                 background = '#10a605',
126                 command = partial(ziffer_operator_press,
127                               '-'))
128 plus = Button(win, text = '+', width = 4, height = 2,
129                 font = ziffern_schrift,
130                 foreground = '#FFFFFF',
131                 background = '#10a605',
132                 command = partial(ziffer_operator_press,
133                               '+'))
134 komma = Button(win, text = ',', width = 4, height = 2,
135                 font = ziffern_schrift,
136                 command = partial(ziffer_operator_press,
137                               '.'))
138 gleich = Button(win, text = '=', width = 10, height = 1,
139                 font = ziffern_schrift,
140                 foreground = '#FFFFFF',
141                 background = '#0570A6',
142                 command = gleich_press)
143
144 # Event-Handler für Enter-Taste festlegen
145 win.bind('<Return>', enter_press)
146
147 # Platzieren der Buttons auf der Oberfläche
148 display.grid(row = 0, column = 0, columnspan = 5,
149               sticky = 'news')
150 loeschen.grid(row = 1, column = 0, columnspan = 2,
151               sticky = 'news')

```

```

152 plusminus.grid(row = 1, column = 2, sticky = 'news')
153 kopieren.grid(row = 1, column = 4, sticky = 'news')
154 ziffer1.grid(row = 2, column = 0, sticky = 'news')
155 ziffer2.grid(row = 2, column = 1, sticky = 'news')
156 ziffer3.grid(row = 2, column = 2, sticky = 'news')
157 ziffer4.grid(row = 3, column = 0, sticky = 'news')
158 ziffer5.grid(row = 3, column = 1, sticky = 'news')
159 ziffer6.grid(row = 3, column = 2, sticky = 'news')
160 ziffer7.grid(row = 4, column = 0, sticky = 'news')
161 ziffer8.grid(row = 4, column = 1, sticky = 'news')
162 ziffer9.grid(row = 4, column = 2, sticky = 'news')
163 ziffer0.grid(row = 5, column = 0, columnspan = 2,
164             sticky = 'news')
165 komma.grid(row = 5, column = 2, sticky = 'news')
166 geteilt.grid(row = 2, column = 4, sticky = 'news')
167 mal.grid(row = 3, column = 4, sticky = 'news')
168 minus.grid(row = 4, column = 4, sticky = 'news')
169 plus.grid(row = 5, column = 4, sticky = 'news')
170 gleich.grid(row = 6, column = 0, columnspan = 5,
171               sticky = 'news')
172
173 # Ereignisschleife
174 win.mainloop()

```

### ■■ Zeilen 1–3: Importanweisungen.

An Steuerelementen benötigen wir nicht mehr als Buttons, Labels und natürlich das Anwendungsfenster selbst. Daher werden die Klassen Button, Label und Tk aus dem **tkinter**-Modul importiert. Darüber hinaus soll das „Display“ unseres Taschenrechners eine spezielle Schrift verwenden, weshalb wir aus dem **font**-Modul die Klasse **Font** importieren. Die letzte Importanweisung benötigen wir, um die Ereignisfunktionen, die auf die Zifferneingaben des Benutzers reagieren sollen, etwas einfacher zu gestalten. Dazu in Kürze mehr.

### ■■ Zeilen 6–30: Event-Handler-Funktionen für Buttons.

Dies sind die Ereignisfunktionen, die auf die unterschiedlichen Benutzeraktionen reagieren. Diese Funktionen schauen wir uns am Ende etwas genauer an, wollen uns aber zunächst mit der Oberfläche beschäftigen. Mit Kenntnis der Oberfläche ist es auch leichter, die Funktionsweise der Event Handler zu verstehen.

### ■■ Zeilen 33–40: Fenster der Anwendung.

Unser Taschenrechner soll einen schwarzen Hintergrund sowie eine feste und nicht durch den Benutzer änderbare Fenstergröße haben.

### ■■ Zeilen 45–52: Schriftarten und Display.

Als nächstes definieren wir zwei Schriftarten, **ziffern\_schrift** für die Ziffern auf den Taschenrechner-Tasten, **display\_schrift** für die Anzeige im Display des Taschenrechners. Das Display selbst bekommt eine grüne Vordergrund- und eine schwarze

Hintergrundfarbe. Mit `display['anchor'] = 'e'` richten wir seinen Inhalt „östlich“ (`e` = `east`), das heißt rechtsbündig aus.

### ■■ Zeilen 54–142: Erzeugen der Buttons.

Als nächstes werden die Button für unser Fenster `win` angelegt, die durch eine Reihe von Eigenschaften charakterisiert sind, nämlich durch ihre Beschriftung (`text`), ihre Breite und Höhe in Textzeichen (`width` und `height`), die Schriftart (`font`), die Vordergrund- und Hintergrundfarbe (`foreground` und `background`) sowie den Event Handler (`command`), der angesprochen wird, wenn der Benutzer auf den Button klickt.

Bei den Event Handlern bedienen wir uns eines kleinen Tricks, um nicht einen eigenen Event Handler für jede der 10 Ziffern und jeden Rechenoperator schreiben zu müssen. Wir haben nur einen Event Handler namens `ziffer_operator_press()` definiert, die wir mit einem Argument aufrufen, und zwar eben der jeweiligen Ziffer oder dem jeweiligen Operator. Der Option `command` des `tkinter-Button`-Objekts muss aber ein Funktionsobjekt übergeben werden, kein Aufruf einer Funktion mit Argumenten. Deshalb erzeugen wir uns mit Hilfe der Funktion `partial` aus dem Modul `functools` ein Funktionsobjekt, in das das Argument gewissermaßen bereits „eingebacken“ ist. Da der Rückgabewert von `partial()` ein dem Funktionsobjekt nicht unähnliches Objekt ist, das aber eben das Argument bereits enthält, können wir eben diesen Rückgabewert als Wert für die `command`-Option unserer Buttons verwenden.

### ■■ Zeilen 144–145: Event Handler für Enter-Taste.

Damit man die Berechnung nicht nur durch Klicken auf den Gleichheitszeichen-Button auf der Oberfläche, sondern auch dadurch auslösen kann, dass man auf der Tastatur <ENTER> drückt, binden wir an das Ereignis des <ENTER>-Drückens einen Event Handler (Zeilen 29–30), der nichts anderes tut, als denjenigen Event Handler aufzurufen, der bei Drücken des Gleichheitszeichen-Button ausgelöst wird (Zeilen 25–26). Trennen müssen wir die beiden Event Handler dennoch, denn der Event Handler, der den Tastendruck verarbeitet, bekommt als Argument standardmäßig ein Ereignis-Objekt übergeben, das wir entgegennehmen müssen, auch, wenn wir es gar nicht verarbeiten.

Werfen wir in diesem Zusammenhang einen kurzen Blick auf die übrigen Event Handler:

- **`ziffer_operator_press()`** (Zeilen 8–9): Dem Event Handler wird die angeklickte Ziffer bzw. der angeklickte Rechenoperator als Argument übergeben. Der Wert des gedrückten Buttons wird im Display einfach an den bestehenden Display-Inhalt angehängt.
- **`loeschen_press()`** (Zeilen 12–13): Leert den Inhalt des Displays.
- **`kopieren_press()`** (Zeilen 16–18): Leert zunächst die Zwischenablage und hängt dann an die Zwischenablage den aktuellen Inhalt des Displays an. Verwendet werden dazu die Funktion `clipboard_clear()` und `clipboard_append()`, die praktischerweise beide von der Tk-Klasse bereitgestellt werden und deshalb auch für unser Fensterobjekt `win` aufgerufen werden.
- **`plusminus_press()`** (Zeilen 21–22): Wird der Plus-/Minus-Button gedrückt, stellen wir dem aktuellen Displayinhalt einfach ein Minus voran. Strenggenommen

müsste man natürlich prüfen, ob bereits ein Minus vorhanden ist und dieses dann entfernen. Aber ebenso wie bei den übrigen Eingaben (wir prüfen ja zum Beispiel auch nicht, ob der Benutzer nicht vielleicht zwei Operatoren hintereinander eingibt) machen wir es uns an dieser Stelle einfach und setzen auf den Verstand des Anwenders.

- **gleich\_press()** (Zeilen 25–26): Dies ist der bereits angesprochene Event Handler, der dann ausgelöst wird, wenn der Benutzer das Ergebnis der Berechnung anfordert. Dabei setzen wir die Funktion `eval()` ein, die einen Python-Ausdruck auswertet und sein Ergebnis zurückliefert. In unserem Fall ist der Python-Ausdruck einfach nur die von Benutzer eingegebene Folgt von Zahlen und Operatoren, die in unserem Taschenrechner-Display angezeigt wird. Mit `eval()` kann aber auch ganz beliebiger Python-Code, der als String-Argument übergeben wird, ausgeführt werden.

#### ■ ■ Zeilen 147–163: Platzieren der Buttons auf der Oberfläche.

Jetzt müssen das Display und die Buttons nur noch auf der Oberfläche platziert werden. Dazu verwenden wir den **Grid**-Geometry-Manager; wir weisen also mit der Methode `grid()` den einzelnen Bedienelemente ihren Platz im Raster des Fens-ters zu, indem wir ihre Zeilen- (**row**) und Spaltennummer (**column**) festlegen. Mit der Option **columnspan** erreichen wir, dass einige Elemente sich horizontal über mehr als nur eine Zelle des Rasters erstrecken, zum Beispiel das Display. Mit der Wert **news** (*north+east+west+south*) der Option **sticky** legen wir fest, dass sich die Elemente in ihrer jeweiligen Rasterzelle vollständig ausdehnen, die Zelle also vollkommen ausfüllen sollen.

#### ■ ■ Zeilen 163–153: Ereignisschleife.

Mit `mainloop()` starten wir die Ereignisverarbeitung für unseren Taschenrechner. Ab sofort reagiert also die Benutzeroberfläche auf die Eingaben des Anwenders indem sie die entsprechenden Event Handler aufruft.

## 22.3 Arbeiten mit Dateien

---

Die Arbeit mit Dateien ist in Python sehr einfach. Sie läuft – nicht anders als in den meisten anderen Programmiersprachen auch – in drei Schritten ab:

6. Die Datei wird geöffnet (ggf. dabei überhaupt erst erzeugt).
7. Die Datei wird bearbeitet (es wird aus ihr gelesen, in sie geschrieben oder an sie angehängt).
8. Die Datei wird nach Abschluss aller Arbeiten geschlossen.

#### ■ Dateien öffnen

Eine Datei wird dabei durch ein *Dateiobjekt* repräsentiert. Ein solches erzeugen wir mit Hilfe der in Python eingebauten Standard-Funktion `open(dateipfad, modus)`. Das Argument **modus** beschreibt dabei den Bearbeitungsmodus, in dem die Datei geöffnet werden soll. Mögliche Ausprägungen des Modus sind:

- "w": Die Datei wird zum *Schreiben* (*write*) geöffnet. Der Dateizeiger wird dazu an den Beginn der Datei gesetzt. Ein möglicherweise vorhandener Inhalt der Datei wird vollständig ersetzt. Wenn die Datei bislang noch nicht existiert, wird sie neu erzeugt. Ein Lesen aus der Datei in diesem Modus ist nicht möglich.
- "a": Die Datei wird zum *Anhängen* (*append*) geöffnet. Der Dateizeiger wird dazu an das Ende der Datei gestellt. Inhalte, die in die Datei geschrieben werden, werden ihr dadurch angehängt. Ein Lesen aus der Datei ist in diesem Modus nicht möglich.
- "r": Die Datei wird zum *Lesen* (*read*) geöffnet. Der Dateizeiger wird dazu an den Beginn der Datei gesetzt. Ein Schreiben in die Datei ist in diesem Modus nicht möglich.
- "r+": Die Datei wird zum Lesen *und* Schreiben geöffnet.

Beachten Sie bitte, dass Sie, wenn Sie auf einem Windows-System arbeiten, die Backslashes, die in der Pfadangabe die einzelnen Pfadbestandteile voneinander trennen, mit einem weiteren Backslash *escapen* müssen, weil Python sie sonst als den Versuch betrachtet, das *folgende Zeichen* zu escapen und ihm damit eine besondere Steuerungsfunktion zuzuweisen (wenn Sie mit dem Escapen nicht mehr vertraut sind, blättern Sie nochmal einige Seiten zurück zu Abschn. 11.2.2).

Wollten wir nun zum Beispiel die Datei **test.txt** im Verzeichnis **C:\Programmieren** zum Schreiben öffnen, würden wir uns zunächst ein entsprechendes Dateiobjekt (das wir hier der Einfachheit halber **datei** nennen) von der Funktion **open()** erzeugen lassen:

```
datei = open("C:\\Programmieren\\test.txt", "w")
```

Dieses Objekt verfügt über eine Reihe von Eigenschaften, mit denen wir seinen Charakter besser verstehen können: **datei.name** liefert uns den Dateinamen als vollständige Pfadangabe, **datei.mode** den Modus, in dem wir die Datei geöffnet haben. Ob die Datei lesbar und/oder schreibbar ist, lässt sich darüber hinaus mit den Methoden **datei.readable()** und **datei.writable()** feststellen, die jeweils einen **bool**-Wert zurückgeben.

### ■ Bearbeiten der Datei

Um nun in die Datei zu *schreiben*, stellt das Dateiobjekt die Methoden **write(text)** und **writelines(zeilen)** zur Verfügung.

**write()** schreibt lediglich *eine* Zeichenkette in die Datei, und zwar ohne abschließenden Zeilenumbruch, es sei denn, **text** enthielte am Ende die Escape-Sequenz **\n** (beispielsweise "**Das ist ein Text mit Zeilenumbruch\n**").

**writelines()** hingegen schreibt *mehrere* Zeichenketten, die als Array übergeben werden; der Name der Funktion ist dabei etwas missverständlich, denn auch **writelines()** schreibt keinen Zeilenumbruch ans Ende jeder Zeichenkette. Wollen Sie also einen Zeilenumbruch nach jeder Zeichenkette erreichen, müssen Sie ihn selbst hinzufügen:

```
zeilen = ["Zeile 1\n", "Zeile 2\n"]
datei.writelines(zeilen)
```

Die Schreib-Methoden geben jeweils die Zahl der geschriebenen Zeichen als Funktionswert zurück.

Zum *Lesen* aus einer Datei stehen die Funktionen **read()**, **readline()** und **readlines()** zur Verfügung. **read()** liest den *gesamten* Dateiinhalt und gibt ihn als String zurück. Mit einem optionalen Argument kann eine *bestimmte Zahl von Zeichen* (gemessen ab der aktuellen Position des Dateizeigers) gemessen werden. Der Dateizeiger steht dabei zu Beginn am Anfang der Datei und rückt bei jedem Lesevorgang entsprechend weiter. Betrachten Sie dazu die folgende Beispiel-Datei:

```
Zeile Nummer eins
Noch eine Zeile
Letzte Zeile
```

Mit **read(3)** würden wir nach dem Öffnen der Datei (der Dateizeiger steht dann am Anfang der Datei), zunächst die Zeichenkette "Zei" einlesen. Danach steht der Dateizeiger auf dem "I" von "Zeile". Ein weiteres **read(19)** würde dann die nächsten 19 Zeichen liefern, also "le Nummer eins\nNoch". Beachten Sie bitte, dass der Zeilenumbruch auch als Zeichen zählt und zwar als *genau ein* Zeichen (obwohl er als Escape-Sequenz in der Form \n mit zwei Zeichen dargestellt wird). Nach diesem erneuten Lesevorgang steht der Dateizeiger nun auf dem Leerzeichen zwischen "Noch" und "eine".

Anders als **read()** gehen die Funktionen **readline()** und **readlines()** vor; sie lesen jeweils eine bzw. mehrere *Zeilen* ein. **readline()** liest dabei immer genau die nächste Zeile ein, **readlines()** dagegen liest *alle* Zeilen oder die als optionales Argument übergebene Zahl von Zeilen ein und gibt dabei ein *Array von Zeichenketten* zurück. Die Methodenaufrufe **readline()** und **readlines(1)** unterscheiden sich also dadurch, dass das Ergebnis des **readline()**-Aufrufs in einer **str**-Wert resultiert, während **readlines()** ein Array liefert, das in diesem Fall nur eine einzelne Zeichenkette als Element enthält.

Wenn der Dateizeiger nicht am Anfang einer Zeile steht, sondern mitten in einer Zeile, wie nach dem Aufruf von **read(3)** oben, dann lesen **readline()** und **readlines()** ab dem Zeichen, auf dem aktuell der Dateizeiger steht. Der Beginn der Zeile (in unserem Beispiel die ersten drei bereits mit **read()** eingelesenen Zeichen) wird dann nicht mehr gelesen.

Beim Lesen können Sie den Dateizeiger mit **seek(zeichenindex)** auf das durch **zeichenindex** angegebene Zeichen setzen, gezählt vom Dateianfang. Dabei trägt das erste Zeichen der Datei den Index **0**. Die aktuelle Position des Dateizeigers liefert die Methode **tell()** des Dateiobjekts.

Übrigens: Wenn Sie die Datei im Modus "**r+**" (Lesen *und* Schreiben) öffnen, können Sie zwar tatsächlich dasselbe File-Objekt für beide Operationen verwenden. Das Schreiben erfolgt allerdings immer am Ende der Datei, das Lesen an der

aktuellen Position des Dateizeigers, der sich genauso verhält wie beim Öffnen der Datei im Modus "r".

### ■ Die Datei schließen

Nach dem Bearbeiten der Datei, schließen Sie sie mit der Methode `datei.close()`. Das Dateiobjekt existiert danach weiter, seine Eigenschaft `datei.closed` nimmt jetzt den Wert **True** an und signalisiert auf diesen Weise, dass das Objekt nicht zum Lesen oder Schreiben zur Verfügung steht. Da das Dateiobjekt aber immer noch den Pfad zu der Datei als Eigenschaft **name** mit sich führt, können Sie es einfach „reaktivieren“:

```
datei = open(datei.name, "r")
```

### ?

#### 22.2 [15 min]

Schreiben Sie ein Programm, das vom Benutzer den Namen einer Datei (vollständige Pfadangabe) abfragt. Vom Dateinhalt soll dann ein gewisser Prozentsatz als Vorschau angezeigt werden. Der Prozentsatz soll dabei vom Benutzer angegeben werden. Zeilenumbrüche sollen bei der Vorschau-Anzeige entfernt werden, damit die Darstellung möglichst kompakt ist.

## 22.4 Übung: Entwicklung eines einfachen Text-Editors

---

Die folgende Übung, für die Sie sich etwas mehr Zeit und Ruhe nehmen sollten, kombiniert viele der Dinge, die Sie in diesem Kapitel gelernt haben, um eine nützliche kleine Anwendung zu entwickeln.

Die Aufgabe besteht darin, mit **tkinter** einen einfachen Text-Editor zu programmieren. Dieser soll es erlauben, Dateien neu anzulegen oder bestehende Dateien zu öffnen, und die Dateien dann zu bearbeiten und wieder zu speichern, entweder unter dem aktuellen oder einem neuen Namen. Auch soll der Benutzer Text in die Zwischenablage kopieren und aus der Zwischenablage wieder einfügen können.

Die Befehle des Editors sollen sowohl über ein Menü als auch über eine Button-Leiste auswählbar sein.

Testen Sie Ihr Programm ausgiebig!

Die geschätzte Zeit für die Bearbeitung dieser Aufgabe beträgt 120 min.

Wenn Ihnen diese Aufgabe noch zu herausfordernd erscheint, entwickeln Sie den Editor nicht selbst, sondern lesen Sie den Code in der Musterlösung und versuchen Sie, diesen zu verstehen, zunächst, ohne dabei auf die erläuternden Hinweise in der Lösung zurückzugreifen.

## 22.5 Zusammenfassung

---

In diesem Kapitel haben wir uns damit beschäftigt, wie Daten über die Konsole ein- und ausgegeben werden können. Außerdem haben wir uns angesehen, wie in Python grafische Benutzeroberflächen realisiert werden können, die dem Benutzer eine bequeme Interaktion mit Ihrem Programm ermöglichen.

Folgende Punkte sollten Sie aus diesem Kapitel unbedingt mitnehmen:

- In der Python-Konsole können Sie Objekte stets mit der eingebauten Python-Funktion **print(Objekt)** ausgeben.
- Vom Benutzer können Informationen mit Hilfe der Methode **input(prompt)** abgefragt werden, die die Benutzereingabe stets als String zurückgibt (der Benutzerinput muss also nötigenfalls konvertiert werden).
- Grafische Benutzeroberflächen (GUIs) lassen sich mit der **tkinter**-Bibliothek, die zum Standardlieferumfang von Python gehört, leicht umsetzen.
- Ein **tkinter**-Programm setzt sich immer zusammen aus der Erzeugung eines Tk-Objekts mit Hilfe der gleichnamigen Konstruktorfunktion, dem Erzeugen und Konfigurieren der Bedienelemente (Widgets), der Festlegung der Anordnung der Bedienelemente (mit einem Geometry-Manager) und dem Starten der Ereignisverarbeitung (Methode **mainloop()** des verwendeten Tk-Objekts).
- Die wichtigsten Steuerelemente (Widgets) für grafische Benutzeroberflächen in **tkinter** sind **Button**, **Menu**, **Entry** (Texteingabe), **Label** (Textanzeige), **Checkbutton** (Mehrfachauswahl von Optionen), **Radiobutton** (Einfachauswahl von Optionen) und **Listbox** (listenartige Darstellung von Texteinträgen mit Einfach- oder Mehrfachauswahl).
- Wichtige Standarddialoge, die aus **tkinter** heraus verwendet werden können (**tkinter**-Modul **filedialog**), sind zur Darstellung von Textmeldungen **messagebox** (mit diversen Varianten, die sich in den angezeigten Icons und Buttons unterscheiden), sowie zur Abfrage von Dateipfaden beim Öffnen oder Speichern **askopenfilename()** und **asksaveasfilename()**.
- Die Widgets werden über Optionen konfiguriert; einige Optionen (nicht aber deren Werte!) sind fast allen Widgets gemeinsam (zum Beispiel die Hintergrundfarbe **background** und die Schriftart **font**), andere sind spezifisch für das jeweilige Steuerelement.
- Alle Widgets besitzen die Methode **config(option = wert, ...)**, mit der die Werte der Optionen eingestellt werden können; zusätzlich kann auf die Optionen in der Form **widget['option']** wie auf ein Dictionary zugegriffen werden.
- Widgets werden auf der Programm-Oberfläche mit Hilfe eines Geometry-Managers angeordnet; **tkinter** kennt mit **Pack** (direkt nebeneinander/untereinander anordnen), **Grid** (entlang eines gedachten Rasters anordnen) und **Place** (durch Angabe von Koordinaten relativ zu einem Bezugspunkt anordnen) drei solcher Anordnungswerkzeuge, die mit den Standard-Methoden **pack()**, **grid()** und **place()** jedes Widgets aufgerufen werden können..

- Zum Lesen und Schreiben von Daten aus bzw. in Dateien wird die betreffende Datei zunächst mit der eingebauten Python-Funktion **open(dateiname, modus)** geöffnet; diese gibt ein **File**-Objekt zurück.
- Modi zum Bearbeiten von Dateien sind **r** (Lesen), **w** (Schreiben), **a** (Anhängen) und **r+** (Lesen und Schreiben).
- Mit den Methoden **read()** und **readlines()** sowie **write()** und **writelines()** des File-Objekts kann aus der Datei gelesen bzw. in die Datei geschrieben werden.
- Die Methode **close()** des File-Objekts schließt die Datei nach Abschluss der Bearbeitung wieder.

## 22.6 Lösungen zu den Aufgaben

### ■ Aufgabe 22.1

```
# Erste Möglichkeit: Drei print()-Anweisungen (jede wird
# automatisch mit end = '\n' abgeschlossen.
print('Erste Zeile')
print('Zweite Zeile')
print('Dritte Zeile')

# Zweite Möglichkeit: Ein String, Zeilen darin durch
# Escape-Sequenz \n trennen.
print('Erste Zeile\nZweite Zeile\nDritte Zeile')

# Dritte Möglichkeit: Ausgabe von drei einzelnen String-Objekten,
# dabei die Escape-Sequenz \n als Separator angeben.
print('Erste Zeile', 'Zweite Zeile', 'Dritte Zeile',
      sep = '\n')
```

### ■ Aufgabe 22.2

Das Programm könnte so aussehen:

```
datei_name = input(
    "Bitte geben Sie einen Dateinamen (mit Pfad) ein: ")
prozent = input(
    "Prazentsatz für Vorschau (ganze Zahl, z.B. 10 für 10%): ")

datei = open(datei_name, "r")
inhalt = datei.read()
datei.close()

inhalt = inhalt.replace("\n", "")
laenge_gesamt = len(inhalt)
laenge_preview = int(laenge_gesamt * int(prozent) / 100)
print("### Vorschau: ", laenge_preview, " Zeichen von ",
      laenge_gesamt, "Zeichen ###")
print(inhalt[0:laenge_preview], "\n####\n")
```

## 22.6 · Lösungen zu den Aufgaben

Die Datei wird zunächst im Lesemodus ("r" geöffnet) und ihr gesamter Inhalt mit `read()` ausgelesen. Danach kann die Datei auch schon wieder geschlossen werden, denn in der String-Variable `inhalt` befindet sich nun der gesamte Dateiinhalt und nur mit diesem arbeiten wir weiter. Nachdem wir den Inhalt um Zeilenumbrüche bereinigt haben, indem wir mit der String-Methode `replace()` die Escape-Sequenz `\n` entfernen, selektieren wir in der letzten Anweisung die erforderlich Zahl von Zeichen, die wir zuvor aus dem vom Benutzer angegebenen Vorschau-Prozentsatz errechnet haben und geben sie auf dem Bildschirm aus. Beim Selektieren der Zeichen aus dem String ist es wichtig, sicherzustellen, dass die Selektionsgrenzen ganzzahlig sind. Das erreichen wir, indem wir bei der Berechnung der Vorschau-Länge das Ergebnis mit `int()` als Integer-Variable speichern.

### ■ Programmieraufgabe Text-Editor

Bei der Entwicklung des Text-Editors sind natürlich viele Varianten möglich. Die Benutzeroberfläche der hier vorgestellten Lösung sehen Sie in Abb. 22.10.

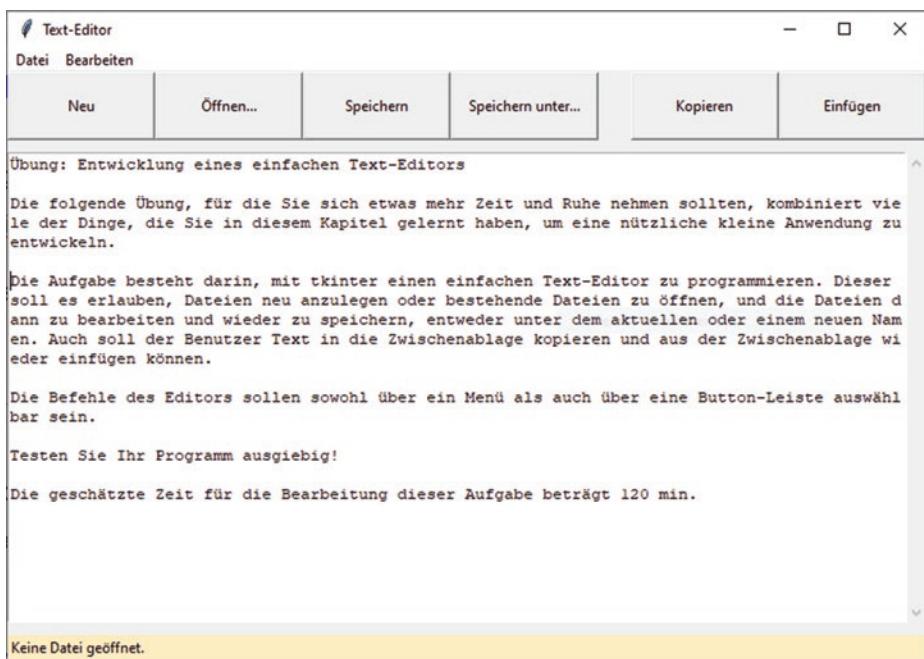


Abb. 22.10 Benutzeroberfläche unseres Text-Editors

Deren Code sieht folgendermaßen aus:

```

1  from tkinter import Tk, Button, scrolledtext, Text, Label, \
2      Menu
3  from tkinter.filedialog import askopenfilename, \
4      asksaveasfilename
5
6
7  # Eventhandler-Funktionen für Buttons und Menüs
8
9  def neu_press():
10     global dateiname
11     text.delete(1.0, 'end')
12     dateiname = ''
13     status['text'] = 'Ungespeicherte neue Datei'
14
15
16 def oeffnen_press():
17     global dateiname
18     dname = askopenfilename(defaultextension = 'txt',
19                             filetypes = [
20                                 ('Textdateien', '*.txt'),
21                                 ('Alle Dateien', '*.*'), ],
22                                 title = 'Datei öffnen....',
23                                 initialdir = 'C:\\\\Windows')
24     datei = open(dname, 'r')
25     text.delete(1.0, 'end')
26     text.insert(1.0, datei.read())
27     datei.close()
28     status['text'] = 'Datei "' + dname + '" geöffnet.'
29     dateiname = dname
30
31
32 def speichernunter_press():
33     global dateiname
34     dname = asksaveasfilename(defaultextension = 'txt',
35                               filetypes = [
36                                   ('Textdateien', '*.txt'),
37                                   (
38                                       'Alle Dateien', '*.*'), ],
39                                   title = 'Datei speichern unter...',
40                                   initialdir = 'C:\\\\Windows')
41     datei = open(dname, 'w')
42     datei.write(text.get(1.0, 'end'))
43     datei.close()
44     status['text'] = 'Datei "' + dname + '" gespeichert.'
45     dateiname = dname
46
47
48 def speichern_press():
49     global dateiname
50     datei = open(dateiname, 'w')
51     datei.write(text.get(1.0, 'end'))

```

## 22.6 · Lösungen zu den Aufgaben

```
52     datei.close()
53     status[
54         'text'] = 'Datei "' + dateiname + '" gespeichert.'
55
56
57 def kopieren_press():
58     auswahl = text.selection_get()
59     text.clipboard_clear()
60     text.clipboard_append(auswahl)
61
62
63 def einfuegen_press():
64     text.insert(text.index('insert'), text.clipboard_get())
65
66
67 def kopieren_press_taste(ereignis):
68     kopieren_press()
69
70
71 def einfuegen_press_taste(ereignis):
72     einfuegen_press()
73
74
75 def beenden_press():
76     win.quit()
77
78
79 # Fenster der Anwendung
80
81 win = Tk()
82
83 win.title('Text-Editor')
84 win.geometry('760x490')
85 win.resizable(height = True, width = True)
86
87
88 # Einrichtung des Menüs
89
90 menuleiste = Menu(win)
91 win.config(menu = menuleiste)
92
93 dateimenu = Menu(menuleiste, tearoff = 0)
94 bearbeitenmenu = Menu(menuleiste, tearoff = 0)
95
96 menuleiste.add_cascade(label = 'Datei', menu = dateimenu)
97 menuleiste.add_cascade(label = 'Bearbeiten',
98                         menu = bearbeitenmenu)
99
100 dateimenu.add_command(label = 'Neu',
101                       command = neu_press)
102 dateimenu.add_command(label = 'Öffnen...', 
103                       command = oeffnen_press)
104 dateimenu.add_command(label = 'Speichern',
105                       command = speichern_press)
```

```
106 dateimenu.add_command(label = 'Speichern unter...',  
107                         command = speichernunter_press)  
108 dateimenu.add_separator()  
109 dateimenu.add_command(label = 'Beenden',  
110                         command = beenden_press)  
111  
112 bearbeitenmenu.add_command(label = 'Kopieren',  
113                         command = kopieren_press)  
114 bearbeitenmenu.add_command(label = 'Einfügen',  
115                         command = einfuegen_press)  
116  
117  
118 # Erzeugen der Steuerelemente  
119  
120 neu = Button(win, text = 'Neu', height = 3, width = 16,  
121                         command = neu_press)  
122 oeffnen = Button(win, text = 'Öffnen...', height = 3,  
123                         width = 16,  
124                         command = oeffnen_press)  
125 speichern = Button(win, text = 'Speichern', height = 3,  
126                         width = 16,  
127                         command = speichern_press)  
128 speichernunter = Button(win, text = 'Speichern unter...',  
129                         height = 3,  
130                         width = 16,  
131                         command = speichernunter_press)  
132 seplabel = Label(win, text = '', height = 3, width = 3)  
133 kopieren = Button(win, text = 'Kopieren', height = 3,  
134                         width = 16,  
135                         command = kopieren_press)  
136 einfuegen = Button(win, text = 'Einfügen', height = 3,  
137                         width = 16,  
138                         command = einfuegen_press)  
139  
140 text = scrolledtext.ScrolledText(win)  
141 text.bind('<Control-k>', kopieren_press_taste)  
142 text.bind('<Control-e>', einfuegen_press_taste)  
143  
144  
145 # Vorbereitung der Statusleiste  
146  
147 status = Label(win, text = 'Keine Datei geöffnet.',  
148                         anchor = 'w',  
149                         background = '#FFEFC4')  
150 dateiname = ''  
151  
152  
153 # Platzieren der Steuerelemente auf der Oberfläche  
154  
155 neu.grid(row = 0, column = 0, sticky = 'news')  
156 oeffnen.grid(row = 0, column = 1, sticky = 'news')
```

```
157 speichern.grid(row = 0, column = 2, sticky = 'news')
158 speichernunter.grid(row = 0, column = 3, sticky = 'news')
159
160 seplabel.grid(row = 0, column = 4, sticky = 'news')
161 kopieren.grid(row = 0, column = 5, sticky = 'news')
162 einfuegen.grid(row = 0, column = 6, sticky = 'news')
163
164 text.grid(row = 1, column = 0, columnspan = 7, pady = 10,
165             sticky = 'news')
166
167 status.grid(row = 2, column = 0, columnspan = 7,
168               sticky = 'news')
169
170
171 # Ereignisschleife
172
173 win.mainloop()
```

#### ■■ Zeilen 5–73. Event-Handler-Funktionen für Buttons und Menüs.

Die Funktionsweise der Event Handler schauen wir uns weiter unten genauer an, wenn klar geworden ist, aus welchen Komponenten sich die Oberfläche zusammensetzen wird.

#### ■■ Zeilen 79–85. Fenster der Anwendung.

Das Fenster **win** wird als Tk-Objekt erzeugt und skaliert. Es soll in seiner Größe für den Benutzer veränderbar sein.

#### ■■ Zeilen 87–115. Einrichtung des Menüs.

Eine neue Menüleiste wird für das **win**-Fenster erzeugt und zwei Drop-Down-Menüs, **dateimenmu** und **bearbeitenmenu** auf der Leiste platziert. Den Menüs werden dann nach und nach die Menüeinträge hinzugefügt.

#### ■■ Zeilen 118–142. Erzeugen der übrigen Steuerelemente.

Hier werden die Buttons sowie das Texteingabefeld erzeugt. Außerdem werden Event Handler an zwei Ereignisse des Texteingabefelds gebunden, um das Drücken der Tastenkombinationen <CTRL>+<K> (Kopieren) und <CTRL>+<E> (Einfügen) zu verarbeiten.

#### ■■ Zeilen 145–150. Vorbereitung der Statusleiste.

Wir erzeugen ein Label als gelblich eingefärbte Statusleiste, auf der dann der Name der aktuell geöffneten Datei angezeigt wird.

■■ Zeilen 153–168. Platzieren der Steuerelemente auf der Oberfläche.

Die Steuerelemente werden mit dem **Grid**-Geometry-Manager auf der Benutzeroberfläche angeordnet, und zwar so, dass sie ihre jeweiligen „Rasterzellen“ voll ausfüllen (**sticky** = 'news', also *north east west south*).

■■ Zeilen 171–183. Ereignisschleife.

**mainloop()** startet die Ereignisverarbeitung für unseren Editor. Damit reagiert die Benutzeroberfläche auf die Eingaben des Anwenders.

■■ Event Handler (Zeilen 7–76).

- **neu\_press()**: Erzeugt eine Datei, indem zunächst der Inhalt des **ScrolledText**-Felds gelöscht wird; gelöscht wird von Zeile 1 (Zeilenummerierung beginnt bei den Text-Widgets in **tkinter** bei 1!), Spalte 0 (Spaltennummerierung beginnt bei 0) bis zum Ende. Statt des Strings 'end' hätte auch die Konstante **END** verwendet werden können, wenn das Modul **constants** mit importiert worden wäre. Dann hätte man mit **constants.END** auf diese Konstante zugreifen können, oder aber, wenn man mit **from tkinter import \*** importiert hätte, sogar nur mit **END**.

Die weiter unten (Zeile 139) erstmals verwendete globale Variable **dateiname**, auf die wir uns den Zugriff mit der Anweisung **global dateiname** sichern (ansonsten würde innerhalb von **neu\_press()** eine gleichnamige *lokale* Variable erzeugt!) wird neu initialisiert und der Text der Statusleiste geupdatet.

- **oeffnen\_press()**: Hier nutzen wir die Funktion **askopenfilename()**, um den Pfad zu der zu öffnenden Datei abzufragen. Danach wird die Datei mit **open()** im Lesemodus geöffnet und der aus ihr mit **read()** herausgelesene Inhalt in unser Text-Feld geschrieben, bevor die Datei mit **close()** wieder geschlossen wird.
- **speichernunter\_press()** und **speichern\_press()**: Beim Speichern gehen wir grundsätzlich ähnlich vor wie beim Öffnen; die Datei wird allerdings im Schreibmodus geöffnet, damit die Methode **write()** den Inhalt des Text-Feld, den wir uns mit der Methode **get()** des **ScrolledText**-Widgets besorgen, auch hineinschreiben kann. Bei „Speichern unter“ fragen wir den Dateinamen mit Hilfe der **tkinter**-Funktion **asksaveasfilename()** ab, bei **speichern\_press()** wird die Datei unter dem bereits verwendeten Namen gespeichert. Das kann also nur dann funktionieren, wenn eine Datei geöffnet oder der Inhalt des **ScrolledText**-Widgets bereits in einer Datei gespeichert wurde.
- **kopieren\_press()** und **einfuegen\_press()** sowie **kopieren\_press\_taste()** und **einfuegen\_press\_taste()**: Beim Kopieren von Text ermitteln wir mit der **selection\_get()**-Methode des **ScrolledText**-Widgets zunächst den Inhalt der Textauswahl, leeren dann die Zwischenablage mit **clipboard\_clear()** und fü-

gen den zu kopierenden Text dann mit Hilfe der Methode `clipboard_append()` in die Zwischenablage ein.

Beim Einfügen von Text bestimmen wir zunächst mit `index('insert')` die aktuelle Einfüge-, also Cursor-Position in unserem `ScrolledText`-Widget und holen uns dann den Inhalt der Zwischenablage mit `clipboard_get()`, um ihn an dieser Stelle einzufügen.

Die Funktionen `kopieren_press_taste()` und `einfuegen_press_taste()`, die wir an die Tastendruck-Ereignisse gebunden haben (Zeilen 131/132), rufen die bereits besprochenen Event Handler auf, sind aber nötig, weil den mit `bind()` an ein Ereignis gebundenen Event Handlern ein Event-Objekt als Argument übergeben wird. Diese Event Handler müssen daher ein Argument vorsehen, während die Event Handler, die wir mit Hilfe der `command`-Option an Buttons und Menüeinträge gebunden haben, keine Argumente besitzen dürfen.



# Wie arbeite ich mit Programmfunctionen, um Daten zu bearbeiten und Aktionen auszulösen?

## Inhaltsverzeichnis

- 23.1 Arbeiten mit Funktionen – 358**
  - 23.1.1 Definition von Funktionen – 358
  - 23.1.2 Funktionsargumente – 359
  - 23.1.3 Rückgabewerte – 365
  - 23.1.4 Lokale und globale Variablen – 366
- 23.2 Funktionen als Klassen-Methoden von Objekten verwenden – 370**
- 23.3 Arbeiten mit Modulen und Packages – 374**
  - 23.3.1 Programmcode modularisieren – 374
  - 23.3.2 Elemente aus Modulen importieren – 375
  - 23.3.3 Die Community nutzen – Der *Python Package Index (PyPI)* – 377
- 23.4 Zusammenfassung – 380**
- 23.5 Lösungen zu den Aufgaben – 381**

## Übersicht

Bislang haben wir uns damit beschäftigt, wie man Python-Programme zum Laufen bringt, wie man mit Variablen/Objekten arbeitet, und wie man Daten ein- und ausgibt. In diesem Kapitel konzentrieren wir uns auf das, was dazwischen passiert, nämlich die Bearbeitung der Daten. Wenn die Ein- und Ausgabe von Daten die Brötchenhälfte unseres „Programm-Burgers“ sind, dann geht es jetzt vor allem um das Fleisch in der Mitte (obwohl es natürlich auch Funktionen zur Ein- und Ausgabe gibt).

Die wichtigste Art, in Programmen Daten zu bearbeiten ist, Funktionen aufzurufen, die die Daten verändern oder andere Aktionen auslösen.

Funktionen sind so wichtig, weil wir mit ihnen bestimmte Aufgaben ausführen können, sogar ohne genau zu wissen, wie das eigentlich im Detail funktioniert. Wir rufen einfach die Funktion auf, und die Funktion tut, was sie tun soll, ohne dass wir diese Funktionalität selbst programmieren müssten und ohne, dass wir verstehen müssen, wie die Funktion arbeitet. Natürlich können wir aber eine Funktion auch selbst definieren. Die Funktion erlaubt es, eine bestimmte Funktionalität zu kapseln und diese von außen zugänglich zu machen. Mit Funktionen lagern wir letztlich Code-Teile aus dem normalen Programmcode aus und machen sie von überall her aufrufbar.

In diesem Kapitel werden Sie lernen:

- wie Sie in Python Funktionen definieren und sie aufrufen können
- wie Sie Funktionen (in der Regel als Methoden von Klassen) in Module und Packages zusammenfassen und aus diesen zur Benutzung in Ihr Programm importieren können
- wie Sie mit dem *Python Package Index (PyPI)* arbeiten, um Funktionalität, die andere Entwickler bereitstellen, in Ihr Programm einzubinden.

## 23.1 Arbeiten mit Funktionen

### 23.1.1 Definition von Funktionen

Erinnern Sie sich an unsere Taschenrechner-Applikation, die wir im letzten Kapitel mit Hilfe des **tkinter**-Package entwickelt haben (► Abschn. 22.2.6)? Dort haben wir einige eigene Funktionen definiert, zum Beispiel diese hier:

```
def loeschen_press():
    display['text'] = ''
```

Das ist eine Funktion, die wir gar nicht selbst aufrufen, sondern eine Ereignisbehandlungsfunktion, die automatisch aufgerufen wird, wenn der Benutzer den „Löschen“-Button klickt, um die aktuelle Anzeige auf dem Display(-Label) zu löschen.

An diesem einfachen Beispiel sieht man aber bereits einiges von dem, was eine Funktion ausmacht. Ihre Definition beginnt mit der Anweisung **def**, gefolgt vom Namen der Funktion. Der Name der Funktion wird mit runden Klammern versehen, die die Funktionsargumente aufnehmen. Selbst aber, wenn die Funktion gar keine Argumente besitzt, müssen die runden Klammern in der Definition (und später auch beim Aufruf) mit angeben werden.

Die **def**-Anweisung, die letztlich den *Funktionskopf* unserer Funktion darstellt, wird mit einem Doppelpunkt abgeschlossen, der andeutet, dass jetzt das folgt, was die Funktion eigentlich tut, der Programmcode, der ausgeführt wird, wenn die Funktion aufgerufen wird, kurz: der *Funktionsrumpf*.

Der Funktionsrumpf ist ein Code-Block, der wie immer dadurch gekennzeichnet wird, dass er eingerückt ist. In diesem Beispiel besteht der Code-Block nur aus einer einzigen Code-Zeile. Übrigens muss der Code-Block mindestens eine Zeile umfassen. Das Programm:

```
def meine_funktion():
    print('Hier läuft das Hauptprogramm.')
```

führt zu einer Fehlermeldung:

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

Es wird also nach der **def**-Anweisung ein eingerückter Code-Block erwartet. Manchmal weiß man bereits, dass man eine Funktion braucht, will sie aus dem Hauptprogramm heraus auch schon aufrufen, weiß aber noch nicht genau, wie die Funktion genau arbeiten soll. Was tun? Eine „leere“ Funktion zu schreiben führt zu der obigen Fehlermeldung, übrigens auch dann, wenn Sie als einzige Zeile in den Code-Block einen Kommentar hineinschreiben; der Kommentar gilt nicht als ausführbare Anweisung, die Python als Code-Block akzeptiert. Um dieses Problem zu lösen, hat Python eine spezielle Anweisung, nämlich **pass**. **pass** macht absolut gar nichts, ist aber eine ausführbare Anweisung. Schreiben Sie also **pass** als einzige Anweisung in Ihren Funktionscode, so macht das Ihre Funktion syntaktisch korrekt, ohne dass die Funktion irgendetwas tut. Beachten Sie dabei bitte, dass **pass** eine Python-Anweisung, keine Funktion ist (ebenso wie **def**) und deshalb nicht mit (leeren, runden) Klammern aufgerufen wird.

### 23.1.2 Funktionsargumente

Lassen Sie uns nochmal an ein Beispiel aus dem ersten Teil des Buches anknüpfen (aus ▶ Abschn. 12.2.2), nämlich die Temperaturumrechnung von Kelvin in Grad Celsius.

Diese Umrechnung lässt sich natürlich hervorragend als Funktion schreiben. Dieses Mal benötigt unsere Funktion aber ein Argument, nämlich die umzurechnende Temperatur:

```
def kelvin_zu_celsius(kelvin):
    print(kelvin, 'Kelvin sind', round(kelvin - 273.15, 2),
          'Kelvin.')
```

## 23

Diese Funktion könnten wir nun aufrufen, zum Beispiel mit:

```
kelvin_zu_celsius(300)
```

Dann erhielten wir als Output:

```
willkommen('Sophie', 'Willkommen', 'Schön, dass Du dabei \
bist.')
```

Natürlich können Funktion auch mehr als nur ein Argument haben. Betrachten Sie das folgende Beispiel, in dem wir eine Willkommensnachricht auf dem Bildschirm ausgeben:

```
def willkommen(name, gruss, nachricht):
    print(gruss, ', ', name, '! ', nachricht, sep = '')
```

Die Funktion könnten wir nun zum Beispiel folgendermaßen aufrufen:

```
willkommen('Sophie', 'Willkommen', 'Schön, dass Du dabei
bist.')
```

Das Ergebnis dieses Aufrufs sieht dann in der (Run-)Konsole so aus:

```
Willkommen, Sophie! Schön, dass Du dabei bist.
```

### ■ Funktions- und Schlüsselwort-Argumente

Welche der beim Funktionsaufruf angegebenen Zeichenketten dem Argument **name**, welche dem Argument **gruss**, und welche dem Argument **nachricht** zugeordnet wird, entscheidet sich anhand der Reihenfolge der Übergabe, also ihrer Position; man spricht daher auch von *Positionsargumenten*.

Wir können den Namen des Arguments beim Aufruf der Funktion aber auch angeben. Das erlaubt einen Aufruf wie den folgenden:

```
print(value, ..., sep=' ', end='\n', file=sys.stdout,  
      flush=False)
```

Dabei übergeben wir das erste Argument anhand seiner Position, die folgenden beiden aber anhand ihres Namens. Beachten Sie bitte, dass wir die Argumente **nachricht** und **gruss** in einer anderen Reihenfolge übergeben, als es der Funktionsdefinition oben entspricht. Da wir die Argumente aber explizit mit ihrem Namen ansprechen, kann Python die übergebenen Werte den Funktionsargumenten trotzdem richtig zuordnen. Der Vorteil dieser *benannten Argumente* (auch als Schlüsselwort-Argumente bezeichnet) ist also, dass die Reihenfolgen, in der wir die Argumente angeben, keine Rolle spielt, was besonders dann angenehm ist, wenn die aufgerufene Funktion eine ganze Reihe von Argumenten hat. Mühselig wäre es, erst einmal die exakte Reihenfolge der Argumente nachschlagen zu müssen.

Vorsicht aber, wenn Sie Positions- und Schlüsselwort-Argumente mischen, wie wir es im Beispiel oben getan haben: Dann nämlich müssen die Positionsargumente stets am Anfang stehen. Sie können also nicht das erste Argument als Schlüsselwort-Argument, das zweite als Positionsargument übergeben.

Übrigens haben wir bereits in der Definition unserer Funktion ein Schlüsselwort-Argument verwendet, nämlich beim Aufruf von **print()**: Hier haben wir mit Hilfe des Arguments **sep** angegeben, dass die einzelnen Zeichenketten nicht durch ein Leerzeichen voneinander getrennt ausgegeben werden sollen, was ansonsten ärgerliche Leerräume an unpassenden Stellen verursacht hätte, zum Beispiel vor dem Ausrufezeichen, das den Gruß abschließt.

### ■ Optionale Argumente

Das gerade angesprochene Argument **sep** der Funktion **print()** ist ein Beispiel für ein *optionales* Argument, das wir angeben *können*, aber *nicht müssen* (in dem Kelvin-zu-Celsius-Umrechnungsbeispiel zum Beispiel haben wir **print()** ohne das Argument **sep** aufgerufen). **sep** besitzt einen Standardwert, nämlich '' (also ein Leerzeichen), das immer dann verwendet wird, wenn die Funktion **print()** ohne explizite Angabe eines Wertes für **sep** aufgerufen wird. Das sieht man auch schön, wenn man in der Python-Konsole die Hilfe für **print()** aufruft. Dort heißt es nämlich:

```
print(value, ..., sep=' ', end='\n', file=sys.stdout,  
      flush=False)
```

Das optionale Argument **sep** ist also mit einem Standardwert „vorbelegt“, während das Argument **value** kein optionales Argument ist. Würden wir darauf verzichten, es beim Funktionsaufruf anzugeben, würden wir eine Fehlermeldung erhalten.

Wenn wir nun unsere **willkommen()**-Funktion so anpassen wollten, dass man keine Grußbotschaft angeben muss, müssen wir einzig den Standardwert des Arguments **gruss** im Funktionskopf ergänzen:

```
def willkommen(name, nachricht, gruss = 'Willkommen'):
    print(gruss, ', ', name, '! ', nachricht, sep = '')
```

## 23

Danach könnten wir unsere Funktion auch so aufrufen:

```
willkommen('Sophie', 'Schön, dass Du dabei bist.')
```

Beachten Sie dabei bitte, dass wir die Reihenfolge der Argumente **gruss** und **nachricht** im Funktionskopf vertauscht haben. Der Grund liegt darin, dass in Python Argumente mit Standardwerten als letztes in der Funktionsdefinition stehen müssen. Folgen mehrere optionale Argumente, müssen diese beim Funktionsaufruf als Schlüsselwort-Argumente, also unter Angabe ihres Namens übergeben werden, denn sonst weiß Python nicht, welchem der optionalen Argumente es welchen der übergebenen Werte zuordnen soll – schließlich kann ja jedes der optionalen Argumente angegeben werden, oder aber eben auch nicht.

### ■ Unbestimmte Anzahl von Argumenten

Bei manchen Funktionen weiß man a priori noch nicht, mit vielen Argumenten sie schließlich aufgerufen werden. Ein gutes Beispiel dafür ist die Funktion **print()**. Sie kann eine ganze Reihe von Variablen bzw. Werten übernehmen und gibt alle auf dem Bildschirm aus. Bei jedem Aufruf von **print()** kann die Zahl der Argumente eine andere sein. Allerdings nur scheinbar, wie wir gleich sehen werden.

Angenommen, wir wollten unsere **willkommen()**-Funktion so erweitern, dass sie nicht nur eine, sondern mehrere Personen begrüßt. Die Anzahl der Personen soll dabei variabel sein. Das erreichen wir durch ein Tupel-Argument, das mit einem Sternchen vor dem Bezeichner gekennzeichnet ist:

```
def willkommen(*name, nachricht, gruss = 'Willkommen'):
    print(gruss, ', ', end = '', sep = '')
    print(*name, end = '', sep=', ')
    print('!', nachricht)
```

Diese Funktion können wir nun mit mehreren Namen aufrufen, zum Beispiel so:

```
willkommen('Sophie', 'Mark', 'Celine',
           nachricht = 'Schön, dass Ihr dabei seid.')
```

Der Output lautet dann:

```
Willkommen, Sophie, Mark, Celine! Schön, dass Ihr dabei seid.
```

Zwei Dinge sind an unserer Funktion bemerkenswert:

1. Im Vergleich zur vorherigen Version von **willkommen()** arbeiten wir hier mit mehreren Print-Anweisungen. Das ist notwendig, um die richtige Positionierung von Kommata und Leerzeichen zu erreichen. Indem wir das (optionale) Argument **end** der Funktion **print()** auf eine leere Zeichenkette setzen (Standardwert ist '**\n**', also Zeilenumbruch) sorgen wir dafür, dass wir trotz mehrerer aufeinanderfolgender **print()**-Aufrufe stets auf derselben Zeile weiterschreiben.
2. Das Argument **name** sammelt jetzt die „losen“ Argumente, also die Namen der Personen, die wir beim Funktionsaufruf angeben, ein und packt sie in ein Tupel mit dem Bezeichner **name**. Mit diesem Tupel können wir dann im Funktionsrumpf arbeiten. In unserem Beispiel „entpacken“ wir das Tupel (indem wir wieder das Sternchen verwenden) und geben seine Elemente mit Hilfe von **print()** aus. Hätten wir im Aufruf von **print()** auf das Sternchen verzichtet, wäre einfach das Tupel „en bloc“ ausgegeben worden, was optisch nicht so ansprechend ist (probieren Sie es aus!).

Eine Alternative zu diesem Vorgehen wäre natürlich gewesen, dass wir beim Aufruf für die Namen der zu begrüßenden Personen tatsächlich nur ein Argument übergeben hätten, nämlich eine Liste, die wir dann in unserer Funktion entsprechend hätten verarbeiten müssen. Mit einer derart angepassten Funktion würde der Funktionsaufruf dann so aussehen:

```
willkommen(['Sophie', 'Mark', 'Celine'],
            nachricht = 'Schön, dass Ihr dabei seid.')
```

Beachten Sie, dass die drei Namen in diesem Fall nur *ein* Argument darstellen, nämlich unsere Namensliste, während wir beim vorherigen Aufruf der Funktion *drei verschiedene* Namensargumente übergeben haben, nämlich '**Sophie**', '**Mark**' und '**Celine**', die Python für uns nur praktischerweise einsammelt und in ein Tupel packt, mit dem wir dann weiterarbeiten können. Diese Art, die Funktion aufzurufen, ist etwas „natürlicher“ und intuitiver und deshalb der Listen-Lösung vorzuziehen.

Wenn eine Funktion ein solches unbestimmtes Tupel-Argument benutzt, müssen alle *folgenden* Argumente als Schlüsselwort-Argumente, also mit ihrem Bezeichner aufgerufen werden; das ist einleuchtend, denn wie soll Python sonst unterscheiden, ob ein übergebener Wert noch zum Tupel-Argument gehört oder bereits zum nächsten Argument.

Übrigens: Hin und wieder werden Sie auch Funktionen mit Argumenten sehen, die mit einem doppelten Sternchen (\*\*) eingeleitet werden; die **tkinter**-Funktion **config()**, die wir benutzen können, um die Optionen von **tkinter**-Widgets zu setzen (siehe ► Abschn. 22.2.3.1) ist eine solche Funktion (schauen Sie in der Hilfe nach!).

Solche Argumente sind auch Sammel-Argumente, allerdings für Schlüsselwort-Argumente. Auf diese Weise lassen sich unterschiedliche Argumente, die als Schlüsselwort-Argumente übergeben werden, aufsammeln und in ein Dictionary packen, dessen Schlüssel die Argumente-Namen und dessen Werte die für diese Argumente übergebenen Werte sind. Anders als die Sammelargumente für nicht-benamte Argumente (also die „\*-Argumente“), muss ein Sammel-Argument für Schlüsselwort-Argumente *immer am Ende* der Argumenten-Liste in der Funktionsdefinition stehen.

#### ■ Datentypen von Funktionsargumenten: Typhinweise (function annotations)

Ihnen ist wahrscheinlich bereits aufgefallen, dass die Argumente von Funktionen *nicht typisiert* sind, das bedeutet, die Datentypen der Argumente sind nicht vorgegeben. Dementsprechend kann Python auch nicht prüfen, ob eine Funktion auch mit Argumenten des jeweils richtigen Typs aufgerufen wird. Als wir unsere **willkommen()**-Funktion geschrieben haben, sind wir natürlich davon ausgegangen, dass die Argumente **name**, **gruss** und **nachricht** allesamt Zeichenketten, also Argumente vom Typ **str** sind. Das muss aber keineswegs so sein! Ein Benutzer unserer Funktion, der sich der genauen Bedeutung der einzelnen Argumente nicht bewusst ist, könnte als Argument **gruss** zum Beispiel einen bool'schen Wert angeben, in der Erwartung, es handele sich um eine Option, die festlegt, ob eine Grußbotschaft angezeigt wird, oder nicht.

Um diesem Problem entgegenzuwirken, kennt Python ein Konzept namens Typhinweise (engl. *type hints*, auch bekannt als *function annotations*). Sie werden in die Funktionsdefinition eingebaut und geben den erwarteten Typ des Arguments an. Das könnte für unsere Funktion **willkommen()** dann so aussehen:

```
def willkommen(name: str, nachricht: str,
               gruss: str = 'Willkommen'):
    print(gruss, ', ', name, '! ', nachricht, sep = '')
```

Wie Sie sehen, steht hinter jedem Argumentbezeichner ein Doppelpunkt, gefolgt von dem für dieses Argument erwarteten Typ.

Nun ist es aber keineswegs so, dass automatisch Python prüft, ob sich der Benutzer tatsächlich an die Typ-Vorgabe hält. Die Vorgabe hat nur indikativen Charakter. Trotzdem ist sie aus zwei Gründen nützlich: Zum einen nämlich gehen die Typhinweise auch in die automatisch erzeugte Hilfe für die Funktion ein und sind dort für jeden Benutzer der Funktion sichtbar. Zum anderen gibt es Entwicklungswerkzeuge, die diese Typhinweise auswerten. Eines dieser Werkzeuge ist *PyCharm*. Wenn Sie etwa versuchen, unsere Funktion **willkommen()** mit einem bool'schen Wert für das Argument **gruss** aufzurufen:

```
willkommen('Sophie', 'Schön, dass Ihr dabei seid.', True)
```

hebt *PyCharm* im Code-Editor-Fenster den bool'schen Wert hervor. Gehen Sie mit der Maus darüber, weist Sie ein kleines Popup-Fenster mit der Meldung **Expected type 'str', got 'bool' instead** auf den Fehler hin.

### 23.1.3 Rückgabewerte

Unsere Funktion **kelvin\_zu\_celsius()** aus dem letzten Abschnitt hat einen Kelvin-Wert als Argument übergeben bekommen, diesen in Grad Celsius umgerechnet und das Ergebnis auf dem Bildschirm ausgegeben. Natürlich könnten wir aber auch auf die Ausgabe verzichten und stattdessen den errechneten Celsius-Wert einfach zurückgeben. In diesem Fall handelt es sich um eine Funktion mit *Rückgabewert*.

Die Rückgabe wird mit der **return**-Anweisung bewerkstelligt.

```
def kelvin_zu_celsius(kelvin: float):  
    return kelvin - 273.15
```

Diese Funktion können wir nun aus unserem Hauptprogramm heraus aufrufen, ihr Ergebnis zum Beispiel zunächst in einer Variablen speichern und dann in der (Run-)Konsole ausgeben:

```
temp = kelvin_zu_celsius(290)  
print(temp)
```

Wir haben in der Funktionsdefinition das Argument **kelvin** mit einem Typhinweis versehen. Gleicher können wir auch mit dem Rückgabewert tun:

```
def kelvin_zu_celsius(kelvin: float) -> float:  
    return kelvin - 273.15
```

Dazu wird hinter den eigentlichen Kopf der Funktion der Typ des Rückgabewerts mit einem Pfeil **->** angeschlossen; der Doppelpunkt leitet wiederum den folgenden Code-Block, also den Funktionsrumpf, ein (Achtung: Bei den Funktionsargumenten hatten wir hierzu den Doppelpunkt verwendet).

Manchmal werden Sie mehr als nur einen Rückgabewert liefern wollen. Dann bietet es sich an, die unterschiedlichen Elemente der Rückgabe in ein Tupel zu verpacken. Im Folgenden wird unsere Funktion **kelvin\_zu\_celsius()** so verändert, dass sie sowohl den errechneten Celsius- als auch den ursprünglichen Kelvin-Wert zurückliefert, und zwar als zwei Elemente eines Tupels:

```
def kelvin_zu_celsius(kelvin: float) -> float:  
    return (kelvin, kelvin - 273.15)
```

Die Klammern um die beiden Werte können auch weggelassen werden und sind hier nur mitgeschrieben worden, um deutlich zu machen, dass hier ein Tupel erzeugt wird. Tatsächlich erzeugt die **return**-Anweisung automatisch ein Tupel, wenn ihr – durch Kommata separiert – mehrere Objekte folgen.

Auf die Elemente des Tupels kann nach dem Funktionsaufruf einfach zugegriffen werden, entweder durch Indizierung oder dadurch, dass das Tupel bereits bei der Zuweisung „entpackt“ wird:

## 23

```
temp = kelvin_zu_celsius(290)
print(temp[1])

cel, kel = kelvin_zu_celsius(290)
print(kel)
```

Mit der **return**-Anweisung wird die Funktion automatisch verlassen. Deshalb sollte **return** immer die letzte Anweisung in einer Funktion sein. Alle hinter ihr stehenden Programmbefehle würden ohnehin nicht mehr ausgeführt.

### 23.1.4 Lokale und globale Variablen

---

Betrachten Sie das folgende Programm:

```
begruebung = 'Moin'

def willkommen(name, nachricht, gruss):
    begruebung = gruss + ', ' + name + '! ' + nachricht
    print(begruebung)

willkommen('Sophie', 'Schön, dass Du mit dabei bist!', 'Hallo')
print(begruebung)
```

Überlegen Sie einmal, was dieser Code in der (Run-)Konsole ausgeben wird. Haben Sie eine Idee? Dann probieren Sie es in Python aus. War Ihre Erwartung richtig?

Startet man das Programm, wird folgendes ausgegeben:

```
Hallo, Sophie! Schön, dass Du mit dabei bist!
Moin
```

Die interessante Frage ist nun, warum die Variable **begruebung**, wenn wir sie mit **print()** ausgeben, nach wie vor den Wert 'Moin' besitzt, den wir ihr zu Beginn des Programms zugewiesen haben. Denn bevor wir ihren Wert ausgeben, rufen wir ja noch die Funktion **willkommen()** auf, die den Wert von **begruebung** ändert, in

unserem Beispiel auf "**Hallo, Sophie! Schön, dass Du mit dabei bist!**". Müsste **begruessung** am Ende des Programms dann nicht diesen Wert beinhalten?

Des Rätsels Lösung besteht darin, dass die Variable **begruessung** in unserem Hauptprogramm und die Variable **begruessung** in der Funktion **willkommen()** letztlich *zwei unterschiedliche* Variablen sind. Die Variable **begruessung**, die wir im Funktionsrumpf von **willkommen()** anlegen, existiert nur innerhalb dieses Code-Blocks, ihr *Gültigkeitsbereich* ist auf die Funktion **willkommen()** beschränkt. Wann immer wir aber innerhalb des Funktionsrumpfs von **willkommen()** auf die Variable **begruessung** zugreifen, arbeiten wir mit der in diesem Code-Block angelegten Variable, nicht mit der Variable, die wir im Hauptprogramm definiert haben. Man könnte also sagen, die Variable **begruessung** in unserer Funktion **willkommen()** „verdeckt“ die gleichnamige Variable des Hauptprogramms. Wir kommen aus der Funktion heraus nicht an die gleichnamige Variable des Hauptprogramms heran, die in der Funktion definierte Variable steht gewissermaßen „im Weg“.

Es gibt aber eine Möglichkeit, trotzdem auf die Variable des Hauptprogramms zuzugreifen. Betrachten Sie dazu den folgenden, leicht angepassten Code:

```
begruessung = 'Moin'

def willkommen(name, nachricht, gruss):
    global begruessung
    begruessung = gruss + ', ' + name + '! ' + nachricht
    print(begruessung)

willkommen('Sophie', 'Schön, dass Du mit dabei bist!', 'Hallo')
print(begruessung)
```

Wie Sie sehen, haben wir in der Funktion **willkommen()** lediglich die Anweisung **global begruessung** ergänzt. Die bewirkt, dass Python bei einem späteren Zugriff auf die Variable **begruessung** nicht etwa eine neue Variable erzeugt, die nur innerhalb der Funktion **willkommen()** gültig ist – man spricht von diesem Gültigkeitsbereich auch als dem *Namensraum* der Funktion. Stattdessen wird der globale Namensraum, also der Namensraum unseres Hauptprogramms, nach einer Variablen diesen Namens durchsucht; wird eine gefunden, wird mit ihr gearbeitet, anderenfalls wird im kleineren Namensraum der Funktion schließlich doch eine Extra-Variable angelegt, die freilich zu existieren aufhört, sobald die Funktion wieder verlassen wird (probieren Sie es aus und ändern Sie innerhalb der Funktion alle Vorkommen von **begruessung** auf einen anderen Bezeichner!).

Sie werden nun vielleicht argumentieren, dass die Situation in unserem Beispiel ja doch etwas künstlich und ihre Problematik letztlich nur darauf zurückzuführen ist, dass in der Definition der Funktion und im Hauptprogramm Variablen gleichen Namens verwendet werden. Stellen Sie sich aber vor, Sie wollten aus der Funktion heraus eine *globale*, das heißt im Namensraum des Hauptprogramms definierte Status-Variablen verändern, zum Beispiel eine Variable, die anzeigt, ob das aktuell bearbeitete Dokument bereits gespeichert ist oder nicht. Diese Status-

Variable soll natürlich unabhängig von der Funktion existieren und auch dann noch vorhanden sein, wenn Ihre Funktion längst wieder verlassen und alle *lokalen*, also in der Funktion selbst (genauer: in ihrem Namensraum) definierten Variablen, längst wieder gelöscht worden sind. Greifen Sie dann ohne eine **global**-Anweisung in Ihrer Funktion auf die Status-Variable zu, wird einfach im Namensraum der Funktion eine lokale Variable gleichen Namens erzeugt, die untergeht, wenn Ihre Funktion vollständig durchlaufen worden ist. Die globale Status-Variable bleibt davon unberührt. Erst durch die **global**-Anweisung machen Sie Python klar, dass Sie mit der globalen Variablen arbeiten und keineswegs eine neue lokale Variable innerhalb Ihrer Funktion erzeugen wollen. Ihre Funktion verändert dann ihr Umfeld (in Gestalt der Status-Variable). Solche sogenannten Nebenwirkungen (engl. *side effects*) versucht man im Allgemeinen zu vermeiden, um die Funktion unabhängiger vom den Programmcode zu machen, der sie aufruft.

### ? 23.1 [20 min]

Schreiben Sie eine Funktion **erzeuge\_website()**, die als Argumente einen Titel, eine Überschrift und einen Text (jeweils als Strings) übernimmt, daraus ein HTML-Dokument (also letztlich eine einfache Webseite) erzeugt und diese in einer Datei namens **website.html** speichert. Dokumentieren Sie Ihre Funktion soweit wie möglich.

Ein HTML-Dokument hat folgenden grundsätzlich folgenden Aufbau (die Einrückungen sind lediglich zu besseren Verdeutlichung der Struktur, sie müssen nicht in die Datei geschrieben werden):

```
<html>
  <head>
    <title>Hier steht der Titel der Webseite</title>
  <head>
  <body>
    <h1>Hier steht eine Überschrift</h1>
    <p>Hier steht ein Text</p>
  </body>
</html>
```

Letztlich ist das HTML-Dokument eine Ansammlung sogenannter Elemente wie etwa **h1** (für Header 1, also Überschriftenlevel 1) oder **p** für Paragraph, also Absatz. Anfang und Ende der Elemente werden durch sogenannte Tags („Schildchen“) markiert, wobei die End-Tags den Anfangtags entsprechen, ergänzt um einen führenden Querstrich. Zwischen Anfangs- und End-Tags steht der Inhalt, wobei HTML-Elemente auch andere HTML-Elemente beinhalten können (so enthält das

## 23.1 · Arbeiten mit Funktionen

umschließende **html**-Element zum Beispiel die Elemente **head** und **body**, die ihrerseits wiederum weitere Elemente beinhalten können).

Die Funktion **erzeuge\_website()** soll ein solches HTML-Dokument erzeugen, die als Argumente übergebenen Informationen „einbauen“ und das Ganze als **website.html** speichern. Danach können Sie diese Datei mit Ihrem Webbrowser betrachten!

### 23.2 [10 min]

Die folgenden beiden Funktionen weisen Fehler auf. Schreiben Sie die Funktionen so um, dass sie syntaktisch korrekt sind und ihren Zweck erfüllen:

a)

```
from random import *
def wuerfel() -> int:
    ''' Erzeugt eine Zufallszahl analog einem Würfel-Wurf '''
    wuerfel_ergebnis = randint(1,6)
```

b)

```
def erzeuge_telefonnummer(land: str, vorwahl: str,
                           nummer: str) -> str:
    '''Generiert eine sauber formatierte Telefonnummer auf
    Basis ihrer Bestandteile; vorwahl ist die Vorwahl mit
    führender Null, land der ISO-Code des Landes'''
    laender_dict = {'DE': '49', 'FR': '33', 'CH': '41',
                    'AT': '43', 'NL': '31', 'BE': '32',
                    'PL': '48', 'DK': '45', 'CZ': '42'}
    return '+' + laender_dict[land] + vorwahl[1:] + nummer
```

### 23.3 [5 min]

Entwickeln Sie eine Funktion **selbstbeschaeftigung()**, die sich nur mit sich selbst beschäftigt, indem sie eine a priori unbestimmte Zahl *benannter* (also *Schlüsselwort*-) Argumente entgegennimmt und zunächst deren Namen und dann deren Werte auf dem Bildschirm ausgibt.

### 23.4 [5 min]

Welche Ausgabe bewirkt das folgende Programm, und warum?

```

from datetime import datetime

umsatz_gesamt = 0.00
letzter_verkauf = 0.00

def buche_verkauf(umsatz_verkauf, artikelnummer):
    global umsatz_gesamt
    letzter_verkauf = umsatz_verkauf
    umsatz_gesamt = umsatz_gesamt + umsatz_verkauf
    zeit = datetime.now()
    print(zeit.strftime(
        '%d.%m.%Y %H:%M:%S') + ' -- Neuer Umsatz: ' + str(
        round(umsatz_verkauf,
              2)) + ' Euro mit Artikelnummer ' + artikelnummer + '.')

buche_verkauf(10.99, 'DE07011981')
buche_verkauf(24.99, 'DE25101878')

print('Gesamtumsatz: ' + str(umsatz_gesamt))
print('Umsatz mit letztem Verkauf: ' + str(letzter_verkauf))

```

## 23.2 Funktionen als Klassen-Methoden von Objekten verwenden

An vielen Stellen haben wir bereits wie selbstverständlich mit Funktionen gearbeitet, die Bestandteil von Klassen sind. Solche Funktionen werden bekanntermaßen auch als *Methoden* bezeichnet (wenn Ihnen das nicht mehr geläufig ist, blättern Sie am besten noch einmal einige Seiten zurück zu ► Abschn. 11.7.4).

Als wir beispielsweise die Anordnung der Steuerelemente unserer **tkinter**-Benutzeroberfläche mit **grid()** festlegten, hatten wir durch Aufrufe wie **kopieren.grid(row = 1, column = 4, sticky = 'news')** (aus dem Taschenrechner-Beispiel in ► Abschn. 22.2.6), eine Methode aufgerufen, hier nämlich die Methode des Objekts **kopieren**, die eine Instanz der **tkinter**-Klasse **Button** ist. Damit Python weiß, die Methoden welchen Objekts genau wir aufrufen wollten, haben wir den Funktionsaufruf mit dem Punkt-Operator an den Bezeichner des betreffenden Objekts geschlossen. Allgemein haben Methodenaufrufe also die Form **objekt.methode(...)**.

Da Methoden nichts weiter sind als ganz normale Funktionen, die dazu dienen, „ihr“ Objekt zu bearbeiten, bräuchten wir sie eigentlich an dieser Stelle gar nicht gesondert behandeln. Wir wollen aber doch noch kurz einen Blick auf drei besondere Themen im Zusammenhang mit Methoden werfen, nämlich,

- wie Methoden als Teil von Klassen definiert werden,
- welche besondere Rolle Konstruktor-Methoden dabei einnehmen, und
- welche nützlichen Standardmethoden Klassen in Python haben, die wir für unsere Zwecke anpassen können.

### ■ Methoden als Teil von Klassen definieren

In ▶ Abschn. 21.7 hatten wir eine Klasse **Produkt** folgendermaßen definiert.

```
class Produkt:
    bezeichnung = ''
    beschreibung = ''
    artikelnummer = ''
    hersteller = ''
    preis = 0.0
```

Diese Klasse besteht ausschließlich aus Eigenschaften/Attributen. Angenommen, wir wollten nun unserer Klasse eine spezielle **anzeigen()**-Methode mitgeben, die die Eigenschaften eines Produkts übersichtlich darstellt. Dazu müssten wir den Code-Block der Klasse einfach um die Definition der Methode erweitern:

```
class Produkt:
    bezeichnung = ''
    beschreibung = ''
    artikelnummer = ''
    hersteller = ''
    preis = 0.0

    def anzeigen(self):
        print('Produkt:', self.bezeichnung,
              '\nBeschreibung:', self.beschreibung,
              '\nArtikelnummer:', self.artikelnummer,
              '\nHersteller:', self.hersteller,
              '\nPreis:', self.preis, '\n')
```

Wie Sie sehen, ist der Kopf der Funktionsdefinition genauso weit eingerückt, wie die Attribute, er ist daher Bestandteil der Klassendefinition.

Die Funktion **anzeigen()** ist damit Bestandteil der Klassendefinition von **Produkt**. Für alle Objekte vom Typ **Produkt** kann die Methode von nun an aufgerufen werden. So könnten wir beispielsweise folgendes Produkt definieren:

```
p = Produkt()
p.preis = 10.99
p.bezeichnung = 'Gartenschaufel'
```

Zur Anzeige der Produkteigenschaften können wir dann ganz bequem unsere selbstdefinierte Methode aufrufen:

```
p.anzeigen()
```

Das resultiert in einem sauber strukturierten Output:

```
Produkt: Gartenschaufel
Beschreibung:
Artikelnummer:
Hersteller:
Preis: 10.99
```

## 23

Die Eigenschaften, denen wir nicht ausdrücklich Werte zugewiesen haben (wie etwa **beschreibung**) werden dabei natürlich mit ihren Standardwerten (also „leeren“ String) dargestellt.

Ihnen ist sicher das Argument **self** in der Definition unserer Methode **anzeigen()** aufgefallen. **self** repräsentiert immer dasjenige Objekt, für das die Methode aufgerufen wird. Auf diese Weise können wir bequem auf die Eigenschaften (und ggf. auch auf andere Methoden) des aktuellen Objektinstanz, für die unsere Methode aufgerufen wird, zugreifen. Dabei muss dieses Argument nicht notwendigerweise **self** heißen (es muss lediglich an erster Stelle in der Argumente-Liste stehen), es ist aber gute Praxis, den leicht verständlichen Bezeichner **self** zu verwenden.

Methoden werden – ebenso wie Eigenschaften – auch vererbt. In ► Abschn. 21.7.2 hatten wir von der Klasse **Produkt** eine Klasse **Buch** abgeleitet, also ein spezielles Produkt. Diese Klasse erbte alle Eigenschaften der Elternklasse, und besaß darüber hinaus noch weitere Eigenschaften, die eben nur für Bücher relevant sind, wie etwa die Seitenzahl. Ähnlich funktioniert auch die Vererbung von Methoden.

Dabei besteht auch die Möglichkeit, Methoden zu *überladen*. Das bedeutet, die speziellere Klasse (in unserem Fall also **Buch**) besitzt eine eigene Methode **anzeigen()**, die vielleicht die besonderen Eigenschaften von Büchern mit darstellt. Damit verfügen nun sowohl die Elternklasse als auch die von ihr abgeleitete Klasse jeweils über eine Methode **anzeigen()**.

Wenn wir dann die Methode **anzeigen()** für ein Objekt vom Typ **Buch** aufrufen, schaut Python zunächst, ob *diese Klasse selbst* über eine entsprechende Methode verfügt; falls ja, wird diese ausgeführt. Falls aber **Buch** selbst keine Methode **anzeigen()** besitzen sollte, wird geprüft, ob die nächst höhere Klasse in der Klassenhierarchie, also die Elternklasse **Produkt**, über eine solche Methode verfolgt. Auf diese Weise ist es möglich, Klassen auf unterschiedlichen Ebenen der Klassenhierarchie mit gleichnamigen Methoden auszustatten, die aber ein auf die jeweilige Klasse speziell abgestimmtes Verhalten zeigen. Der Benutzer hingegen kann (zumindest dem Namen nach) einfach immer die gleiche Methode aufrufen und muss sich nicht mit den Spezifika der verschiedenen Klassen befassen. Eine große Stärke der objektorientierten Programmierung!

### ■ Die besondere Rolle der Konstruktor-Methode

Im Beispiel oben hatten wir eine Instanz der Klasse **Produkt** mit Hilfe von deren Standardkonstruktor **Produkt()** erzeugt. Dieser gibt einfach ein neues Objekt dieser Klasse zurück. Wir können aber alternativ auch einen *eigenen* Konstruktor definieren, der vielleicht einige Argumente übernimmt, also beispielsweise die Be-

zeichnung des Produkts und seinen Preis als die beiden wichtigsten Eigenschaften. Diese als Argumente übergebenen Eigenschaften würde der Konstruktor dann den entsprechenden Attributen zuweisen und das solcherart definierte Objekt zurückliefern.

Einen eigenen Konstruktor erzeugen wir dadurch, dass wir die Standardmethode `__init__()` (jeweils zwei Unterstriche!) anpassen. Wir würden also unserer Klassendefinition folgende Methodendefinition hinzufügen:

```
def __init__(self, bezeichnung, preis):
    self.bezeichnung = bezeichnung
    self.preis = preis
```

Auch die Konstruktorfunktion führt als erstes Argument wieder `self`, hier also das Objekt, das durch sie erzeugt wird. Die weiteren Argumente haben wir frei definiert. Damit könnten wir nun das bereits zuvor verwendete Produkt mit der Bezeichnung "Gartenschaufel" zum Preis von **10.99** auch wie folgt erzeugen, wobei auch hier wiederum gilt, dass `self` beim Aufruf nicht angegeben werden muss, denn darum kümmert sich Python selbst:

```
p = Produkt('Gartenschaufel', 10.99)
```

Beachten Sie bitte, dass wir keineswegs die Konstruktor-Funktion `__init__()` unter deren Namen aufrufen, sondern den Konstruktor der Klasse, dessen Bezeichner mit dem der Klasse identisch ist. Im Hintergrund ruft Python dann allerdings die `__init__()`-Methode auf, entweder die Standardversion, oder, wenn wir diese überschrieben haben, unsere eigene Variante. Normalerweise wird man `__init__()` nicht selbst aufrufen, es sei denn, man möchte aus dem Konstruktor einer abgeleiteten Klasse heraus den Konstruktor der Elternklasse aufrufen.

### ■ Zwei nützliche Standardmethoden von Klassen

Wenn Sie ein Objekt unserer Klasse **Produkt** in der Python-Konsole erzeugen (dazu müssen Sie zunächst die Definition der Klasse in der Konsole ausführen!) und dann den Namen des Objekts, zum Beispiel `p`, eingeben, erhalten Sie eine recht unschöne Anzeige:

```
> p
<__main__.Produkt at 0x1ff91106048>
```

Das lässt sich zum Glück ändern, und zwar, indem wir die Standardmethode `__repr__()` überschreiben. Die wird immer dann aufgerufen, wenn der Benutzer den Bezeichner eines Objekts in die Konsole eingibt und gibt den String zurück, der dann in der Konsole angezeigt werden soll.

Wir könnten nun also die Methode überladen, indem wir der Definition unserer Klasse **Produkt** zum Beispiel folgende Methodendefinition hinzufügen:

```
def __repr__(self):
    return 'Produkt: ' + self.bezeichnung + '\nPreis: ' + str(
        self.preis)
```

**23**

Geben wir nun den Bezeichner unseres Objekts in die Konsole ein, erhalten wir einen aussagekräftigeren Output (Achtung: Nach Anpassung der Klassendefinition müssen Sie ein *neues* Objekt dieser Klasse **Produkt** erzeugen, damit es die neue Methode `__repr__()` besitzt):

```
> p
Produkt: Gartenschaufel
Preis: 10.99
```

Auf ähnliche Weise können wir bestimmen, was passieren soll, wenn der Benutzer die Funktion `print()` aufruft und versucht, unser Objekt anzuzeigen. `print()` ruft im Hintergrund automatisch die Methode `__str__()` auf und gibt deren Rückgabewert aus. Somit könnten wir in der Definition unserer **Klasse Produkt** die Methode `__str__()` folgendermaßen überladen:

```
def __str__(self):
    return 'Produkt "' + self.bezeichnung + '" (' + \
        str(self.preis) + ' EUR)'
```

Anschließend können wir dann die Methode `print()` mit unserem Objekt **p** als Argument aufrufen und erhalten eine hübschere Darstellung:

```
> print(p)
Produkt "Gartenschaufel" (10.99 EUR)
```

## 23.3 Arbeiten mit Modulen und Packages

### 23.3.1 Programmcode modularisieren

Python erlaubt es, Code zur besseren Wiederverwendung in andere Dateien auszulagern. So könnten Sie zum Beispiel Funktionen oder ganze Klassen, die Sie entwickelt haben und in verschiedenen Programmen benutzen wollen, in einer Python-Datei zusammenfassen und dann von anderen Programmen aus darauf zugreifen. Man spricht bei solchen Dateien, die ausgelagerten Programmcode aufnehmen, von *Modulen*. Ein Modul ist also nicht anderes als Programmcode, der zur Wiederverwendung in einer eigenen Python-Datei zusammengefasst worden ist.

Name	Änderungsdatum	Typ	Größe
__pycache__	06.08.2019 12:26	Dateiordner	
test	20.12.2018 22:33	Dateiordner	
__init__.py	27.06.2018 06:07	JetBrains PyChar...	163 KB
__main__.py	27.06.2018 06:07	JetBrains PyChar...	1 KB
colorchooser.py	27.06.2018 06:07	JetBrains PyChar...	2 KB
commondialog.py	27.06.2018 06:07	JetBrains PyChar...	2 KB
constants.py	27.06.2018 06:07	JetBrains PyChar...	2 KB
dialog.py	27.06.2018 06:07	JetBrains PyChar...	2 KB
dnd.py	27.06.2018 06:07	JetBrains PyChar...	12 KB
filedialog.py	27.06.2018 06:07	JetBrains PyChar...	15 KB
font.py	27.06.2018 06:07	JetBrains PyChar...	7 KB
messagebox.py	27.06.2018 06:07	JetBrains PyChar...	4 KB
scrolledtext.py	27.06.2018 06:07	JetBrains PyChar...	2 KB
simplerdialog.py	27.06.2018 06:07	JetBrains PyChar...	12 KB
tix.py	27.06.2018 06:07	JetBrains PyChar...	76 KB
ttk.py	27.06.2018 06:07	JetBrains PyChar...	57 KB

Abb. 23.1 Verzeichnisstruktur des Python-Moduls `tkinter`

Mehrere inhaltlich zusammenhängende Module wiederum lassen sich zu einem *Package* zusammenfassen. Während ein Modul technisch gesehen nichts anderes als eine Python-(.py)-Datei ist, stellt ein Package ein *Verzeichnis* dar, in dem *mehrere* Module, also mehrere .py-Dateien liegen. Damit Python weiß, dass dieses Verzeichnis ein Package sein soll, muss in dem Verzeichnis auch eine Datei namens `__init__.py` (doppelte Unterstriche!) liegen. Diese Datei darf durchaus leer sein, sie zeigt Python lediglich an, dass dieses Verzeichnis als Package zu betrachten ist. Natürlich darf `__init__.py` aber auch selbst Code enthalten. Ein gutes Beispiel dafür ist das Package `tkinter`, von dem wir schon ausgiebig Gebrauch gemacht haben.

Abb. 23.1 zeigt die Verzeichnisstruktur dieses Packages.

Hier sehen Sie auch, dass die Datei `__init__.py` sehr umfangreich ist, also keineswegs einfach nur eine leere Hülle darstellt, sondern durchaus eine Menge Code beinhaltet.

### 23.3.2 Elemente aus Modulen importieren

Nachdem wir gesehen haben, dass Module und Packages es erlauben, Programmcode aus einem Programm auszulagern, stellt sich natürlich die Frage, wie genau wir dann auf den ausgelagerten Programmcode, also etwa auf Klassen und Funktionen, zugreifen können. Schließlich befindet sich deren Programmcode ja nicht mehr in unserer Hauptprogramm-Datei, muss also irgendwie „verfügbar“ gemacht werden. Dieses Verfügbar machen wird als *Importieren* bezeichnet und ist etwas, das wir, ohne es genauer zu besprechen, bereits etliche Male gemacht haben.

### ■ Ausgewählte Klassen importieren

Wenn Sie zum Beispiel zu unserem Taschenrechner-Programm in ▶ Abschn. 22.2.6 zurückblättern, finden Sie dort zu Beginn des Programms folgende zwei Anweisungen:

```
from tkinter import Tk, Button, Label
from tkinter.font import Font
```

23

Sie können leicht erkennen, dass diese stets dem Aufbau **from modulname\_oder\_pakagename import klassenliste** folgen. Mit der ersten Anweisung zum Beispiel werden aus dem **tkinter**-Modul die Klassen **Tk**, **Button** und **Label** importiert. Mit der zweiten Anweisung wird aus dem Modul **font**, das zum Package **tkinter** gehört (deshalb **tkinter.font**), die Klasse **Font** importiert.

Wenn Sie nochmal einen Blick auf die Verzeichnisstruktur des **tkinter**-Packages (▣ Abb. 23.1) werfen, so sehen Sie, dass es tatsächlich eine Datei **font.py** gibt, das Modul **font**, aus dem wir mit der zweiten Anweisung die Klasse **Font** importieren. Suchen Sie einmal auf Ihrer Festplatte in der Python-Installation den Pfad **\Lib\ tkinter**, und öffnen Sie die Datei **font.py**. Darin werden Sie unter anderem eine Definition der Klasse **Font** finden, die wir importieren.

Was aber ist dann mit der ersten Import-Anweisung? Hier importieren wir zwar drei Klassen, aber scheinbar direkt aus dem Package **tkinter**. In welcher Datei aber sind nun diese drei Klassen enthalten? Sie haben es sich wahrscheinlich schon gedacht: Diese Klassen stecken in der **\_\_init\_\_.py**-Datei, wovon Sie sich auch ganz leicht überzeugen können, indem Sie diese Datei öffnen.

Nachdem wir die Klassen mit einer **import**-Anweisung importiert haben, können wir sie in unserem Programm verwenden, und zwar ohne weiteres einfach dadurch, dass wir ihren Bezeichner verwenden, wie etwa in Zeile 35 mit **win = Tk()**.

### ■ Den gesamten Inhalt eines Moduls importieren

Statt beim Import ausdrücklich die Liste der Klassen anzugeben, die wir importieren wollen, hätten wir auch eine *Wildcard* benutzen und so einfach alle Klassen importieren können:

```
from tkinter import *
```

Dieses Vorgehen ist allerdings unter Python-Programmierern verpönt, weil man so nicht genau weiß, was man eigentlich alles importiert und auf diese Weise gegebenenfalls Namenskonflikte mit anderen Klassen entstehen, die man bereits in seinem Programmcode verwendet hat. Der kontrollierte Ansatz mit expliziter Angabe der zu importierenden Klassen wird daher üblicherweise vorgezogen.

### ■ Das gesamte Modul importieren

Ein anderer Weg des Imports besteht darin, einfach das gesamte Modul zu importieren:

```
import tkinter as tk
```

Dabei kann auf den letzten Teil der Anweisung, **as tk**, auch verzichtet werden. Allerdings macht die Verwendung von **as** den Zugriff auf das Modul leichter, vor allem, wenn der Modulname lang ist. Denn bei dieser Art von Import muss der Modulname beim Zugriff auf die Klassen des Moduls *immer mit angegeben werden*, also zum Beispiel:

```
loeschen = tkinter.Button(win, text = 'Löschen',
                           command = loeschen_press)
```

Bei Verwendung der Umbenennung mit Hilfe von **as** verkürzt sich das zu:

```
loeschen = tk.Button(win, text = 'Löschen',
                           command = loeschen_press)
```

**as** kann übrigens auch verwendet werden, um einzelne Klassen „umzubenennen“, die mit **from modul import klasse** importiert werden; das ist nützlich, vor allem um Namenskonflikte mit bereits vorhandenen (ggf. von Ihnen selbst entwickelten!) Klassen zu vermeiden.

### 23.3.3 Die Community nutzen – Der Python Package Index (PyPI)

Python kommt von Haus aus mit einer Reihe von Modulen und Packages, darunter auch das bereits verwendete Package **tkinter**.

Über die standardmäßig installierten Packages hinaus bietet der *Python Package Index (PyPI)* unter ► <https://pypi.org/> eine Vielzahl von Packages für beinahe jede erdenkliche Aufgabe. Jedes Package verfügt über eine eigene Seite mit einigen wichtigen Informationen über das Package, zum Beispiel dessen Autor, die Lizenz, unter der es zur Verfügung gestellt wird oder die zur Verwendung des Packages benötigt Python-Version. Regelmäßig wird auch eine kurze Beschreibung des Packages angeboten, die wichtig ist, um zu entscheiden, ob ein Package, das dem Namen nach gut klingt, tatsächlich den gewünschten Zweck erfüllt. Die Beschreibungen auf der PyPI-Seite des Packages sind aber oft recht dürftig, und so ist es gut, dass manche Packages eine eigene Homepage besitzen, die von der PyPI-Seite aus verlinkt ist und weiterführende Informationen zu dem Package bietet.

Ein Beispiel für eine solche PyPI-Seite ist ► <https://pypi.org/project/numpy/>, die Seite des bekannten Packages **NumPy**, das Python um Datentypen für die effiziente Arbeit mit mehrdimensionalen Arrays ergänzt.

Der *Python Package Index* bietet eine schier unfassbare Menge an Packages. Da die Packages aber durchaus von sehr unterschiedlichem Umfang und unterschiedlicher Qualität sind, empfiehlt es sich, im *Python Package Index* nicht einfach

„drauflos zu suchen“, sondern zunächst im Internet in den einschlägigen Diskussionsforen, auf Blogs und anderen Websites interessante „Package-Kandidaten“ zu ermitteln und diese dann auszuprobieren (gegebenenfalls zunächst auch einfach mehrere Packages für denselben Zweck installieren).

### ■ Packages installieren mit PyCharm

Wenn Sie ein Package gefunden haben, das Sie benutzen möchten, müssen Sie das Package installieren, bevor Sie es verwenden können.

23

In *PyCharm* können Sie dies über die grafische Benutzeroberfläche bewerkstelligen, indem Sie im Menü *File* auf den Menüpunkt *Settings* gehen und dann auf *Project | Project Interpreter*. Hier werden Ihnen nun die Packages gelistet, die aktuell zur Verfügung stehen. Durch Klick auf das kleine Plus-Button gelangen Sie in einen Dialog, in dem Sie ein Package aus dem *Python Package Index* auswählen und installieren können.

### ■ Packages installieren mit pip

Wenn Sie lieber über die Konsole des Betriebssystems als über die grafische Benutzeroberfläche von *PyCharm* arbeiten wollen, können Sie **pip** verwenden, ein Programm, dass die Administration von Packages erlaubt. **pip** ist übrigens eine rekursive Abkürzung, enthält sich also in der Langform *pip installs packages* wiederum selbst.

Wenn Sie mit **pip** arbeiten wollen, müssen Sie zunächst sicherstellen, dass es auch tatsächlich installiert ist. Wechseln Sie dazu in das **Scripts**-Verzeichnis Ihrer Python-Installation. Wenn Sie mit Microsoft Windows arbeiten und Python in **C:\python37** installiert haben, hat **pip** den Pfad **C:\python37\Scripts\pip.exe**. Sollte **pip** dort nicht vorhanden sein, wechseln Sie in das Verzeichnis, in dem **python.exe** (in unserem Beispiel typischerweise **C:\python37**) liegt und führen Sie folgenden Anweisung aus:

```
python -m ensurepip --default-pip
```

**pip** prüft selbst, ob es auf dem aktuellen Stand ist. Sollte das mal nicht der Fall sein, können Sie sehr einfach ein Update auf die aktuelle Version durchführen:

```
python -m pip install --upgrade pip
```

Danach können Sie mit **pip** ganz einfach Packages installieren, indem Sie die Anweisung **pip install packagename** ausführen, also zum Beispiel:

```
pip install NumPy
```

**pip** kann aber noch viel mehr. So können Sie sich Informationen zu einem Package mit **pip show** anzeigen lassen, zum Beispiel für **NumPy**:

```
pip show NumPy
```

Mit **pip deinstall packagename** können Sie ein Package auch wieder deinstallieren, mit **pip search suchbegriff** können Sie den *Python Package Index* direkt von der Kommandozeile durchsuchen, zum Beispiel:

```
pip search webscraping
```

Hilfe zu **pip** und seinen zahlreichen Optionen erhalten Sie mit der Anweisung

```
pip -h
```

oder mit

```
pip --help
```

### ■ Virtuelle Umgebungen

Python kann bei Bedarf das Package in eine sogenannte virtuelle Umgebung (engl. *virtual environment*) installieren. Dann wird das Package nicht der allgemeinen Package-Bibliothek hinzugefügt, sondern in einer separaten Bibliothek für Ihr aktuelles Projekt installiert. Auf diese Weise können Sie in unterschiedlichen Projekten mit unterschiedlichen Versionen ein- und desselben Packages arbeiten. Das kann wichtig sein, wenn Ihr Projekt zum Beispiel eine ältere Version eines Packages benötigt, weil es mit der aktuellen Version des Packages nicht lauffähig ist. Mit einem *virtual environment* halten Sie Ihr Projekt lauffähig, während Sie anderenorts mit der aktuellen Version des Packages arbeiten können. Möglich wird dies dadurch, dass Sie bei der Installation von Packages auch die zu installierende Version angeben können und auf diese Weise nicht zwingend die aktuellste Version benutzen müssen.

Ähnliches gilt übrigens für Python selbst. Sie können festlegen, mit welchem Python-Interpreter Sie arbeiten wollen. Mit den Python-Versionen 3.x wurden gegenüber den älteren Versionen 2.x einige wesentliche Änderungen an der Sprachdefinition vorgenommen und Python-Projekte, die unter Python 2.x entwickelt wurden, sind unter Version 3.x nicht notwendigerweise uneingeschränkt lauffähig. In *PyCharm* können Sie einfach festlegen, welcher der *Project Interpreter*, also der Interpreter, den Sie im aktuellen Projekt verwenden wollen, sein soll. Auf diese Weise ist ein älteres Projekt, das unter Python 2.x entwickelt wurde, weiterhin lauffähig, ohne es aufwendig umbauen zu müssen. Beim Erzeugen einer virtuellen Umgebung wird der Interpreter, mit dem Sie arbeiten wollen, mit in die virtuelle Umgebung kopiert.

Sie können aber natürlich auch einen Projektinterpreter auswählen, ohne eine virtuelle Umgebung zu erzeugen. Wenn Sie also einfach mit dem Python 2.x-

Interpreter und den für diesen Interpreter (in seiner „Hauptinstallation“) installierten Packages arbeiten wollen, können Sie in *PyCharm* den Projektinterpreter einfach auf diesen Interpreter umstellen.

Empfehlenswertes ist es auf jeden Fall, gleich beim Anlegen Ihres Projekts auf das Erzeugen einer virtuellen Umgebung zu verzichten, wenn Sie nicht zwingend eine benötigen (was in aller Regel nicht der Fall sein dürfte).

## 23

### 23.4 Zusammenfassung

---

In diesem Kapitel haben wir gesehen, wie Funktionen in Python definiert und benutzt werden; außerdem haben wir uns mit der Funktionsweise von Modulen und Packages beschäftigt und den *Python Package Index (PyPI)* als wichtige Bezugsquelle nützlichen Programmcodes kennengelernt.

Folgende Punkte sollten Sie aus diesem Kapitel unbedingt mitnehmen:

- Funktionen werden in Python mit der **def**-Anweisung definiert und bestehen aus dem Funktionskopf mit Bezeichner und Argumenten der Funktion sowie dem Funktionsrumpf, dem (eingerückten) Code-Block, der beim Aufruf der Funktion ausgeführt wird.
- Optionale Argumente erhalten im Funktionskopf der Funktionsdefinition ihren Standardwert zugewiesen (**argument = standard\_wert**).
- Die Funktionsargumente stehen in der Funktionsdefinition ohne Datentyp, können aber mit einem Type Hint der Form **: datentyp** versehen werden, der zwar nicht bindend ist, aber von vielen IDEs verarbeitet und zudem in der Hilfe zur Funktion angezeigt wird; neben Funktionsargumenten können auch die Rückgabewerte der Funktion mit Type Hints der Form **-> datentyp** versehen werden.
- Rückgabewerte werden mit dem Schlüsselwort **return** zurückgegeben.
- Variablen, die innerhalb von Funktionen definiert werden sind ebenso wie Funktionsargumente lokale Variablen und daher nur innerhalb der Funktion verwendbar; möchte man von innerhalb einer Funktion auf eine globale Variable zugreifen, muss das Schlüsselwort **global** verwendet werden.
- Auch, wenn eine Funktion keine Argumente besitzt müssen bei ihrem Aufruf (ebenso wie bei ihrer Definition) die runden Argumente-Klammern angegeben werden.
- Beim Aufruf einer Funktion können die Argumente auch als Schlüsselwort-Argumente, das heißt, mit ihrem Namen übergeben werden (in der Form **argument = wert**); die Reihenfolge der Argumente spielt dann keine Rolle.
- Python-Code lässt sich in Modulen und mehrere Module in Packages zusammenfassen.
- Klassen aus Modulen, die im Programm verwendet werden, müssen zunächst importiert werden, entweder durch explizite Angabe der zu importierenden Klassen in der Form **from modulname\_oder\_packagename import klassenliste** (empfohlene Vorgehensweise) oder durch Import *aller* Klassen in der Form **from modulname\_oder\_packagename import \***; auch kann das Modul als

## 23.5 · Lösungen zu den Aufgaben

Ganzes importiert werden mit einer Anweisung der Form **import modulname\_oder\_packagename**.

- Die wichtigste Quelle für Python-Module ist der *Python Package Index (PyPI)*; hier finden sich Lösungen für viele unterschiedliche Programmieraufgaben; die Recherche nach einem geeigneten Package lohnt allemal, bevor man selbst in die Programmierung der gesuchten Funktionalität einsteigt.
- Die Installation von Modulen des *PyPI* erfolgt entweder mit Hilfe des Kommandozeilenprogramms **pip** oder über die verwendete IDE.

## 23.5 Lösungen zu den Aufgaben

### ■ Aufgabe 23.1

Die Funktion **erzeuge\_website()** könnte so aussehen:

```
def erzeuge_website(title: str, header: str, text: str):  
    ''' Erzeugt eine einfache Website und speichert sie als  
    website.html.  
  
    Argumente:  
    -- title: Der Titel der Website  
    -- header: Überschrift des Dokuments  
    -- text: Der eigentliche Inhalt'''  
  
    html_content = '<html><head><title>' + title + \  
                  '</title></head><body><h1>' + \  
                  header + '</h1><p>' + text + \  
                  '</p></body></html>'  
    html_file = open('C:\website.html', 'w')  
    html_file.write(html_content)  
    html_file.close()
```

Ein möglicher Aufruf wäre dann:

```
erzeuge_website('Meine erste Python-Website',  
                 'Erster Abschnitt',  
                 'Hier könnte ein längerer Text stehen')
```

Die Funktion wird mit drei String-Argumenten für Titel, Überschrift und Textinhalt der zu erzeugenden Website aufgerufen. Einen Rückgabewert besitzt sie nicht. Stattdessen erzeugt sie den HTML-Code der Website als String-Variablen **html\_content** und schreibt diesen dann in die Datei **website.html**. Statt hier zunächst einen großen String mit dem Dateiinhalt zu erzeugen und diesen dann in die Datei zu schreiben, wäre es natürlich ebenso möglich gewesen, mit mehreren **write()**-Anweisungen den Dateiinhalt Schritt für Schritt in die Datei zu schreiben, ohne ihn zu Beginn bereits vollständig zusammenzubauen.

Wenn Sie die fertige Funktion nun aufrufen, können Sie die erzeugte Website danach in Ihrem Webbrower öffnen.

Neben der eigentlichen Funktion war aber in der Aufgabenstellung auch Dokumentation gefordert. Dazu arbeiten wir zunächst mit einem Docstring. Er beschreibt kurz, was die Funktion tut und welche Bedeutung ihre Argumente haben. Wenn Sie die Funktion in die Python-Konsole kopieren und dort ausführen, können Sie im Anschluss mit `help(erzeuge_website)` die Hilfe der Funktion betrachten, die sich aus genau diesem Docstring speist. Zur weiteren Dokumentation haben wir in den Funktionskopf Typhinweise für die Argumente eingebaut.

Auf eine Kommentierung des Codes im Funktionsrumpf wurde verzichtet, da der Code recht einfach ist.

### ■ Aufgabe 23.2

- Die Funktion `wuerfel()` erzeugt mit Hilfe der aus dem Modul `random` importierten Funktion `randint()` eine Zufallszahl, wie man sie auch beim Würfel erhalten könnte. Diese Zufallszahl wird in der Integer-Variablen `wuerfel_ergebnis` abgelegt. Allerdings haben wir vergessen, das Ergebnis auch mit `return` zurückzugeben. Wenn Sie diese Funktion aufrufen und ihren Rückgabewert in einer Variablen auffangen oder in der Konsole ausgeben, werden Sie feststellen, dass Sie jedes Mal den besonderen Wert `None` erhalten. Er zeigt an, dass die Funktion keinen Wert zurückgibt. Fügen Sie nun aber die fehlende `return`-Anweisung ein, so erhalten Sie tatsächlich wie beim Würfeln einen zufälligen Ganzahlwert zwischen 1 und 6 zurück.
- Die Funktion `erzeuge_telefonnummer()` leidet an zwei Problemen: Zum einen steht der Doppelpunkt im Funktionskopf *vor* dem Typhinweis, obwohl er syntaktisch korrekt eigentlich am Ende des Funktionskopfes platziert werden müsste; schließlich leitet er ja den folgenden Code-Block, also den Funktionsrumpf, ein. Zum anderen wird im Rumpf der Funktion das Argument `nummer` verwendet, als die eigentliche Teilnehmeranschlussnummer ohne Vorwahlen. Die taucht aber in der Argumente-Liste im Funktionskopf überhaupt nicht auf. Selbstverständlich kann im Code der Funktion nicht auf ein Argument zugegriffen werden, dass der Funktion gar nicht übergeben wird. Nach diesen Korrekturen kann die Funktion dann aufgerufen werden, um eine hübsch formatierte Telefonnummer zu erzeugen, zum Beispiel bei diesem Aufruf:

```
print(erzeuge_telefonnummer('DE', '0171', '3456789'))
```

Der Ländercode 'DE' wird dabei mit Hilfe eines Dictionaries in die entsprechende Ländervorwahl übersetzt, die führende 0 der Vorwahl wird mit Hilfe des Ausdrucks `vorwahl[1:]`, der alle Zeichen zwischen dem zweiten um dem String-Ende selektiert, abgetrennt.

### ■ Aufgabe 23.3

Die Funktion `selbstbeschaeftigung()` könnte so aussehen:

```
def selbstbeschaeftigung(**args):
    print(list(args.keys()))
    print(list(args.values()))
```

Wie Sie sich erinnern, kann auf eine unbestimmte Liste benannter, also Schlüsselwort-Argumente, mit einem Argument zugegriffen werden, dem im Funktionskopf zwei Sternchen vorangestellt werden. Dieses Argument ist dann ein Dictionary mit den Namen der Argumente als Schlüsseln und den übergebenen Werten als Werten des Dictionary-Einträge. Dementsprechend können wir uns mit Hilfe der Methoden **keys()** und **values()** die Schlüssel, also die Argumente-Namen und ihre Werte ermitteln lassen. Ein Aufruf dieser Funktion könnte dann so aussehen:

```
selbstbeschaeftigung(erstes_argument ='Ein str-Argument',
                     noch_ein_argument2 = 5)
```

#### ■ Aufgabe 23.4

Dieses Programm, das neue Verkäufe „verbucht“, erzeugt folgende Ausgaben:

```
24.08.2019 12:48:24 -- Neuer Umsatz: 10.99 Euro mit
Artikelnummer DE07011981.
24.08.2019 12:48:24 -- Neuer Umsatz: 24.99 Euro mit
Artikelnummer DE25101878.
Gesamtumsatz: 35.98
Umsatz mit letztem Verkauf: 0.0
```

Die ersten beiden Ausgaben werden direkt von der Funktion **buche\_verkauf()** generiert. Die folgenden beiden Ausgaben, nämlich zum Gesamtumsatz und dem Umsatz des letzten Verkaufs, sind lediglich Ausgaben der Variablen **umsatz\_gesamt** und **letzter\_verkauf**. Diese beiden Variablen werden im Hauptprogramm angelegt und zunächst mit dem Wert **0.00** vorbelegt. Die Funktion versucht dann, diese Variablen zu verändern, was ihr im Fall von **umsatz\_gesamt** auch zu gelingen scheint. Zum derzeitigen Wert von **buche\_verkauf()** addiert sie jeweils den Wert des neu verbuchten Verkaufs. Die Summe unserer beiden Verkäufe ist tatsächlich 35.98. Was aber ist mit **letzter\_verkauf**? Obwohl auch dieser Variablen im Code der Funktion **buche\_verkauf()** ein neuer Wert zugewiesen wird, hat sie am Ende immer noch jenen Wert, mit dem sie zu Beginn des Programms initialisiert wurde. Was ist passiert? Tatsächlich sind die beiden Variablen, die zu Beginn initialisierte und die im Funktionsrumpf von **buche\_verkauf()** verwendete, zwei unterschiedliche Objekte. Die im Funktionsrumpf verwendete Variable ist eine *lokale* Variable, die am Ende der Funktion zu existieren aufhört. Wertezuweisungen auf diese Variable haben keinen Effekt auf die zu Beginn des Programms, außerhalb der Funktion initialisierte Variable. Wollten wir stattdessen diese Variablen verändern, müssten

wir Python das mit einer **global**-Anweisung mitteilen, wie wir es auch für **umsatz\_gesamt** getan haben. Dadurch weiß Python, dass wir keine lokale Variable erzeugen, sondern in die gleichnamige globale Variable schreiben wollen, die außerhalb der Funktion angelegt worden ist.



# Wie steuere ich den Programmablauf und lasse das Programm auf Benutzeraktionen und andere Ereignisse reagieren?

## Inhaltsverzeichnis

- 24.1 if-else-Konstrukte – 386**
  - 24.1.1 Einfache if-else-Konstrukte – 386
  - 24.1.2 Verschachtelte if-else-Konstrukte – 389
  - 24.1.3 if-else-Konstrukte mit zusammengesetzten Bedingungen – 390
  - 24.1.4 if-else-Konstrukte mit alternativen Bedingungen (elif) – 392
- 24.2 Ereignisse – 394**
- 24.3 Zusammenfassung – 394**
- 24.4 Lösungen zu den Aufgaben – 395**

## Übersicht

Als nächstes werden wir uns damit beschäftigen, wie man in Python-Programm verzweigen und – je nach Situation – mal den einen, mal den anderen Code-Teil ausführen kann. Erst dadurch wird unser Programmablauf richtig interessant und ist nicht einfach nur die immer gleiche Ausführung derselben Folge von Python-Anweisungen.

In diesem Kapitel werden Sie lernen:

- wie Sie im Programmcode mit **if-else**-Konstrukten verzweigen können
- wie **if-else**-Konstrukte ineinander verschachtelt werden
- wie in **if-else**-Konstrukten mit **elif** mehrere alternative Bedingungen berücksichtigt werden können
- wie Bedingungen aufgebaut werden und welche Vergleichsoperatoren dabei eingesetzt werden können
- wie Sie mehrere Bedingungen mit Hilfe logischer Operatoren zur einer Gesamtbedingung verknüpfen können
- wie Sie Ihr Programm auf Ereignisse reagieren lassen.

## 24.1 if-else-Konstrukte

### 24.1.1 Einfache if-else-Konstrukte

Greifen wir nochmal unsere Umrechnung von Temperaturen zwischen den Einheiten Kelvin und Grad Celsius auf. Dazu hatten wir im vorangegangenen Kapitel (► Abschn. 23.1.3) folgende Funktion definiert:

```
def kelvin_zu_celsius(kelvin: float) -> float:
    return kelvin - 273.15
```

Angenommen nun, wir wollten diese einfache Funktion verbessern, indem wir sie weniger anfällig für Fehleingaben machen. Eine Temperatur in Kelvin kann, wie Sie sich erinnern, niemals negativ sein. Der absolute Nullpunkt liegt bei 0 Kelvin, was 273,15 Grad Celsius entspricht. Niedrigere Temperaturen sind physikalisch unmöglich. Bei 0 Kelvin ist einfach überhaupt keine Wärme mehr vorhanden, kälter kann es dementsprechend nicht mehr werden.

Wenn wir also verhindern wollen, dass unsere Funktion einen unzulässigen Wert zurückgibt, weil das Funktionsargument bereits unzulässig war, müssen wir prüfen, ob das Funktionsargument **kelvin** größer oder gleich 0 ist. Ist **kelvin** kleiner 0, also unzulässig, soll unsere Funktion darauf reagieren; sie könnte das tun, indem sie eine Fehlermeldung in der Konsole anzeigt, oder aber einen speziellen Fehlercode zurückgibt, anhand dessen der Programmierer, der unsere Funktion einsetzt, prüfen kann, ob die Umrechnung erfolgreich war. Wir werden im Folgenden den zweitgenannten Weg beschreiben, der deshalb vorzuziehen ist, weil er es dem Verwender unserer Funktion erlaubt, auf einen Fehler so zu reagieren, wie er es

## 24.1 · if-else-Konstrukte

will, während wir ihn beim erstgenannten Weg dazu zwingen würden, unsere Fehlermeldung zu „genießen“, die er vielleicht so gar nicht haben möchte (vielleicht arbeitet er ja in einer ganz anderen Sprache?).

Um unsere Funktion nun robust gegenüber Fehleingaben zu machen, müssen wir ein Wenn-Dann-Konstrukt ergänzen, das prüft, ob der **kelvin**-Wert unzulässig, also kleiner als 0 ist, und in diesem Fall einen Fehlercode zurückzugeben, zum Beispiel **-1000**. Die angepasste Funktion könnte dann so aussehen:

```
def kelvin_zu_celsius(kelvin: float) -> float:
    if kelvin < 0:
        erg = -1000
    else:
        erg = kelvin - 273.15
    return erg
```

Sie sehen, dass wir hier eine Programmverzeigung eingebaut haben: Auf das Schlüsselwort **if** (*wenn*) folgt die zu prüfende Bedingung, in unserem Fall, ob die Kelvin-Temperatur kleiner 0 ist. Ist die Bedingung erfüllt, wird der folgende Code-Block ausgeführt, der – wie bereits die Code-Blöcke bei Funktionen – nach einem Doppelpunkt beginnt und eingerückt ist. In unserem Beispiel umfasst der Code-Block nur eine einzelne Anweisung nämlich eine Zuweisung, mit der wir einer Zwischenvariable namens **erg** den späteren Rückgabewert der Funktion (hier als den Fehlercode) zuweisen, jedoch könnten hier natürlich beliebig viele weitere Anweisungen stehen. Es folgt – auf der gleichen Einrückungsebene wie **if** – das **else**-Schlüsselwort (*anderenfalls*), wiederum gefolgt von einem Code-Block, der nur dann ausgeführt wird, wenn die **if**-Bedingung nicht erfüllt ist. Der **else**-Zweig ist gewissermaßen die logische Umkehrung der **if**-Bedingung, immer wenn die **if**-Bedingung nicht erfüllt ist, springt das Programm direkt zum **else**, ohne den Programmcode im **if**-Code-Block auszuführen. In unserem Beispiel bedeutet ein Durchlaufen des **else**-Code-Blocks, dass unsere **kelvin**-Argument zulässig ist und damit in einen Celsius-Wert umgerechnet werden kann.

Die **return**-Anweisung nach dem **if-else**-Konstrukt wird auf jeden Fall ausgeführt, ganz gleich, ob das Programm in den **if**- oder in den **else**-Ast verzweigt. So stellen wir sicher, dass stets der in **erg** gespeicherte Funktionswert zurückgegeben wird, der – je nachdem, ob der **if**- oder der **else**-Zweig durchlauen worden ist, den umgerechneten Celsius-Wert oder den Fehlercode **-1000** enthält.

In unser Hauptprogramm könnten wir diese Funktion nun zum Beispiel so einbinden:

```
kelv = input(
    'Bitte geben Sie eine Temperatur in Kelvin ein: ')
cel = kelvin_zu_celsius(float(kelv))
if cel == -1000:
    print('Sie haben eine unzulässige Kelvin-Temperatur \
          eingegeben!')
else:
    print(round(float(kelv), 2), 'Kelvin sind',
          round(cel, 2), 'Grad Celsius.')
```

Wie Sie sehen, fragen wir vom Benutzer eine Temperatur in Kelvin ab, rechnen diese um und schauen, dann, ob dabei ein Fehler aufgetreten ist; je nachdem, ob das der Fall ist, geben wir entweder eine entsprechende Fehlermeldung oder aber den in Celsius umgerechneten Wert aus.

24

Ihnen wird das doppelte Gleichheitszeichen aufgefallen sein, dass wir zum Vergleich der Variable **cel** mit dem Fehlercode **-1000** benutzen. Python unterscheidet, wie viele andere Sprachen auch, zwischen dem *Zuweisungsoperator* **=** und dem *Vergleichsoperator* **==**. Dem mathematischen Ungleichheitszeichen (**≠**) entspricht in Python der Vergleichsoperator **!=**, also „nicht gleich“, denn das Ausrufezeichen ist, wie in vielen anderen Programmiersprachen auch, der *logische Operator NICHT*, der den Wahrheitsgehalt einer Aussage herumdreht, in diesem Fall also der Aussage, dass die beiden Werte links und rechts vom Gleichheitszeichen-Operator gleich sind.

Auf den **else**-Zweig in einem **if-else**-Konstrukt kann übrigens auch verzichtet werden. Unsere Funktion **kelvin\_zu\_celsius()** könnte also auch so aussehen:

```
def kelvin_zu_celsius(kelvin: float) -> float:
    erg = -1000
    if kelvin >= 0:
        erg = kelvin - 273.15
    return erg
```

Diese Formulierung der Funktion leistet exakt dasselbe wie die vorangegangene Option. Wir setzen zunächst die Variable **erg**, unseren späteren Rückgabewert, auf **-1000**, also auf Fehler. Wenn jetzt nichts mehr passiert, gibt die Funktion also den Fehlerwert zurück. Es passiert aber dann noch etwas, wenn das **kelvin**-Argument ein zulässiger Wert ist. Dann nämlich wird der Wert von **erg** (derzeit der Fehlercode) durch den in Celsius umgerechneten Wert ersetzt.

Übrigens: Natürlich hätten wir die gesamte Logik der Prüfung, ob die Benutzereingabe zulässig ist, oder nicht, von der Funktion **kelvin\_zu\_celsius()** auch ins Hauptprogramm verlegen und die Funktion einfach immer nur dann aufrufen können, wenn die Eingabe zulässig war. Andernfalls würde eben eine entsprechende Fehlermeldung ausgegeben.

### ? 24.1 [15 min]

Modifizieren Sie den Code des Texteditors aus ► Abschn. 22.4 so, dass

- eine Datei nur dann beim Öffnen gelesen und beim Speichern geschrieben wird, wenn auch tatsächlich ein Dateiname vorhanden ist (also zum Beispiel der Benutzer im Datei-Öffnen-Dialog nicht auf „Abbrechen“ geklickt hat, ohne eine Datei auszuwählen);
- ein Klick auf „Speichern“ den „Datei speichern unter“-Dialog öffnet, wenn bislang noch kein Dateiname angegeben worden ist.

Durch diese Ergänzungen vermeiden Sie die unschönen Fehlermeldungen in der Run-Konsole, wenn der Benutzer beispielsweise auf „Speichern“ klickt, ohne dass zuvor ein Dateiname festgelegt worden ist; in diesem Fall nämlich versucht Python, eine Datei mit leerem Dateinamen (") zu öffnen, was natürlich fehlschlägt.

Konstellationen „abzufangen“, in denen die weitere Ausführung des Programms zu einem Fehler führen würde, ist ein häufiges Einsatzgebiet von Verzweigungen mit Hilfe von **if-else**-Konstrukten.

### 24.1.2 Verschachtelte if-else-Konstrukte

Angenommen nun, wir wollten unsere Funktion `celsius_zu_kelvin()` zu einer Funktion `temperatur_umrechnen(temperatur: float, nachCelsius: bool)` umbauen, die eine Temperaturangabe wahlweise von Kelvin nach Celsius oder von Celsius nach Kelvin umrechnet. In welche Richtung umgerechnet werden soll, soll ein neues **bool**-Argument `nachCelsius` angeben. Besitzt dieses Argument den Wert `True`, so wird von Kelvin *nach Celsius* umgerechnet, anderenfalls umgekehrt. Auch in der neuen Funktion `temperatur_umrechnen()` wollen wir sicherstellen, dass als Argument `temperatur` nur zulässige Temperaturen angegeben werden, also Temperaturen, die größer 0 sind, wenn es sich um einen Kelvin-Wert handelt (`nachCelsius == True`) bzw. größer als -273.15 Grad, falls `temperatur` einen Celsius-Wert beinhaltet (`nachCelsius == False`).

Unsere Funktion könnte dann so aussehen:

```

1 def temperatur_umrechnen(temperatur: float,
2                             nachCelsius: bool) -> float:
3     if nachCelsius == True:
4         if temperatur >= 0:
5             erg = temperatur - 273.15
6         else:
7             erg = -1000
8     else:
9         if temperatur >= -273.15:
10            erg = temperatur + 273.15
11        else:
12            erg = -1000
13    return erg

```

Die Funktion könnten wir nun beispielsweise mit `print(temperatur_umrechnen(100, False))`

Aufrufen, um 100 Grad Celsius nach Kelvin umzurechnen.

Wenn Sie sich die Funktion genauer ansehen, stellen Sie fest, dass hier zwei **if-else**-Konstrukte ineinander verschachtelt worden sind: Das „äußere“ prüft (Zeile 3), in welche Richtung der Anwender die Temperatur umrechnen möchte und verzweigt entsprechend, die „inneren“ Konstrukte (Zeilen 3 und 9) prüfen, ob das Argument

**temperatur** einen im jeweiligen Programmzweig gültigen Wert besitzt. Ist dies der Fall, wird die Umrechnung durchgeführt und in der lokalen Variable **erg** gespeichert (Zeilen 5 und 10), anderenfalls wird **erg** mit dem Fehlercode **-1000** („ungültige Temperaturangabe“) belegt (Zeilen 7 und 12).

Dem „äußerem“ **if** und dem „äußerem“ **else** folgt also jeweils ein (natürlich eingerückter) Code-Block, der jeweils wiederum ein weiteres **if-else**-Konstrukt enthält. Der Verschachtelungstiefe sind dabei theoretisch zwar keine Grenzen gesetzt, praktisch wird der Code aber natürlich mit zunehmender Verschachtelungstiefe immer schwerer lesbar, fast wie ein natürlichsprachlicher Satz, der viele Nebensätze aufmacht und am Satzende in schneller Folge wieder schließt. Es empfiehlt sich der Übersichtlichkeit wegen spätestens bei mehr als zwei ineinander verschränkten Konstrukten eine entsprechende Kommentierung vorzunehmen, insbesondere, um zu beschreiben, zu welcher Bedingung die vielen unterschiedlichen **else** eigentlich gehören. Gleiches gilt, wenn der Code innerhalb der **if** und der **else**-Blöcke lang ist (im Beispiel oben umfassen diese Code-Blöcke ja jeweils nur eine einzelne Zeile), weil man dann viel Code liest und plötzlich auf ein **else** stößt, von dem man, ohne viel zu scrollen, gar nicht mehr weiß, zu welcher **if**-Bedingung es eigentlich gehört.

Übrigens: Die „äußere“ Bedingung hätten wir statt **if nachCelsius == True** auch einfacher als **if nachCelsius** formulieren können; der Vergleich mit dem Wert **True** darf also entfallen. Das liegt daran, dass Python, wenn kein Vergleichswert angegeben ist, standardmäßig mit **True** vergleicht.

### 24.1.3 if-else-Konstrukte mit zusammengesetzten Bedingungen

Die Funktion **temperatur\_umrechnen()** aus dem vorangegangenen Abschnitt hätten wir auch so schreiben können:

```

1 def temperatur_umrechnen(temperatur: float,
2                           nachCelsius: bool) -> float:
3     if nachCelsius == True and temperatur >= 0:
4         erg = temperatur - 273.15
5     else:
6         if nachCelsius == False and temperatur >= -273.15:
7             erg = temperatur + 273.15
8         else:
9             erg = -1000
10    return erg

```

Diese Formulierung der Funktion leistet dasselbe wie die Funktion oben, ist aber in Hinblick auf die Verschachtelung der **if-else**-Konstrukte anders aufgebaut. Das „äußere“ Konstrukt prüft gleich zwei Bedingungen auf einmal, nämlich, ob eine Umrechnung nach Celsius gewünscht ist (**nachCelsius == True**) und ob die angegebene (Kelvin-)Temperatur dafür zulässig ist (**temperatur >=0**). Diese beiden (Teil)-

## 24.1 · if-else-Konstrukte

Bedingungen werden mit **and** zu einer Gesamtbedingung verknüpft, die genau dann wahr ist, wenn *beide* Teilbedingungen wahr sind. In diesem Fall wird die Umrechnung vorgenommen (Zeile 4). Ist aber mindestens eine Teilbedingung nicht erfüllt, also entweder **nachCelsius == False** oder aber die Temperatur  $< 0$  (oder sogar beides!), wird das Programm im **else**-Block des äußeren **if-else**-Konstrukts fortgesetzt (Zeile 6).

Dort wird nun wiederum eine zusammengesetzte Bedingung geprüft, nämlich, ob eine Umrechnung von Celsius nach Kelvin gefordert ist und ob das Argument **temperatur** für diese Umrechnung zulässig ist. Sind beide Teilbedingungen erfüllt, wird die Umrechnung vorgenommen (Zeile 7). Ist aber auch diese Gesamtbedingung nicht erfüllt, geht das Programm in den „innersten“ **else**-Block: Da **nachCelsius** ja nur die Werte **True** oder **False** annehmen kann (zumindest bei „sachgemäßem“ Aufruf der Funktion), bleibt jetzt nur noch die Möglichkeit, dass das Argument **temperatur** einen für die geforderte Umrechnung unzulässigen Wert besitzt. Also wird in diesem Fall die Variable **erg** mit dem Fehlercode **-1000** belegt (Zeile 9).

In diesem Beispiel haben wir zwei Teilbedingungen mit einem logischen *UND* (**and**) verknüpft. Wichtige weitere logische Operatoren neben dem **and** sind **or**, das logische *ODER*, mit dem zwei Bedingungen dergestalt verknüpft werden, dass die Gesamtbedingung genau dann wahr ist, wenn *entweder* die eine oder die andere oder *beide* Bedingungen wahr sind, und **not**, das logische *NICHT*, mit dem sich der Wahrheitsgehalt einer Aussage umkehren lässt. Die Bedingung **nachCelsius == True and temperatur >= 0** hätte man also – zugegebenermaßen etwas umständlich – zum Beispiel auch schreiben können als **not(nachCelsius == False or temperatur < 0)**. Dann hätte die Gesamtbedingung also gefordert, dass es nicht zutreffen darf, dass entweder **nachCelsius** gleich **False** ist, oder **temperatur < 0** ist, oder beides. Anders herum ausgedrückt: Nur, wenn **nachCelsius == True** ist und **temperatur >= 0**, ist die Gesamtbedingung erfüllt und der darauffolgende Code-Block (nämlich die Umrechnung von Kelvin in Grad Celsius) wird ausgeführt.

Übrigens: Nicht immer müssen Sie zusammengesetzte Bedingungen mit logischen Operatoren verknüpfen. Statt zum Beispiel

```
if x >= 0 and x < 10
```

zu schreiben, erlaubt Python auch die kompaktere Schreibweise

```
if 0 <= x < 10
```

was exakt dieselbe Prüfung veranlasst, nämlich, ob der Wert der Variable **x** mindestens 0 aber kleiner als 10 ist.

### 24.2 [10 min]

Schreiben Sie eine Funktion **alter\_in\_sekunden(alter\_jahre: int) -> int**, die eine Altersangabe in Jahren, die als Argument **alter\_jahre** übergeben wird, in Sekunden

umrechnet; die Funktion soll also die *mindestens* bereits erlebten Sekunden errechnen und zurückgeben. Dabei sollen spezifische Fehlermeldungen ausgegeben werden, wenn das Argument **alter\_jahre** keine Integer-Variable ist, oder das Alter kleiner als 0 oder größer als 120 ist.

Wenn Ihnen nicht mehr präsent ist, wie Sie prüfen können, ob ein Wert eine Ganzzahl ist, blättern Sie noch einmal zurück zu ► Abschn. 21.4.1.

#### 24.1.4 if-else-Konstrukte mit alternativen Bedingungen (elif)

24

Manchmal möchte man mehrere gleichartige Bedingungen prüfen. Viele andere Sprachen bieten dafür, wie wir in ► Abschn. 14.6 gesehen haben, ein Verzweigung-Fall-Konstrukt, das sich hervorragend dazu eignet, einen Ausdruck (zum Beispiel eine Variable) auf mehrere unterschiedliche Werte hin zu prüfen.

Ein solches Konstrukt gibt es in Python nicht. Allerdings gibt es mit dem Schlüsselwort **elif** die Möglichkeit, sehr effizient alternative Bedingungen zu der **if**-Bedingung eines **if-else**-Konstrukts abzuprüfen; diese Art, die Bedingungen zu formulieren, ist sogar flexibler als ein herkömmliches Verzweigung-Fall-Konstrukt.

Nehmen wir als Beispiel an, Sie hätten ein Programm entwickelt, das es dem Benutzer erlaubt, eine Datei auszuwählen und dann anzugeben, was mit der Datei geschehen soll. Dabei kann die Datei umbenannt, gelöscht, in ein anderes Verzeichnis kopiert oder dorthin verschoben werden. Um die gewünschte Aktion auszuwählen, gibt der Benutzer einfach den ersten Buchstaben der gewünschten Aktion an, also **u** (umbenennen), **l** (löschen), **k** (kopieren) oder **v** (verschieben). Nach der Eingabe des Benutzers ist dieser Aktionswunsch dann in einer **str**-Variablen **aktion** gespeichert.

Wie sieht nun die Bedingung aus, die prüft, welche Aktion der Benutzer ausführen möchte? Mit den verschalteten **if-else**-Konstrukten aus ► Abschn. 24.1.2 könnten wir folgende Formulierung wählen:

```
if aktion == 'u':
    # Code, der ausgeführt wird, wenn der Benutzer 'u'
    # eingegeben hat, also Umbenennen der Datei
else:
    if aktion == 'l':
        # Code, der ausgeführt wird, wenn der Benutzer 'l'
        # eingegeben hat, also Löschen der Datei
    else:
        if aktion == 'k':
            # Code, der ausgeführt wird, wenn der Benutzer 'k'
            # eingegeben hat, also Kopieren der Datei
        else:
            if aktion == 'v':
                # Code, der ausgeführt wird, wenn der Benutzer
                # 'v' eingegeben hat, also Verschieben der Datei
            else:
                # Code, der ausgeführt wird, wenn der Benutzer
                # keinen der zulässigen Aktioncodes eingegeben
                # hat
```

## 24.1 · if-else-Konstrukte

Diese verschaltete Konstruktion prüft der Reihe nach, welcher Aktionscode eingegeben worden ist. Wurde nicht '**u**' eingegeben, wird als nächstes '**l**' geprüft, wurde auch dieses nicht eingegeben, dann '**k**' und so weiter.

Durch die Verschachtelung wirkt das Ganze einigermaßen unübersichtlich. Mit Hilfe des **elif**-Schlüsselwortes gelingt es, diesen Programmteil deutlich klarer zu strukturieren:

```
if aktion == 'u':
    # Umbenennen
elif aktion == 'l':
    # Löschen
elif aktion == 'k':
    # Kopieren
elif aktion == 'v':
    # Verschieben
else:
    # Kein gültiger Aktionscode
```

Mit **elif** können – auf der gleichen Ebene wie das einleitende **if** – weitere Bedingungen geprüft werden. Der (optionale) **else**-Code-Block am Ende tritt dann in Aktion, wenn weder die **if**-Bedingung noch eine der **elif**-Bedingungen angeschlagen hat. Trifft eine der **elif**-Bedingungen zu, so wird deren Code-Block ausgeführt und das Programm dann nach dem **else**-Code-Block, also hinter dem **if-elif-else**-Konstrukt fortgesetzt. Die weitere **elif**-Bedingungen werden nicht mehr geprüft, denn **elif** formuliert *alternative* Bedingungen.

Auf diese Weise lässt sich ein stark verschachteltes und schwer zu verstehendes Konstrukt durch eine sehr übersichtliche, leicht lesbare Schreibweise ersetzen.

### 24.3 [10 min]

Betrachten Sie das **if-elif-else**-Konstrukt im folgenden Programmauszug.

- Welche Werte nehmen die Variablen **a** und **b** nach dem Durchlaufen des Programmauszugs an, wenn ihre Werte zu Beginn **a = 100, b = 50** sind?
- Welche Werte nehmen die Variablen **a** und **b** nach dem Durchlaufen des Programmauszugs an, wenn ihre Werte zu Beginn **a = 110, b = 40** sind?
- Welche Wertekombinationen von **a** und **b** führen dazu, dass am Ende **b = 0** ist?

Wenn Sie bei den Teilaufgaben a. und b. nicht direkt auf die Lösung kommen, geben Sie den Code in Python ein und probieren Sie es aus.

```
if a > 90 and b <= 20:
    b = 15
elif b < 10:
    b = 0
elif (a > 90 and b < 50) or (a == 100 and b > 50):
    b = 20
    if a >= 100 or b <= 50:
        a = 5
    else:
```

```

        a = 25
elif a >= 90 or b <= 50:
    b = 5
else:
    b = 10
    a = 20

```

**24**

## 24.2 Ereignisse

---

Ebenso wie **if-(elif-)else**-Konstrukte dienen *Ereignisse* dazu, den Programmablauf zu steuern, um zum Beispiel auf Eingaben des Benutzers reagieren zu können. Anders allerdings als bei den **if-(elif-)else**-Konstrukten, wird bei Ereignissen das Programm nicht einfach linear durchlaufen und dabei bestimmte Teile durch Verzweigungen ausgeführt, während andere Programmteile durch die Verzweigung gleichsam „übersprungen“ werden. Ereignisse führen dazu, dass eine bestimmte Funktion, der *Event Handler* aufgerufen und der darin enthaltene Code ausgeführt wird.

Sehr schön sieht man das an unseren **tkinter**-Programmen wie etwa der Taschenrechner-Applikation aus ► Abschn. 22.2.6. Die Funktion **gleich\_press()** zum Beispiel ist ein Event Handler, der immer dann aufgerufen wird, wenn der Benutzer auf den Button mit dem Gleichheitszeichen geklickt hat. Ist der Code im Event Handler vollständig abgearbeitet, springt das Programm zurück in die Hauptschleife des Programms – in **tkinter**-Programmen ist das die Funktion **mainloop()**. In dieser Warteschleife „lauert“ das Programm darauf, dass das nächste Ereignis eintritt, für das ein Event Handler definiert ist. Sobald das geschieht, bekommt der Event Handler die Kontrolle; wenn der durchgelaufen ist, geht das Programm wieder in den „Lauermodus“. In ► Abschn. 25.2 werden wir mit einfachen Mitteln ein Programm mit Hauptschleife und Event Handlern nachbauen, das in der Python-(Run-)Konsole läuft, also keine grafische Benutzeroberfläche besitzt.

## 24.3 Zusammenfassung

---

In diesem Kapitel haben wir uns damit beschäftigt, wie im Programmcode Verzweigungen eingebaut werden können, sodass nicht immer der gesamte Code, sondern, abhängig von Bedingungen, nur einzelne Teile ausgeführt werden. Die zentralen Werkzeuge dazu sind das **if-else**-Konstrukt und Ereignisse.

Folgende Punkte sollten Sie aus diesem Kapitel unbedingt mitnehmen:

- **if-else**-Konstrukte erlauben es, die Ausführung von Teilen des Programmcodes von Bedingungen abhängig zu machen; dabei folgt hinter **if** und hinter **else** jeweils der Code-Block der ausgeführt werden soll, wenn die Bedingung bzw.

## 24.4 · Lösungen zu den Aufgaben

ihrer Alternative erfüllt ist; auf den **else**-Zweig kann auch verzichtet werden; der allgemeine Aufbau ist also: **if bedingung: code\_block else: code\_block**.

- **if-else**-Konstrukte können ineinander verschachtelt werden.
- Mit **elif** können mehrere weitere Bedingungen formuliert werden, die nur geprüft werden, wenn die **if**-Bedingung und ggf. weitere, vorangegangene **elif**-Bedingungen nicht erfüllt waren.
- Bedingungen werden regelmäßig durch Vergleiche gebildet; der Operator zur Prüfung auf Gleichheit ist dabei das doppelte Gleichheitszeichen (**==**), das einfache Gleichheitszeichen wird für Zuweisungen verwendet; für Ungleichheitsrelationen kommt **!=** zum Einsatz.
- Mehrere Bedingungen können mit logischen Operatoren (vor allem **and**, **or**, **not**) zu einer Gesamtbedingung verknüpft werden.
- Ereignisse, wie sie vor allem bei Programmen mit grafischer Benutzeroberfläche Sinn machen, werden durch die Definition einer speziellen Funktion (Event Handler) verarbeitet, die immer dann aufgerufen wird, wenn das Ereignis ausgelöst wurde.

## 24.4 Lösungen zu den Aufgaben

### Aufgabe 24.1

Angepasst werden müssen die (Event-Handler-)Funktionen **speichernunter\_press()**, **speichern\_press()** und **oeffnen\_press()**. Die ergänzten **if-else**-Konstrukte sind in der folgenden Beispiellösung kursiv hervorgehoben:

```
def speichernunter_press():
    dname = asksaveasfilename(defaultextension = 'txt',
                               filetypes = [
                                   ('Textdateien', '*.txt'),
                                   ('Alle Dateien', '*.*')],
                               title='Datei speichern unter...', initialdir='C:\\Windows')

    if dname != '':
        datei = open(dname, 'w')
        datei.write(text.get(1.0, END))
        datei.close()

    status['text'] = 'Datei "' + dname + '" gespeichert.'
    global dateiname
    dateiname = dname
```

```

def speichern_press():
    global dateiname
    if dateiname != '':
        datei = open(dateiname, 'w')
        datei.write(text.get(1.0, END))
        datei.close()

        status['text'] = 'Datei "' + dateiname + '" gespeichert.'
    else:
        speichernunter_press()

def oeffnen_press():
    global dateiname
    dname = askopenfilename(defaultextension = 'txt',
                            filetypes = [
                                ('Textdateien', '*.txt'),
                                ('Alle
Dateien', '.*')],
                            title='Datei öffnen...', 
                            initialdir='C:\\Windows')

    if dname != '':
        datei = open(dname, 'r')
        text.delete(1.0, END)
        text.insert(1.0, datei.read())
        datei.close()

        status['text'] = 'Datei "' + dname + '" geöffnet.'
        dateiname = dname

```

### ■ Aufgabe 24.2

Eine mögliche Formulierung der Funktion könnte so aussehen:

```

def alter_in_sekunden(alter_jahre: int) -> int:
    if not isinstance(alter_jahre, int):
        print('Fehler: Die Altersangabe ist keine Ganzzahl!')
    else:
        if alter_jahre < 0 or alter_jahre > 120:
            print('Fehler: Unplausibel Altersangabe!')
        else:
            return alter_jahre * 365 * 24 * 60 * 60

```

Dabei wird zunächst in einem äußeren **if-else**-Konstrukt mit Hilfe der Funktion **isinstance()** geprüft, ob das Argument **alter\_jahre** kein (!) Ganzzahlwert ist. Ist das der Fall, wird eine Fehlermeldung ausgegeben und die Funktion verlassen. Wir machen uns hier den Umstand zu Nutze, dass **isinstance** einen **bool**-Wert zurückgibt und Python einen Ausdruck in einer Bedingung standardmäßig gegen **True** vergleicht. Alternativ hätten wir natürlich schreiben können **if not isinstance(alter\_jahre, int) == True** oder aber auch **if isinstance(alter\_jahre, int) != True**.

## 24.4 · Lösungen zu den Aufgaben

Ist `alter_jahre` aber ein Ganzzahlwert, wird im inneren `if-else`-Konstrukt geprüft, ob das Alter außerhalb der zulässigen Grenzen liegt. In diesem Fall wird die Funktion wiederum mit einer Fehlermeldung verlassen. Ist aber auch diese Hürde erfolgreich genommen, gibt die Funktion das Alter in Sekunden zurück.

Aufrufen könnte man die Funktion dann zum Beispiel mit

```
print(alter_in_sekunden(38))
```

### ■ Aufgabe 24.3

- a.  $a = 100, b = 5$  (es greift die Bedingung:  $a \geq 90 \text{ or } b \leq 50$ )
- b.  $a = 5, b = 20$  (es greifen die Bedingungen:  $(a > 90 \text{ and } b < 50) \text{ or } (a == 100 \text{ and } b > 50)$  und  $a \geq 100 \text{ or } b \leq 50$ )

Nie. Dazu müsste die Bedingung  $a \geq 95 \text{ and } b < 15$  erfüllt sein. Diese ist aber eine Alternativbedingung zu  $a > 90 \text{ and } b \leq 20$ . Jede Werte-Kombination von  $a$  und  $b$ , die  $a \geq 95 \text{ and } b < 15$  erfüllt, erfüllt auch  $a > 90 \text{ and } b \leq 20$ . Die Alternativbedingung kommt daher niemals zum Zuge, sie ist „toter Programmcode“.



# Wie wiederhole ich Programmanwei- sungen effizient?

## Inhaltsverzeichnis

- 25.1 Abgezählte Schleifen (for) – 400**
  - 25.1.1 Einfache for-Schleifen – 400
  - 25.1.2 Verschachtelte for-Schleifen – 405
  - 25.1.3 Listenkomprehensionsausdrücke – 406
- 25.2 Bedingte Schleifen (while) – 407**
- 25.3 Schleifen vorzeitig verlassen  
und neustarten – 410**
- 25.4 Zusammenfassung – 412**
- 25.5 Lösungen zu den  
Aufgaben – 413**

## Übersicht

Als nächstes wenden wir uns dem Wiederholen von Programmieranweisungen zu. Damit lassen sich viele aufwendige Aufgaben elegant lösen.

Wie die meisten Programmiersprachen kennt Python dazu sowohl abgezählte (**for**-) als auch bedingte (**while**-) Schleifen. Eine Python-Besonderheit sind die auf den **for**-Schleifen basierenden Listenkomprehensionsausdrücke, mit denen man die Erzeugung von Listen auf sehr kompakte Weisen schreiben kann.

In diesem Kapitel werden Sie lernen:

- wie Sie (abgezählte) **for**-Schleifen formulieren
- wie Sie (bedingte) **while**-Schleifen formulieren
- was Listenkomprehensionsausdrücke sind, und wie Sie damit auf elegante Weise Listen erzeugen können
- wie Sie Schleifen oder einzelne Durchläufe von Schleifen vorzeitig beenden können, und wann das nützlich ist.

## 25.1 Abgezählte Schleifen (for)

### 25.1.1 Einfache for-Schleifen

#### ■ Aufbau von for-Schleifen

Abgezählte Schleifen in Python werden mit dem Schlüsselwort **for** eingeleitet; daher werden wir im Weiteren synonym auch einfach von *for-Schleifen* sprechen.

Bei **for**-Schleifen in Python wird nicht etwa eine Laufvariable mit jedem Schleifendurchlauf hochgezählt, wie es in vielen anderen Sprachen der Fall ist, sondern es wird stets ein *Objekt durchlaufen*, das Elemente besitzt, die nacheinander angeprochen werden können. Ein solches Objekt wird in Python auch *iterierbar* genannt, weil man sich schrittweise von einem Element zum nächsten „durchhangeln“ kann.

Beispiele für solche Objekte sind Listen oder Tupel. Sie alle verfügen über Elemente, die sich eine abgezählte Schleife nacheinander vornehmen kann. Dabei kommt es nicht darauf ab, dass die Elemente in dem Objekt in einer festen Reihenfolge vorliegen, wie das bei Listen und Tupel tatsächlich der Fall ist. Entscheidend ist lediglich, dass sich (außer natürlich beim letzten) immer ein *nächstes* Element finden lässt. Das ist auch der Fall zum Beispiel bei Sets und Dictionaries, in denen die Elemente eben *nicht* sequentiell, in einer für den Programmierer ausnutzbaren Reihenfolge abgelegt sind. Trotzdem besitzen die Elemente von Sets und Dictionaries natürlich intern eine Reihenfolge (im Regelfall die, in der die Elemente dem Set bzw. Dictionary hinzugefügt worden sind) und somit lässt sich auch bei diesen Objekten stets ein *nächstes* Element bestimmen.

Kommt Ihnen die Iterierbarkeit von Objekten und das Durchlaufen ihrer Elemente nun reichlich abstrakt vor, keine Sorge: Im folgenden Beispiel sehen Sie, wie einfach **for**-Schleifen in Python aufgebaut sind. Hier haben wir eine Schleife, die

## 25.1 Abgezählte Schleifen (for)

nichts anderes tut, als die Zahlen zwischen 1 und 10 auf dem Bildschirm auszugeben:

```
for i in [1,2,3,4,5,6,7,8,9,10]:  
    print(i)
```

Auch in Python besitzen **for**-Schleifen eine Laufvariable, in unserem Beispiel **i**. Sie wird direkt nach dem Schlüsselwort **for** definiert. Die Laufvariable nimmt bei jedem Schleifendurchlauf den Wert eines Elements des durchlaufenen Objekts an. Das durchlaufene Objekt ist unserem Beispiel eine Liste mit den Zahlen 1 bis 10. Nach der Reihe wird also die Laufvariable **i** nun jeweils mit dem Wert eines Listen-elements beladen; beim ersten Schleifendurchlauf mit 1, beim zweiten mit 2, bis die Laufvariable schließlich mit 10 den Wert des letzten Listenelements annimmt. Das durchlaufene Objekt, unsere Liste, ist iterierbar, Python weiß also immer, welches Element beim nächsten Schleifendurchlauf an der Reihe ist. Mit dem sich jeweils ändernden Wert der Laufvariable können wir natürlich in dem Code-Block, der bei jedem Schleifendurchlauf ausgeführt wird, arbeiten – und genau das tun wir hier im Beispiel: Der Code-Block wird folgt eingerückt nach dem mit dem Schlüsselwort **for** eingeleiteten und mit Doppelpunkt abgeschlossenen *Kopf* der Schleife. In unserem Beispiel besteht der Code-Block nur aus einer einzigen Anweisung, nämlich der Ausgabe der Laufvariable. Wenn Sie den Code des Beispiels ausführen, erhalten Sie als Output erwartungsgemäß die Zahlen von 1 bis 10.

Nicht immer ist es praktikabel, eine Liste von Zahlen so explizit anzugeben, wie wir das in unserem Beispiel gemacht haben, insbesondere dann nicht, wenn die die Liste sehr lang ist, oder aber ihre Grenzen zum Zeitpunkt der Entwicklung des Programms noch gar nicht feststehen (sondern sich aus Variablen im Programm ergeben). In diesen Fällen ist die Funktion **range(von, bis, schrittweite=1)** ein nützliches Werkzeug. Sie erzeugt eine Folge von Zahlen zwischen **von** und **bis** im Abstand von **schrittweite**, wobei letzteres ein optionales Argument ist und als **1** angenommen wird, sollte es beim Aufruf von Range nicht ausdrücklich angegeben werden. Achtung: Der Wert **bis** selbst gehört nicht zu der erzeugten Zahllensequenz (Sie erkennen hier vielleicht die Analogie zur Indizierung mit einem Indexbereich; in ► Abschn. 21.6.1.2 hatten wir diese Art der Indizierung mit Hilfe des Doppelpunktoperators besprochen). Einfach vergegenwärtigen können Sie sich das, wenn Sie in der Python-Konsole **list(range(1,10))** eingeben. Sie erhalten eine Folge ganzer Zahlen von 1 bis 9 (das Rückgabebjekt von **range()** muss zur Ausgabe zunächst in eine Liste konvertiert werden, denn die **print()**-Methode der gleichnamigen Klasse **range** zeigt die Zahlenfolge selbst nicht an – probieren Sie es aus! Iterierbar ist dieses Objekt allerdings sehr wohl).

Damit würde sich unser obiges Beispiel vereinfachen zu:

```
for i in range(1,11):  
    print(i)
```

Betrachten Sie nun das folgende Beispiel, in dem wir während der Schleife den Wert unserer Laufvariable anpassen:

```
mein_range = range(1,11)
for i in mein_range:
    i = 2
print(mein_range)
```

25

An der Ausgabe, die von der abschließenden `print()`-Anweisung erzeugt wird, sieht man, dass sich das Objekt `mein_range` nicht verändert hat, obwohl wir unserer Laufvariable `i`, die bei jedem Schleifendurchlauf ja ein anderes Element von `mein_range` repräsentiert, jeweils den Wert **2** zuweisen. Das allerdings hat offenbar keine Wirkung. Die Ursache liegt darin, dass die Laufvariable in einer `for`-Schleife stets nur eine *Kopie* des Elements darstellt, das gerade im Fokus steht. Wir verändern also durch die Zuweisung `i=2` keineswegs das jeweilige Element unseres `range`-Objekts, sondern lediglich die Laufvariable selbst.

### ■ Beispiele für for-Schleifen

Nun ist die Ausgabe einer Zahlenfolge nicht unbedingt eine besonders nützliche Anwendung von `for`-Schleifen. Deshalb schauen wir uns im nächsten Beispiel einen einfachen Verschlüsselungsalgorithmus an, der auch als *ROT13* bekannt ist. Jedes Zeichen eines Strings kann als ein Zahlencode interpretiert werden, der in dem jeweiligen Zeichensatz das Zeichen eindeutig bestimmt. So repräsentiert zum Beispiel im sehr gebräuchlichen UTF-8-Zeichensatz der Zahlencode 65 das große „A“, 66, das große „B“ und so weiter. Die Kleinbuchstaben folgen ab 97. Diese Konvertierbarkeit von Buchstaben in Zahlen macht sich der ROT13-Algorithmus zu Nutze, indem er einfach jeden Buchstabencode um eine bestimmte Zahl – im Original 13 – erhöht. So wird aus dem „A“ (Code 65) das „N“ (Code 78), aus „B“ (Code 66) ein „O“ (Code 79) usw.

Wir wollen eine Funktion `rot13(s : str, decodieren: bool, verschiebung: int = 13)` entwickeln, die das String-Argument `s` übernimmt, und in Abhängigkeit des `bool`-Arguments `decodieren` diesen String entweder ver- oder entschlüsselt. Dabei soll standardmäßig eine Zeichencode-Verschiebung von 13 verwendet werden, aber auch eine andere Verschiebung mit Hilfe des gleichnamigen Arguments eingestellt werden können.

Natürlich stellt sich die Frage, wo genau nun hier eine `for`-Schleife zum Einsatz kommt. Sie haben es aber wahrscheinlich schon geahnt: Der String `s` kann natürlich in eine Liste von Zeichen zerlegt werden; diese Liste können wir durchlaufen und so der Reihe nach jedes einzelne Zeichen des Strings bearbeiten. Dabei verwenden wir die Funktionen `ord(zeichen)` und `chr(code)`, die den numerischen Code für ein bestimmtes Zeichen bzw. das Zeichen zu einem numerischen Code zurückgeben.

Die Funktion `rot13()` könnte dann so aussehen:

## 25.1 Abgezählte Schleifen (for)

```
def rot13(s: str, decodieren: bool,
          verschiebung: int = 13) -> str:
    s = list(s)
    res = list()
    for c in s:
        if decodieren == True:
            res.append(chr(ord(c) - verschiebung))
        else:
            res.append(chr(ord(c) + verschiebung))
    return ''.join(res)
```

Die **for**-Schleife durchläuft den String **s**, den wir zuvor in eine Liste einzelner Zeichen umgewandelt haben. Die Laufvariable **c** repräsentiert also in jedem Schleifendurchlauf jeweils ein anderes Zeichen des Strings **s**.

Mit **append()** bauen wir nach und nach eine neue Liste zusammen, die den codierten bzw. decodierten String enthält. Am Ende schließlich werden die Zeichen der Liste mit **join()** zu einer Zeichenkette zusammengebunden, an einen leeren String (dessen **join()**-Methode wir dazu verwenden) angehängt und als Rückgabewert retourniert.

Die Funktion können wir nun zum Beispiel mit **print(rot13('HALLO WELT', False))** aufrufen, was **UNYYl-dRYa** in der Konsole ausgibt. Mit **print(rot13(' UNYYl-dRYa, True))** können wir den verschlüsselten Text wieder dechiffrieren. Wenn sich am Ende wieder ergibt '**HALLO WELT**', wissen Sie, dass Ihre Funktion richtig arbeitet!

Zum Abschluss dieses Abschnitts noch ein anderes Beispiel. Dieses Mal wollen wir ein Programm schreiben, das den Benutzer ein Verzeichnis eingeben lässt und dann die darin enthaltenen Unterverzeichnisse und Dateien auflistet. Dazu verwenden wir Funktionen aus dem Python-Standard-Package **os**. Dieses Package stellt Möglichkeiten zum Zugriff auf Betriebssystem-Funktionalitäten und -Ressourcen bereit. Dazu gehören auch Funktionen, die es erlauben, mit dem Dateisystem zu arbeiten. Das Schöne an **os** ist dabei, dass es plattformübergreifend konzipiert ist; ein Programm, das **os**-Funktionen verwendet und auf Ihrem Windows-Computer funktioniert, wird daher ebenso auf einem Mac oder Linux-Computer arbeiten. Um die Spezifika dieser unterschiedlichen Betriebssysteme müssen Sie sich dabei nicht kümmern. Das Package **os** stellt eine einheitliche, plattformübergreifende Programmier-Schnittstelle für alle diese Systeme bereit, die von den Spezifika der Systeme abstrahiert.

Der Code unseres Programms könnte nun so aussehen:

```
from os import listdir, sep
from os.path import isdir, isfile, getsize

verzeichnis = input('Bitte geben Sie ein Verzeichnis ein: ')

if isdir(verzeichnis):

    if verzeichnis[-1] != sep:
```

```

verzeichnis = verzeichnis + sep

print('\nInhalt von:', verzeichnis)
inhalt = listdir(verzeichnis)

print('\n----- Verzeichnisse:')

for elem in inhalt:
    if isdir(verzeichnis + elem):
        print(elem)

print('\n----- Dateien:')

for elem in inhalt:
    if isfile(verzeichnis + elem):
        print(verzeichnis + elem, '\tGröße: ',
              getsize(verzeichnis + elem), sep = '')

else:
    print(verzeichnis, ' ist kein gültiges Verzeichnis!')

```

An den Importen erkennen Sie, welche Elemente des Moduls **os** wir nutzen wollen. Die Funktion **listdir(verzeichnis)** gibt den Inhalt des angegebenen Verzeichnisses als Liste zurück. Die Konstante **sep** repräsentiert den Separator, der auf dem jeweiligen System in Pfadangaben verwendet wird, bei Windows also der Backslash (\), bei Linux der Forward slash (/).

Mit Hilfe der Funktionen **isdir(verzeichnis)** und **isfile(datei)** aus dem **path**-Modul schließlich können wir ermitteln, ob ein bestimmter Verzeichnis- oder Dateipfad tatsächlich zu einem gültigen Verzeichnis oder einer gültigen Datei führt. Beide Funktionen geben einen **bool**-Wert zurück. Die letzte importierte Funktion, **getsize(datei)**, gibt die Größe der angegebenen Datei in Bytes zurück. Bitte beachten Sie: Wann immer Sie eine Datei angeben, muss jeweils der gesamte Pfad mitgenannt werden, anderenfalls wird die Datei nicht gefunden.

Im Code werden zwei **for**-Schleifen verwendet, die Schritt für Schritt den mit Hilfe von **listdir()** ermittelten Verzeichnisinhalt durchgehen. Die erste Schleife verarbeitet nur die Verzeichnisse, sodass diese im Programm-Output dann oben stehen, die zweite nur Dateien, die im Output dann unterhalb der Verzeichnisliste erscheinen.

### ? 25.1 [10 min]

Schreiben Sie eine Funktion, die alle Vokale aus einem String entfernt.

### ? 25.2 [10 min]

Schreiben Sie eine Funktion, die einen String vollständig umkehrt und dabei in Großbuchstaben umwandelt.

### 25.1.2 Verschachtelte for-Schleifen

for-Schleifen können auch ineinander verschachtelt werden. Betrachten Sie dazu das folgende Beispiel:

```
buchstaben = ['A', 'B', 'C', 'D', 'E', 'F', 'G']
zahlen = [1, 2, 3, 4, 5, 6, 7, 8, 8, 10]

for b in buchstaben:
    for z in zahlen:
        print(b + str(z), ' ', end = '')
    print('\n')
```

Hier sehen wir zwei **for**-Schleifen: Eine *äußere* Schleife, die die Liste der Buchstaben von A bis G durchläuft und eine *innere*, die sich die Zahlen von 1 bis 10 vornimmt. Dabei befindet sich die innere Schleife eingerückt im Code-Block der äußeren Schleife. Bei jedem Durchlauf der äußeren Schleife wird das innere **for**-Konstrukt vollständig durchlaufen, also die Zahlen von 1 bis 10 abgearbeitet. Danach geht die äußere Schleife in ihren nächsten Durchlauf, das heißt, in die Bearbeitung des nächsten Buchstabens.

Was wird dieses kleine Programm wohl ausgeben?

Das hier sehen wir, wenn wir das Programm starten:

```
A1  A2  A3  A4  A5  A6  A7  A8  A8  A10
B1  B2  B3  B4  B5  B6  B7  B8  B8  B10
C1  C2  C3  C4  C5  C6  C7  C8  C8  C10
D1  D2  D3  D4  D5  D6  D7  D8  D8  D10
E1  E2  E3  E4  E5  E6  E7  E8  E8  E10
F1  F2  F3  F4  F5  F6  F7  F8  F8  F10
G1  G2  G3  G4  G5  G6  G7  G8  G8  G10
```

Das Programm schreibt also für den aktuell in der äußeren Schleife durchlaufenen Buchstaben, den Buchstaben mit Index **b** in der Liste **buchstaben**, die Kombination aus diesem Buchstaben und *allen* Zahlen von 1 bis 10, die in der inneren Schleife durchlaufen werden, hintereinander; beachten Sie die Argumentzuweisung **end = ''** in der Funktion **print()**, die dazu führt, dass nach jeder Ausgabe *kein* Zeilenumbruch erfolgt. Erst nachdem die innere Schleife vollständig durchlaufen worden ist, wird ein Zeilenumbruch (**\n**) ausgegeben und die äußere Schleife geht in den nächsten Durchlauf.

Der Code-Block der äußeren Schleife besteht also aus der kompletten inneren **for**-Schleife und der **print()**-Anweisung, die den Zeilenumbruch erzeugt.

Natürlich können auf diese Weise nicht nur zwei, sondern auch noch mehr **for**-Schleifen ineinander verschachtelt werden. Aufpassen muss man allerdings mit der Laufzeit des Programms, denn jede weitere **for**-Schleife multipliziert grundsätzlich die Zahl der Schleifendurchläufe mit ihrer eignen Durchlaufzahl.

### 25.1.3 Listenkomprehensionsausdrücke

25

Eine Besonderheit in Python sind die sogenannten *Listenkomprensionsausdrücke*. Was zunächst furcheinflößend klingen mag ist nichts anderes als eine **for**-Schleife, die eine Liste erzeugt. Für solche **for**-Schleifen gibt es eine kompakte, elegante Schreibweise, eben den Listenkomprehensionsausdruck.

Im letzten Abschnitt haben wir in einem Beispiel mit Hilfe der Funktion **listdir()** aus dem Modul **os** den Inhalt eines Verzeichnisses als Liste ermittelt und dann in einer **for**-Schleife jedes Element mit Hilfe der Funktion **isfile()** daraufhin geprüft, ob es sich um eine Datei (und nicht etwa um ein Verzeichnis) handelt. Nur wenn es sich tatsächlich um eine Datei handelte, haben wir das Element auf dem Bildschirm ausgeben. Dasselbe haben wir in einer weiteren **for**-Schleife mit Verzeichnissen gemacht. Auf diese Weise konnte wir Verzeichnisse und Dateien in der Ausgabe sauber voneinander trennen. Einfacher wäre es nun gewesen, gleich von Anfang an zwei Listen zu erzeugen, eine mit den Dateien, eine mit den Verzeichnissen und diese dann mit zwei simplen **for**-Schleifen anzeigen zu lassen.

Das ist eine Aufgabe, die wie geschaffen für die Verwendung von Listenkomprehensionsausdrücken ist. Betrachten Sie folgende Formulierung:

```
inhalt = listdir(verzeichnis)
dateien = [f for f in inhalt if isfile(verzeichnis + f)]
verzeichnisse = [f for f in inhalt if isdir(verzeichnis + f)]
```

Zunächst besorgen wir uns mit **listdir()** den Inhalt unseres ausgewählten Verzeichnisses. Dann folgen zwei Listenkomprehensionsausdrücke, mit denen wir aus der Liste **inhalt** jeweils die Dateien und Verzeichnisse herausfiltern.

Der Listenkomprehensionsausdruck gibt etwas zurück, nämlich die Elemente, die in der Ergebnisliste stehen sollen. Das was zurückgegeben wird, steht zu Beginn in den eckigen Klammern, die die Konstruktion einer Liste signalisieren; in unserem Beispiel ist das **f**. Jetzt folgt eine **for**-Schleife, nämlich **for f in inhalt**. Damit wird jedes Element der Liste **inhalt** durchlaufen. Zurückgegeben werden dabei allerdings nur die Elemente, die die mit **if** definierte Bedingung erfüllen, in unserem ersten Beispiel also all' jene Elemente, für die die Funktion **isfile()** den Wert **True** zurückgibt, die also Dateien sind. Der Listenkomprehensionsausdruck geht somit alle Elemente von **inhalt**, der Liste mit dem Verzeichnisinhalt, durch und gibt diejenigen zurück, bei denen es sich um eine Datei handelt.

## 25.2 Bedingte Schleifen (while)

Die **if**-Bedingung ist dabei optional; wollten wir zum Beispiel eine Liste aller Elemente in dem Verzeichnis erhalten, allerdings in Großbuchstaben, könnten wir auf die Bedingung verzichten, würden aber das Ergebnis vor der Rückgabe noch modifizieren, indem wir die Funktion **upper()** aufrufen:

```
inhalt_gross = [f.upper() for f in inhalt]
```

An diesem Beispiel sehen Sie zudem, dass der Ausdruck, der durch die **for**-Schleife des Listencomprehensionsausdrucks mit immer neuen Elementen „gefüttert“ wird, durchaus auch ein komplexerer Ausdruck sein kann, als einfach nur das von der **for**-Schleife gelieferte Objekt selbst; in diesem Fall eben ein Methodenaufruf für dieses Objekt.

Ein anderes Beispiel ist das folgende, bei dem wir den Wert der durch die **for**-Schleife gelieferten Variable quadrieren, wenn die Variable selbst glatt durch zwei teilbar, also eine gerade Zahl ist (dazu verwenden wir den Modulo-Operator `%`, der den Rest einer Division zurückgibt)

```
quadrate = [x*x for x in range(1,11) if x % 2 == 0]
```

Hier ist der Ausdruck, der durch die **for**-Schleife mit Werten beschickt wird, eben `x*x`.

Damit haben Listencomprehensionsausdrücke also allgemein die folgende Form (wobei, wir bereits gesehen haben, die Bedingung optional ist):

```
liste = [ausdruck for laufvariable in iterierbares_objekt if  
bedingung]
```

### 25.3 [5 min]

Schreiben Sie einen Listencomprehensionsausdruck, der die Großbuchstaben von A bis Z als Liste zusammenfasst. Tipp: Arbeiten Sie mit Funktionen den **ord(zeichen)** und **chr(code)**, die wir auch in ► Abschn. 25.1.1 verwendet haben.

Wie würde eine äquivalente **for**-Schleife aussehen, die die Buchstabenliste ohne Listencomprehensionsausdruck erzeugt?

## 25.2 Bedingte Schleifen (while)

Python kennt mit der **while**-Schleife eine bedingte Schleifenkonstruktion, also eine Schleife, die so lange läuft, wie eine Laufbedingung erfüllt ist. Pythons **while**-Schleife ist dabei *kopfgesteuert*, das bedeutet, die Bedingung wird *vor* jedem Durchlauf geprüft. Ist die Bedingung auch vor dem ersten potentiellen Durchlauf nicht erfüllt, so wird die Schleife überhaupt nicht durchlaufen.

Mit einer **while**-Schleife kann natürlich dasselbe Verhalten simuliert werden, wie mit einer **for**-Schleife. Im folgenden Beispiel gibt eine **while**-Schleife die Zahlen 1 bis 10 auf dem Bildschirm aus:

```
i = 1
while i <= 10:
    print(i)
    i = i + 1
```

25

Wir initialisieren eine Variable **i**, die wir als Laufvariable verwenden, mit dem Wert 1. Die Schleife prüft vor jedem Durchlauf, ob die Bedingung, dass **i** kleiner oder gleich 10 ist, nach wie vor erfüllt ist. Ist das der Fall, wird der hinter dem Doppelpunkt beginnende, wie üblich eingerückte Code-Block durchlaufen. Danach springt die Schleife zurück in den Schleifenkopf, wo wiederum die Laufbedingung geprüft wird.

Der häufigste Fehler bei einer solchen Konstruktion ist es, zu vergessen, die Laufvariable auch hochzuzählen (wie es dem Autor natürlich im ersten Versuch auch passiert ist), denn anders als bei der **for**-Schleife übernimmt diese Aufgabe die **while**-Schleife nicht selbst. Die **while**-Schleife läuft stur so lange weiter, wie die Laufbedingung erfüllt ist. Um den Rest müssen wir uns selbst kümmern. Ist die Laufbedingung *immer* erfüllt, haben wir eine Unendlich-Schleife hergestellt, was in aller Regel unerwünscht ist.

Ihre wahre Stärke spielt die **while**-Schleife aber dann aus, wenn – anders als im Beispiel oben – die Zahl der Schleifendurchläufe im Vorhinein noch nicht festgelegt werden kann, zum Beispiel, weil das Durchlaufen der Schleife von einem Ereignis abhängen soll, das sich erst aus der Interaktion mit dem Benutzer ergibt.

Betrachten Sie das folgende Beispiel. Es listet alle in einem vom Benutzer ausgewählten Verzeichnis enthaltenen Dateien auf und erlaubt es ihm, eine dieser Dateien zu öffnen bzw. zu starten. Letzteres bewerkstelligen wir mit der Funktion **startfile(datei)** aus dem Modul **os**, das wir bereits in den vorangegangenen Abschnitten verwendet haben. Der Benutzer kann dabei die zu öffnende/startende Datei über eine Nummer angeben, mit der die Datei gelistet wird. Auch bieten wir dem Benutzer die Möglichkeit, das Verzeichnis zu wechseln oder aber das Programm komplett zu beenden.

Der Kern des Programms besteht aus einer **while**-Schleife, mit der wir den Benutzer immer wieder nach seinen Aktionswünschen fragen (Variable **wahl**), die er über einen Buchstaben ('v' für Verzeichnis wechseln, 's' für Datei öffnen/starten, 'x' für Beenden) eingeben kann. Die Bedingung der **while**-Schleife lautet, dass die Auswahl des Benutzers von 'x' verschieden ist. So lange das der Fall ist, läuft das Programm weiter, ist das nicht mehr der Fall, endet das Programm.

Hier der vollständige Programmcode:

## 25.2 Bedingte Schleifen (while)

```
from os import listdir, startfile
from os.path import isfile, isdir

wahl = ''
verz_vorhanden = False

while wahl != 'x':
    wahl = input(
        'Verzeichnis lesen (v), Datei starten (s), Beenden (x)? ')
    if wahl != 'x':
        if wahl == 'v':
            verzeichnis = input(
                'Bitte geben Sie ein Verzeichnis ein: ')
            if isdir(verzeichnis):
                dateien = listdir(verzeichnis)
                nummer = 0
                for dat in dateien:
                    if isfile(verzeichnis + dat):
                        nummer = nummer + 1
                        print(nummer, ': ', dat, sep = '')
                verz_vorhanden = True
            else:
                print('''
                    '' ist kein gültiges Verzeichnis! Bitte \
                    'erneut versuchen.', sep = '')
        elif wahl == 's':
            if verz_vorhanden == True:
                nummer = int(input(
                    'Bitte Dateinummer angeben: '))
                if nummer >= 1 and nummer <= dateien.__len__():
                    startfile(
                        verzeichnis + dateien[nummer - 1])
                else:
                    print('Kein Verzeichnis geladen!')
            else:
                print('Eingabe ', wahl,
                    ' nicht zulässig! Bitte erneut versuchen.',
                    sep = '')
        else:
```

Wenn Sie das Programm ausführen, beachten Sie bitte, dass die Verzeichnisangabe immer mit einem Pfad-Separator, also dem Backslash (\) auf Windows-Systemen bzw. dem Forwardslash (/) auf Mac- und Linux-Systemen enden muss, sonst wird der im Programm zusammengesetzte Dateipfad ungültig.

Innerhalb der **while**-Schleife findet zunächst die Abfrage der vom Benutzer gewünschten Aktion statt. Wenn diese Wahl von 'x' verschieden ist, werden die Aktionen 's' (starten) und 'v' (Verzeichnis wechseln) abgeprüft und verarbeitet. Andernfalls, wenn also der Benutzer 'x' eingegeben hat, passiert im Inneren der

**while**-Schleife gar nichts mehr, die Schleife springt also zurück auf die Prüfung der Laufbedingung im Schleifenkopf und stellt fest, dass die Laufbedingung **wahl != 'x'** nicht mehr erfüllt ist. Daher wird die Programmverarbeitung hinter dem Code-Block der **while**-Schleife fortgesetzt. In unserem Beispiel ist das Programm damit beendet.

Wie Sie an diesem Programmbeispiel sehen können, können wir mit **while**-Schleifen letztlich eine Ereignissteuerung herstellen, indem wir auf eine Benutzereingabe warten, diese, wenn sie kommt, verarbeiten, und dann wiederum auf die nächste Benutzereingabe warten; so lange, bis der Benutzer das Programm beenden möchte. Die **mainloop()**-Funktion in den **tkinter**-Programmen aus ► Abschn. 22.2 macht letztlich nichts anderes.

Ereignissteuerung ist also keineswegs ein Hexenwerk, sondern lässt sich mit klassischen Kontrollstrukturen wie **while** und **if** realisieren.

25

### 25.4 [20 min]

Schreiben Sie ein Programm, dass die Funktion **temperatur\_umrechnen()** aus ► Abschn. 24.1.2 verwendet, um Temperaturen zwischen Kelvin und Grad Celsius (und umgekehrt) umzurechnen. Der Benutzer des Programms soll dabei so lange eine „Menü“ aus den Aktionen 'k' (Umrechnung von Grad Celsius nach Kelvin), 'c' (Umrechnung von Kelvin nach Grad Celsius) und 'x' (Programm beenden) angeboten bekommen, bis er das letzgenannte Befehlskürzel eingibt.

## 25.3 Schleifen vorzeitig verlassen und neustarten

### ■ Schleifen per Anweisung beenden

Mit den Anweisungen **break** und **continue** bietet Python Möglichkeiten, den Ablauf von **for**- und **while**-Schleifen noch genauer zu steuern.

Der Aufruf der **break**-Anweisung führt dazu, dass die Schleife abgebrochen und die Programmausführung *hinter* dem Schleifen-Code-Block fortgesetzt wird.

Betrachten Sie dazu folgendes Beispiel:

```
while True:
    x = input('Ihre Eingabe: ')
    if x == 'x':
        break
    print('Ihre Eingabe war: ', x)
print('Schleife beendet.')
```

Diese **while**-Schleife besitzt eine Bedingung (**True**), die per definitionem immer wahr ist. Sie läuft so lange, bis der Benutzer 'x' eingibt. Wenn das geschieht, wird die Schleife mit der Anweisung **break** an Ort und Stelle beendet. Der Aufruf von **print()**, der sich im Schleifen-Code-Block hinter dem **break** befindet, wird nicht mehr ausgeführt.

Die folgende Funktion nutzt **break**, um eine **for**-Schleife vorzeitig zu verlassen; das kann zum Beispiel dann nützlich sein, wenn es darum geht, festzustellen, ob

### 25.3 Schleifen vorzeitig verlassen und neustarten

irgendetwas existiert. Sobald ein Exemplar von dem Gesuchten gefunden worden ist, machen weitere Schleifendurchläufe keinen Sinn mehr, denn sie können das Ergebnis der Suche ja nicht mehr ändern. In dieser Situation ist es geschickt, die Schleife zu verlassen, um Rechenzeit zu sparen und die Programmausführung zu beschleunigen.

Im folgenden Beispiel sehen wir eine Funktion, die die in einem Verzeichnis vorhandenen Dateien daraufhin überprüft, ob darunter eine Excel-Datei mit einer Größe von mehr als 1 MB (= 1.000.000 Bytes) vorhanden ist.

```
from os import listdir
from os.path import isfile, getsize

def excel_gross(verzeichnis: str) -> bool:
    dateien = [f for f in listdir(verzeichnis) if
               isfile(verzeichnis + f)
               and (f[-5:].lower() == '.xlsx'
                     or f[-4:].lower() == '.xls')]

    res = False
    for dat in dateien:
        if getsize(verzeichnis + dat) > 1000000:
            res = True
            break
    return res
```

Dazu wird zunächst eine Liste der Excel-Dateien in dem zu durchsuchenden Verzeichnis, das der Funktion als Argument **verzeichnis** übergeben wird, erzeugt, und zwar mit Hilfe eines Listenkomprehensionsausdrucks. Darin wird geprüft, welche Dateinamen eine auf Excel-Dateien hindeutende Endung besitzen (wir suchen hier der Einfachheit halber nur nach **.xls**- und **.xlsx**-Dateien und ignorieren andere mögliche Dateinamenserweiterungen von Excel-Dateien). Der Listenkomprehensionsausdruck prüft also die Bedingung, dass das jeweilige Verzeichnis-Inhaltselement eine Datei ist und diese zugleich auf **.xls** oder **.xlsx** endet.

Danach durchläuft eine **for**-Schleife die durch den Listenkomprehensionsausdruck erzeugte Liste und prüft mit der Funktion **getsize(datei)**, ob irgendeine dieser Dateien größer ist als 1 MB. Sobald eine solche Datei gefunden wurde, wird die Schleife mit **break** verlassen. Zuvor wird der Rückgabewert **res** der Funktion noch von seinem Vorwert **False**, auf den er vor Beginn der Schleife initialisiert wurde, auf **True** umgeschaltet. Hinter der Schleife wird dann die **return**-Anweisung ausgeführt, die diesen Wert auch tatsächlich an den Aufrufer zurückgibt.

Natürlich hätte man die Bedingung, dass die Dateigröße 1 MB überschreiten muss, auch gleich in den Listenkomprehensionsausdruck mit einbauen können und dann auf die **for**-Schleife gänzlich verzichten können (Wie? Probieren Sie es aus!). Das Beispiel zeigt aber, wie man eine Suchschleife effizient gestalten kann, indem man sie abbricht, sobald das Gesuchte gefunden worden ist. Denselben Effekt hätte man auch mit einer **while**-Schleife erreichen können, die dazu folgendermaßen aussehen müsste:

```
i = 0
res = False
while i <= dateien.length() and res == False:
    if getsize(verzeichnis + dateien[i]) > 1000000:
        res = True
    i = i + 1
```

Die Formulierung mit der **for**-Schleife mutet da schon etwas intuitiver an (und vermeidet das Risiko, zu vergessen, die Laufvariable **i** selbst hochzuzählen).

### ■ Schleifen per Anweisung mit dem nächsten Durchlauf fortsetzen

25

Während **break** die Schleife, innerhalb derer es aufgerufen wird, vollständig verlässt, führt die Anweisung **continue** dazu, dass die Schleifenausführung einfach mit dem nächsten Schleifendurchlauf fortgesetzt wird. Das wird deutlich am folgenden Beispiel, indem wir zwar in einer **for**-Schleife die Zahlen von 1 bis 10 durchlaufen, jedoch die im Schleifen-Code-Block befindliche **print()**-Anweisung für alle geraden Zahlen „abklemmen“, indem wir die Schleife vorher mit **continue** in ihren nächsten Durchlauf schicken:

```
for i in range(1,10):
    if i % 2 == 0:
        continue
    print(i)
```

Dabei nutzen wir den *Modulo-Operator %*, der den Divisionsrest zurückgibt; ist dieser gleich 0, handelt es sich um eine gerade Zahl und die Schleife wird mit dem nächsten Durchlauf fortgesetzt, der Aufruf von **print()** am Ende des Schleifen-Code-Blocks wird dann gar nicht mehr erreicht. Er wird nur bei ungeraden Zahlen erreicht, sodass auch nur diese auf dem Bildschirm ausgegeben werden.

## 25.4 Zusammenfassung

In diesem Kapitel haben wir uns mit **for**- und **while**-Schleifen sowie den auf den **for**-Schleifen basierenden Listenkomprehensionsausdrücken beschäftigt. Außerdem haben wir gesehen, wie Schleifen oder der aktuelle Schleifendurchlauf vorzeitig verlassen werden können.

Folgende Punkte sollten Sie aus diesem Kapitel unbedingt mitnehmen:

- Python kennt als abgezählte Schleife das **for**-Konstrukt, das die Elemente eines iterierbaren Objekts durchläuft; iterierbar ist ein Objekt, wenn Python sich von einem Element/Bestandteil des Objekts zum nächsten „hangeln“ kann (wie das etwa bei Listen, Tupeln, Sets oder Dictionaries der Fall ist).
- Der Grundaufbau der **for**-Schleife ist: **for laufvariable in iterierbares\_objekt: code\_block**; die **laufvariable** repräsentiert dabei bei jedem Schleifendurchlauf ein anderes Element, das Bestandteil von **iterierbares\_objekt** ist.

## 25.5 Lösungen zu den Aufgaben

- Änderungen an der Laufvariable ändern *nicht* das durch sie repräsentierte Element des durchlaufenen Objekts.
- **for**-Schleifen können beliebig ineinander verschachtelt werden.
- Eine besondere Art der **for**-Schleife ist der Listenkomprehensionsausdruck; er erzeugt mit einer sehr kompakten Syntax eine Liste; die allgemeine Form lautet: **liste = [ausdruck for laufvariable in iterierbares \_objekt if bedingung]**, also zum Beispiel die Liste der quadrierten geraden Zahlen von 1 bis 10: **quadrate = [x\*x for x in range(1,11) if x % 2 == 0]**, wobei **%** der Modulo-Operator ist, der den Rest einer Division zurückgibt.
- Bedingte Schleifen werden mit **while** konstruiert; ihre allgemeine Form lautet: **while bedingung: code\_block**.
- Sowohl **for**- als auch **while**-Schleifen können mit der **break**-Anweisung vorzeitig verlassen werden.
- Die **continue**-Anweisung bricht den aktuellen Schleifendurchlauf ab und setzt die Schleife mit dem nächsten Durchlauf fort.

## 25.5 Lösungen zu den Aufgaben

---

### ■ Aufgabe 25.1

Eine Funktion, die alle Vokale aus einem String entfernt, könnte so aussehen:

```
def vokale_entfernen(s: str) -> str:
    res = list()
    s = list(s)
    for i in range(0,s.__len__(),1):
        if s[i] not in ('a','e','i','o','u'):
            res.append(s[i])
    return ''.join(res)
```

Test, ob die Funktion den gewünschten Effekt hat:

```
print(vokale_entfernen('Hallo Welt!'))
```

### ■ Aufgabe 25.2

Zur Umkehrung der Zeichen im String wird der in eine Liste einzelner Zeichen umgewandelte String in einer **for**-Schleife *von hinten nach vorne* durchlaufen. Die Schrittweite, das dritte Argument der Funktion **range()**, ist daher **-1**. Weil **range()** bis *vor* die angegebene Grenze läuft, muss die zweite Bereichsgrenze ebenfalls **-1** sein, damit der letzte zurückgegebene Indexwert **0** ist und so den Zugriff auf das erste Zeichen von **s** erlaubt.

```
def umkehren_gross(s: str) -> str:
    s = list(s)
    res = list()

    for i in range(s.__len__() - 1, -1, -1):
        res.append(s[i].capitalize())
    return str(''.join(res))
```

Die Funktion könnte dann zum Beispiel so aufgerufen werden:

```
print(umkehren_gross('Hallo Welt!'))
```

25

### ■ Aufgabe 25.3

Ein Listenkomprehensionsausdruck, der die Buchstaben von **A** bis **Z** als Liste zusammenfasst, könnte so aussehen:

```
gross_buchstaben = [chr(b) for b in range(ord('A'), ord('Z')+1)]
```

Wir durchlaufen also mit der Variable **b** die Zeichencodes beginnend mit dem Code des Buchstabens A, bis hin zum Code des Buchstabens Z (Achtung: **range()** liefert einen Wertebereich, der die angegebene rechte/obere Grenze nicht mehr enthält, daher **+1**). Diese Codes wandeln wir dann mit Hilfe der Funktion **chr()** wieder in Buchstaben um.

Eine äquivalente **for**-Schleife würde so aussehen:

```
gross_buchstaben = list()
for b in range(ord('A'), ord('Z')+1):
    gross_buchstaben.append(chr(b))
```

### ■ Aufgabe 25.4

Das Programm könnte zum Beispiel so aussehen:

## 25.5· Lösungen zu den Aufgaben

```
wahl = ''  
  
while wahl != 'x':  
    wahl = input(  
        'Umrechnung Grad Celsius zu Kelvin (k), Kelvin zu \n  
        Grad Celsius(c), Beenden(x) ? ')  
    if wahl == 'k':  
        temp = input('Bitte geben Sie eine Temperatur in Grad \n  
        Celsius ein: ')  
        orig_einheit = 'Grad Celsius'  
        ziel_einheit = 'Kelvin'  
        nach_celsius = False  
    elif wahl == 'c':  
        temp = input(  
            'Bitte geben Sie eine Temperatur in Kelvin ein: ')  
        orig_einheit = 'Kelvin'  
        ziel_einheit = 'Grad Celsius'  
        nach_celsius = True  
    elif wahl == 'x':  
        pass  
    else:  
        print('"'', wahl, '"' ist keine zulässige Eingabe. \n  
        Bitte erneut versuchen.',sep = '')  
  
    if wahl == 'k' or wahl == 'c':  
        erg = temperatur_umrechnen(float(temp),  
                                    nach_celsius)  
    if erg != -1000:  
        print(temp, orig_einheit, 'sind', erg, ziel_einheit)
```

Beachten Sie, dass das Programm auf die Funktion **temperatur\_umrechnen()** aus ► Abschn. 24.1.2 zurückgreift. Diesen Code benötigen Sie also zur Ausführung des Programms ebenfalls.

Beachten Sie bitte weiterhin, dass die Variable **wahl**, die den Befehl des Benutzers aufnimmt, bereits vor der **while**-Schleife initialisiert werden muss. Andernfalls stößt die **while**-Schleife bei der Prüfung ihrer Laufbedingung auf eine nicht vorhandene Variable, was eine Fehlermeldung auslöst.



# Wie suche und behebe ich Fehler auf strukturierte Art und Weise

## Inhaltsverzeichnis

- 26.1 Fehlerbehandlung  
zur Laufzeit – 418**
  - 26.1.1 Fehler durch gezielte Prüfungen  
abfangen – 418
  - 26.1.2 Versuche...Fehler-Konstrukte  
(try...except) – 420
- 26.2 Fehlersuche und -beseitigung  
während der Entwicklung – 422**
  - 26.2.1 Haltepunkte – 423
  - 26.2.2 Anzeige des Variableninhalts und  
Verwendung von Watches – 425
  - 26.2.3 Schrittweise Ausführung – 426
- 26.3 Zusammenfassung – 426**
- 26.4 Lösungen zu den  
Aufgaben – 427**

## Übersicht

Fehler gehören zum Programmieralltag, eine unangenehme aber leider unumstößliche Wahrheit. Deshalb beschäftigen wir uns zum Abschluss unserer Tour durch die Grundlagen von Python damit, wie Fehler gefunden, analysiert und behandelt werden können.

In diesem Kapitel werden Sie lernen:

- wie Sie durch geeignete Überprüfungen in Ihrem Programmcode das Programm robuster machen, zum Beispiel gegenüber Fehleingaben des Benutzers
- wie Sie Fehler zur Laufzeit mit Hilfe sogenannter Ausnahmen abfangen und geordnet bearbeitet können
- wie Sie die Debugging-Werkzeuge von *PyCharm* nutzen können, um bereits während der Entwicklung Fehler zu finden und zu analysieren.

# 26

## 26.1 Fehlerbehandlung zur Laufzeit

### 26.1.1 Fehler durch gezielte Prüfungen abfangen

Wir haben in ► Kap. 16 Fehler *zur Laufzeit* definiert als Fehler, die sich erst aus Umständen der konkreten Programmausführung ergeben. Zwar gibt es dann Anwendungsszenarien, in denen der Code genau das angestrebte Ziel fehlerfrei erreicht; und genau diese Szenarien standen bei der Entwicklung im Vordergrund, anhand solcher Szenarien wurde das Programm getestet. Allerdings kann es während der Ausführung doch zu Situationen kommen, in denen das Programm *nicht* das tut, was es soll; insbesondere ist das dann der Fall, wenn die Programmausführung außer Kontrolle gerät und das Programm „abstürzt“, also mit einem Fehler abbricht und (ohne Neustart des Programms) nicht weiter verwendet werden kann.

Eine häufige Ursache solcher Fehler sind Probleme mit Typ-Umwandlungen von Variablen. Betrachten Sie dazu das folgende Beispiel:

```
from math import sqrt

x_str = input('Geben Sie eine Zahl ein, aus der die Quadratwurzel
              gezogen werden soll: ')
x_float = float(x_str)
print(sqrt(x_float))
```

Das Programm liest vom Benutzer eine Zahl ein und berechnet daraus die Quadratwurzel. Dazu wird die Funktion **sqrt()** aus dem Modul **math** verwendet.

Da die Funktion **input()** immer einen String zurückgibt, muss die Nutzereingabe zunächst mit einer normalen Typkonversion in eine **float**-Variable, also eine

Fließkommazahl, umgewandelt werden (wenn Ihnen das nicht mehr geläufig ist, blättern Sie noch einmal zurück zu ► Abschn. 21.5).

Wenn Sie das Programm nun starten und beispielsweise 25 eingeben, erhalten Sie als Ausgabe die Quadratwurzel, also 5. Das Programm arbeitet offenbar fehlerfrei, zumindest in dem Anwendungsszenario, das wir getestet haben. Was aber passt, wenn der Benutzer keine Zahl, sondern eine Zeichenkette (zum Beispiel 'hallo') eingibt? Wir erhalten sofort von Python eine unschöne Fehlermeldung angezeigt:

```
x_float = float(x_str)
ValueError: could not convert string to float: 'hallo'
```

Der Fehler lässt sich natürlich leicht abfangen, indem wir zunächst in einer **if**-Anweisung prüfen, ob die Eingabe des Benutzers tatsächlich eine Zahl ist. Unser Programm könnte dann zum Beispiel so aussehen:

```
from math import sqrt

x_str = input('Geben Sie eine ganze, positive Zahl ein, aus der
              die Quadratwurzel gezogen werden soll: ')
if x_str.isnumeric():
    x_float = float(x_str)
    print(sqrt(x_float))
else:
    print('Ihre Eingabe "', x_str, '" ist keine ganze, positive Zahl!', sep='')
```

Auf diese Weise „immunisieren“ wir das Programm gegen Fehleingaben des Benutzers. Das Programm wird dadurch „robuster“, es kann jetzt mit mehr Szenarien, die in der realen Welt auftreten könnten, umgehen, ohne abzustürzen.

Vielleicht haben Sie noch eine weitere mögliche Fehlerquelle ausgemacht: Die Eingabe negativer Zahlen. Diese Art der „Fehleingabe“ fangen wir hier scheinbar nicht ab. Tatsächlich tun wir das allerdings schon, denn **isnumeric()** prüft lediglich, ob die Zeichenkette ausschließlich Ziffern enthält. Dezimaltrennzeichen oder Minus-Zeichen, die im String enthalten sind, führen automatisch dazu, dass der String nicht als Zahl betrachtet wird. Damit würde unsere selbstgebaute Fehlermeldung ausgegeben werden, sodass die Funktion **sqrt()** niemals mit einer negativen Zahl aufgerufen werden kann. Irritierenderweise besitzt Python von Haus aus keine Funktion, um festzustellen, ob ein String eine wie auch immer geartete Zahl enthält, also eine Zahl, die ggf. Vorzeichen, Dezimaltrennzeichen und Exponenten beinhaltet. Deshalb haben wir es uns hier einfach gemacht und lediglich die Eingabe einer positiven, ganzen Zahl zugelassen.

### 26.1.2 Versuche...Fehler-Konstrukte (try...except)

Nicht alle Probleme, die zur Laufzeit auftreten können, lassen sich so explizit beschreiben, dass wir sie mit einer Bedingung wie im obigen Beispiel abfangen können. Beispielsweise kann es beim Öffnen von Dateien zum Schreiben zahlreiche Ursachen dafür geben, dass das Öffnen misslingt. So kann die Datei etwa schreibgeschützt sein, wenn sie bereits vorhanden ist, oder der Datenträger schreibgeschützt sein, oder aber der verbleibende Platz auf dem Datenträger nicht ausreichen, um die Datei zu wie gewünscht zu schreiben.

Nun ist es natürlich gut, wenn man möglichst viele dieser potentiellen Fehlerquellen abfängt, um dem Benutzer die Ursache des Fehlers präzise mitteilen zu können und ihn so in die Lage zu versetzen, das Problem zu beheben, also zum Beispiel den Schreibschutz aufzuheben oder ausreichend Platz auf dem Datenträger freizuräumen. Tatsächlich werden aber, selbst wenn man einige mögliche Fehlerursachen durch gezielte Prüfungen ausschaltet, immer noch andere potentielle Fehlerquellen übrigbleiben, die man übersieht oder kaum vorhersehen kann, insbesondere bei so komplizierten Operationen wie Arbeiten mit dem Dateisystem.

Deshalb gibt es in Python noch einen anderen Weg, mit Fehlern umzugehen, nämlich mit einem *Versuche...Fehler*-Konstrukt, wie wir es in ► Abschn. 16.2 kennengelernt haben. Dieses Konstrukt besteht in Python aus den Schlüsselwörtern **try**, **except**, **else** und **finally**, wobei letztere beide optional ist.

Damit könnte unser Programm von oben so aussehen:

```
x_str = input('Geben Sie eine Zahl ein, aus der die Quadratwurzel  
gezogen werden soll: ')  
try:  
    x_float = float(x_str)  
    print(sqrt(x_float))  
except:  
    print('Ihre Eingabe ''', x_str, ''' ist keine Zahl!', sep='')
```

Hinter dem Schlüsselwort **try** folgt ein Code-Block mit Anweisungen, deren Ausführung „versucht“ wird. Gelingt diese Ausführung nicht fehlerfrei, wird der Code-Block hinter dem Schlüsselwort **except** ausgeführt. Läuft aber alles wie geplant, wird der **except**-Code-Block übersprungen und die Ausführung des Programms erst dahinter fortgesetzt.

Hier können wir nun auch gebrochene Zahlen mit Dezimaltrennzeichen eingeben, ebenso übrigens wie negative Zahlen. Geben Sie einmal eine negative Zahl ein! Was geschieht? Sie erhalten die Fehlermeldung **'Ihre Eingabe ist keine Zahl!'**. Das stimmt natürlich nicht; das Problem liegt tatsächlich an anderer Stelle, nämlich beim Ziehen der Quadratwurzel. Das Programm bricht mit einem Fehler ab, weil Sie versuchen, die Quadratwurzel einer negativen Zahl zu ziehen. Nun fängt aber das **try...except**-Konstrukt *alle* Fehler ab, die auftreten könnten, also auch den, der durch das negative Argument von **sqrt()** entsteht.

Wir haben nun zwei Möglichkeiten: Entweder, wir formulieren die Fehlermeldung allgemeiner („Es ist ein Fehler aufgetreten“), was für den Anwender natürlich nur wenig hilfreich ist, oder wir versuchen, die Fehlerursachen stärker zu differenzieren. Das ist über *Ausnahmeklassen* möglich. In Python wird, wie in vielen anderen Programmiersprachen auch, von Fehlern als „Ausnahmen“ gesprochen, also als Situationen, in denen das Programm sich ausnahmsweise nicht so verhält, wie es eigentlich sollte. Eine Ausnahmeklasse deckt nun Fehler (also Ausnahmen) einer bestimmten Art ab. Die Ausnahmeklasse **ZeroDivisionError** beispielsweise deckt die Ausnahme ab, dass bei einer Division der Divisor gleich 0 ist.

Die Ausnahmeklasse eines Fehlers wird übrigens in den Python-Fehlermeldungen auch angezeigt. So können Sie ermitteln, mit welcher Ausnahmeklasse Sie arbeiten müssen.

Das **except**-Schlüsselwort erlaubt es uns, bestimmte Klassen von Ausnahmen explizit abzufangen.

In unserem Beispiel hilft uns das allerdings nur beschränkt weiter, denn sowohl der Versuch, einen String, der keine Zahl beinhaltet, in eine **float**-Variable umzuwandeln, als auch die Berechnung der Quadratwurzel einer negativen Zahl führen zu einer Ausnahme derselben Ausnahmeklasse, nämlich **ValueError**. Nehmen wir nun aber an, wir erweitern unser Programm um eine Zeile, die den Kehrwert der eingegebenen Zahl bestimmt. Dann können wir die beiden möglichen Ausnahmeklassen durchaus differenzieren:

```
x_str = input('Geben Sie eine ganze Zahl ein, aus der Quadratwurzel  
und Kehrwert gezogen werden sollen: ')  
try :  
    x_float = float(x_str)  
    wurzel = sqrt(x_float)  
    kehrwert = 1/x_float  
except ValueError:  
    print('Ihre Eingabe "', x_str, '" ist keine oder eine  
negative Zahl!',  
          sep='')  
except ZeroDivisionError:  
    print('Der Kehrwert von 0 kann nicht berechnet werden!')  
except:  
    print('Sonstiger Fehler!')  
else:  
    print('Wurzel:', wurzel)  
    print('Kehrwert:', kehrwert)
```

In dieser Variation unseres Beispiels fangen wir Ausnahmen der Typen **ValueError** und **ZeroDivisionError** getrennt ab und können daher auch spezifischere Fehlermeldungen ausgeben.

Darüber hinaus erkennen Sie noch zwei weitere Unterschiede zum bisher verwendeten Aufbau des **try...except**-Konstrukts. Zum einen nämlich finden Sie neben den beiden **except**-Schlüsselwörtern, die jeweils spezifische Ausnahmeklassen

ansprechen, noch ein weiteres **except**, das ohne Ausnahmeklasse angegeben wird. Diese Konstruktion sorgt dafür, dass etwaige *andere* Fehler, die in keine der beiden explizit angegebenen Ausnahmeklassen fallen, ebenfalls behandelt werden. Zum anderen sehen Sie einen **else**-Block; die darin enthaltenen Anweisungen werden immer dann ausgeführt, wenn keine Ausnahme hervorgerufen worden ist („geworfen“, wie es im Programmierer-Jargon auch heißt).

Die **print**-Anweisungen hätte man natürlich ebenso gut in den **try**-Code-Block mit aufnehmen können. Auch dort würden sie nur ausgeführt werden, wenn keine Ausnahme auftritt. Allerdings bemüht man sich im Allgemeinen, den **try**-Block möglichst klein zu halten, um deutlich zu machen, was hier eigentlich die „riskante“ Aktion ist, von der man befürchtet, sie könne eine Ausnahme werfen.

Die Schlüsselwörter **try**, **except** und **else** müssen immer in dieser Reihenfolge verwendet werden. **else** ist allerdings optional und kann daher auch weggelassen werden.

## 26

Nicht betrachtet haben wir bisher das Schlüsselwort **finally**, das optional noch hinter dem **else**-Code-Block folgen kann und dessen Code-Block Anweisungen enthält, die *auf jeden Fall* immer ausgeführt werden, egal, ob eine Ausnahme geworfen wurde, oder nicht. Die Verwendung von **finally** ist besonders dann interessant, wenn zum Beispiel innerhalb eines **except**-Blocks die aktuelle Funktion mit **return**, oder die aktuelle Schleife mit **break** verlassen wird. Die Anweisungen hinter dem gesamten **try...except**-Konstrukt werden dann nicht mehr ausgeführt, sehr wohl aber diejenigen Anweisungen, die im **finally**-Code-Block enthalten sind. Erst nach deren Ausführung wird dann die Funktion oder Schleife verlassen.

### ? 26.1 [15 min]

Schreiben Sie ein Programm, das vom Benutzer eine Zahl, einen Operator (Addition, Subtraktion, Multiplikation, Division) und eine weitere Zahl einliest, dann die dadurch beschriebene Rechenoperation ausführt und das Ergebnis anzeigt. Das Programm soll möglichst robust gegen alle möglichen Fehler sein, die bei seiner Ausführung auftreten können.

### ! 26.2 [15 min]

Wie könnte eine alternative Formulierung des Programms aus Aufgabe 26.1 aussehen, die eine ähnliche Robustheit aufweist, aber auf die Verwendung des **try...except**-Konstrukt verzichtet?

## 26.2 Fehlersuche und -beseitigung während der Entwicklung

Zur Fehlersuche und -beseitigung während der Entwicklung stehen Ihnen eine Reihe von praktischen Werkzeugen zur Verfügung, wenn Sie mit einer integrierten Entwicklungsumgebung wie *PyCharm* arbeiten. Die wichtigsten davon sind *Haltepunkte (Breakpoints)*, *Variablen- und Ausdrucksbeobachtung (Watches)* und das *schrittweise Ausführen* des Programmcodes.

### 26.2.1 Haltepunkte

#### ■ Stop-Zeichen der Programmausführung

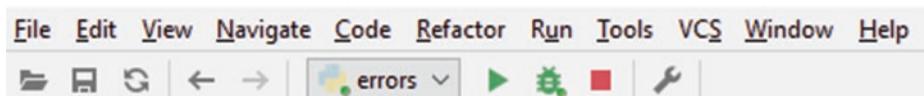
Ein Haltepunkt ist eine besondere Markierung einer Programmzeile, die dazu führt, dass das Programm nur bis zu dieser Zeile ausgeführt wird, und die Ausführung dann zunächst angehalten wird. Sie kann später fortgesetzt werden; dabei starten alle Variablen wieder mit dem Wert, den sie zum Zeitpunkt des Anhaltens am Haltepunkt hatten.

Haltepunkte können beispielsweise nützlich sein, um zu testen, an welcher Stelle ein Fehler im Programm ausgelöst wird, der das Programm zum Absturz bringt. Läuft das Programm nämlich fehlerfrei bis zum Haltepunkt durch, liegt die Fehlerursache auf jeden Fall nicht in dem Code-Teil oberhalb des Haltepunkts. Wenn Sie einen Haltepunkt im Code-Block eines `if-else`-Konstruktks setzen, können Sie leicht feststellen, ob dieser Zweig des Konstruktks tatsächlich durchlaufen wird, ob also die jeweilige Bedingung erfüllt ist. Hält das Programm nämlich an, ist dies der Fall, hält es aber nicht an und läuft bis zum Ende durch, war die Bedingung offenbar nicht erfüllt. Auch können Sie mit Haltepunkten die Programmausführung pausieren, um sich den Inhalt von Variablen, die im Programm verwendet werden, anzuschauen; das geschieht mit *Watches*, die wir uns als nächstes genauer ansehen.

#### ■ Haltepunkte setzen und wieder entfernen

Bevor Sie aber richtig mit dem Debugging beginnen können, müssen Sie zunächst einmal einen Haltepunkt setzen. Dazu klicken Sie im Code-Editor in den Randstreifen neben dem Eingabebereich, dort, wo auch die Zeilennummern zu sehen sind. Es erscheint eine rote Markierung, die den Haltepunkt signalisiert. Das sehen Sie in Abb. 26.1. Ein erneuter Klick an derselben Stelle entfernt den Haltepunkt wieder. Natürlich können Sie bei Bedarf auch mehrere Haltepunkte an verschiedenen Stellen des Programms setzen. Gleichzeitig öffnet sich im unteren Bereich des Fensters eine neue Konsole 5: *Debug* und Ihr Programm wird direkt im Debug-Modus gestartet. Die Debug-Konsole, die Sie auch in Abb. 26.2 sehen, ist letztlich nichts anderes als eine Run-Konsole mit einigen speziellen Funktionen.

Nun können Sie das Programm ausführen. Das geschieht durch Klick auf das Käfersymbol in der Symbolleiste (Abb. 26.3), dasselbe Symbol auf der links an der Debug-Konsole angebrachten Symbolleiste oder durch Drücken der Tastenkombination `<SHIFT>+<F9>`. Beachten Sie bitte, dass das Programm, wenn Sie es über die normale Run-Funktion (zum Beispiel mit Hilfe des grünen Pfeil-Buttons von der Symbolleiste) starten, *nicht* im Debug-Modus ausgeführt wird! Das heißt, Ihr Programm wird dann auch nicht am Haltepunkt anhalten. Wenn Sie Ihr Programm debuggen wollen, müssen Sie es auch im Debug-Modus starten.



■ Abb. 26.1 Code mit gesetztem Haltepunkt in PyCharm

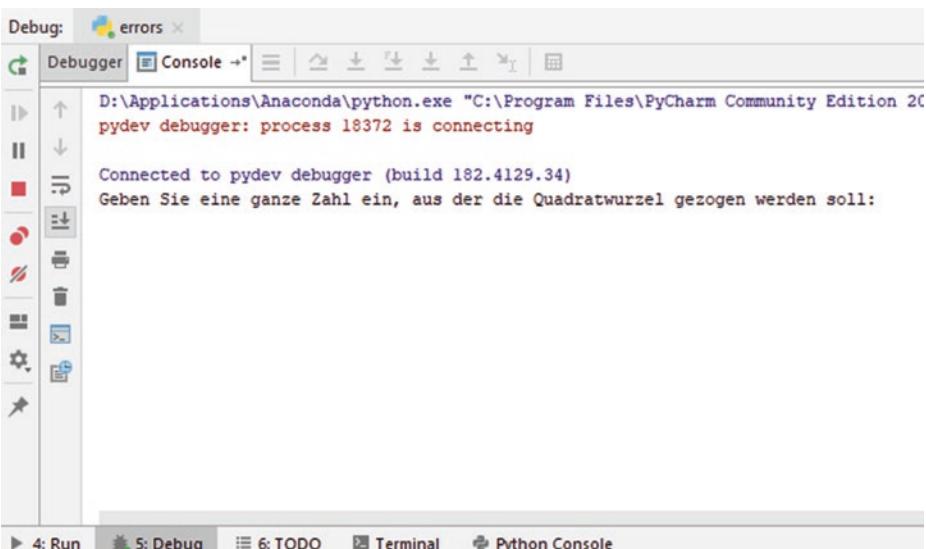


Abb. 26.2 Debug-Konsole in *PyCharm*

```

1  from math import sqrt
2
3  x_str = input('Geben Sie eine ganze Zahl ein, aus der die Quadratwurzel gezogen werden soll: ')
4  if x_str.isnumeric():
5      x_float = float(x_str)
6      print(sqrt(x_float))
7  else:
8      print('Ihre Eingabe ', x_str, ' ist keine Zahl!', sep='')
```

Abb. 26.3 *PyCharm*-Symbolleiste mit Debug-Symbol (Käfer rechts neben Pfeil-Symbol)

Nachdem das Programm den Haltepunkt erreicht hat, wird die weitere Ausführung zunächst ausgesetzt, und zwar *bevor* die Zeile, in der der Haltepunkt gesetzt ist, ausgeführt wird. Fortsetzen können Sie sie mit einem Klick auf den grünen Fortsetzungspfeil in der am linken Rand der Debug-Konsole platzierten Symbolleiste, oder wiederum durch Klicken des Käfersymbols in der *PyCharm*-Symbolleiste oder durch Drücken der Tastenkombination <SHIFT>+<F9>. Das Programm läuft dann weiter bis zum nächsten Haltepunkt oder bis zum Programmende, wenn kein weiterer Haltepunkt der Programmausführung im Wege steht.

Stoppen können Sie das Programm jederzeit mit dem roten Stop-Symbol. Dann wird das Programm vollständig zurückgesetzt, das heißt, beim nächsten Start im Debug-Modus läuft das Programm wieder von Anfang an und hält wieder am ersten Haltepunkt, der bei der Ausführung erreicht wird.

Durch Klick mit der rechten Maustaste auf das rote, runde Haltepunktsymbol am Rand neben der Code-Zeile, in der Sie den Haltepunkt installiert haben, können Sie den Haltepunkt mit einer Bedingung versehen. In unserem Beispiel aus Abb. 26.1 könnten wir zum Beispiel die Bedingung `x_float < 20` eingeben. Auf

diese Weise würde die Programmausführung im Debug-Modus nur dann am Haltepunkt stoppen, wenn die Variable `x_float` einen Wert kleiner 20 hat. Andernfalls würde das Programm weiterlaufen, als gäbe es den Haltepunkt überhaupt nicht.

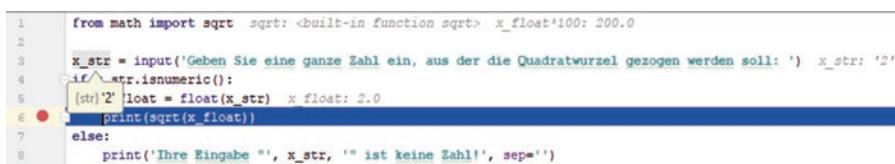
## 26.2.2 Anzeige des Variableninhalts und Verwendung von Watches

Ihnen wird sicher aufgefallen sein, dass beim Erreichen eines Haltepunktes die Debug-Konsole umschaltet auf Ansicht namens *Debugger*. Auf der rechten Seite dieses Bereichs sieht man die bis hierher im Programm verwendeten Variablen und ihre Werte (Abb. 26.2). Die Werte werden außerdem direkt im Programmcode eingeblendet, jeweils hinter den Zeilen, in denen die Variablen verwendet werden, und sind hinsichtlich des Syntax-Highlightings formatiert wie Kommentare. Darüber hinaus können Sie sich den Inhalt einer Variablen jederzeit anzeigen lassen, indem Sie mit der Maus über der Variablen stehen bleiben; es erscheint sogleich ein kleines Popup mit Typ und Wert der Variablen. Beides sehen Sie in Abb. 26.4.

Der Debugger erlaubt Ihnen allerdings nicht einfach nur die Anzeige des Variableninhalts. Sie können auch ganze Ausdrücke, die auf Variablen basieren, formulieren und sich deren Wert dann am Haltepunkt anzeigen lassen. Weil man die Werte dieser Ausdrücke überwacht, spricht man in dem Zusammenhang auch von *Watches*. Klicken Sie auf der Symbolleiste links neben dem Variablen-Bereich der Debug-Konsole auf das Plus-Button („New Watch“). Danach entsteht ein neuer, zunächst leerer Eintrag in der Variablenliste auf der rechten Seite. Hier können Sie nun den Ausdruck, dessen Wert Sie überwachen wollen, eingeben, also zum Beispiel `x_float < 5`. Dieser Ausdruck ist entweder wahr oder falsch, also vom Typ `bool`. Sein Wert wird in der Variablen- und Watchliste angezeigt, sobald die Programmausführung den Haltepunkt erreicht. Wie Sie in Abb. 26.5 sehen, können Sie bei der Formulierung der überwachten Ausdrücke natürlich auch auf die Python-Funktionen zurückgreifen.

### ■ Den Variableninhalt manuell anzeigen lassen

Der Debugger bietet zur Überwachung der Werte von Variablen oder auf ihnen basierenden Ausdrücken sehr praktische Funktionalitäten. Häufig genügt aber bereits ein sehr viel einfacheres Vorgehen, um die Fehlersuche durch die Anzeige der Werte von Variablen und Ausdrücken zu unterstützen: Die Ausgabe der Werte im Programmcode mit Hilfe der Funktion `print()`. Fügen Sie also einfach an den rele-



```

1 from math import sqrt  sqrt: <built-in function sqrt>  x_float: 200.0
2
3 x_str = input('Geben Sie eine ganze Zahl ein, aus der die Quadratwurzel gezogen werden soll: ')  x_str: '2'
4 if not x_str.isnumeric():
5     float(x_str)  x_float: 2.0
6     print(sqrt(x_float))
7 else:
8     print('Ihre Eingabe "', x_str, '" ist keine Zahl!', sep=' ')

```

Abb. 26.4 Anzeige des Variableninhalts hinter den Code-Zeilen und in Mouseover-Popup

Abb. 26.5 Watches im Variablen- und Watchbereich

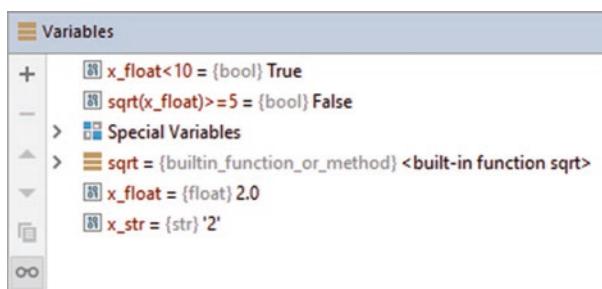


Abb. 26.6 Schrittweise Ausführung des Programms mit Step Over



26

vanten Stellen im Programm vorläufig `print()`-Anweisungen ein und schon sehen Sie die entsprechenden Ausgaben in der Run-Konsole. Einfacher geht es nicht! Wie bereits die Haltepunkte, können Sie solche Ausgaben natürlich auch dazu verwenden, um zu überprüfen, ob bestimmte Teile Ihres Programms überhaupt durchlaufen werden. Vergessen Sie nur nach dem Debuggen nicht, diese Ausgabeanweisungen wieder zu entfernen. Der Anwender Ihres Programms wird es Ihnen danken.

### 26.2.3 Schrittweise Ausführung

Gerade, wenn Sie mit Watches arbeiten, aber auch zur Ermittlung der genauen Zeile, in der ein Fehler auftritt, ist eine Debugger-Funktionalität hilfreich, die als *schrittweise Ausführung* bezeichnet wird. Dabei wird das Programm Zeile für Zeile ausgeführt, als wäre in jeder einzelnen Zeile ein Haltepunkt gesetzt.

Beginnen können Sie damit, sobald Ihr Programm einen Haltepunkt erreicht hat. Klicken Sie, um von dort aus schrittweise weiterzumachen, auf den Button „Step over“ (Abb. 26.6) oder drücken Sie die Taste <F8>. Damit wird die nächste Zeile Ihres Programms ausgeführt. Mit weiteren Klicks auf den Button oder Drücken von <F8> wird jeweils eine weitere Zeile ausgeführt. Auf diese Weise können Sie sich schrittweise durch Ihr Programm bewegen und seine Ausführung beobachten. Wollen Sie zwischenzeitlich die schrittweise Ausführung verlassen, klicken Sie, wie bereits bei der Verwendung von Haltepunkten, auf das rote Stop-Symbol in der PyCharm-Symbolleiste, der Symbolleiste am linken Rand der Debug-Konsole oder drücken Sie <STRG>+<F2>.

### 26.3 Zusammenfassung

In diesem Kapitel haben wir uns damit beschäftigt, wie in Python Fehler *zur Laufzeit* durch Abprüfen kritischer Zustände im Programm sowie die geeignete Behandlung von Ausnahmen (Laufzeitfehlern) verarbeitet und zuvor bereits *zur Entwicklungszeit* mit Hilfe von Debugging-Werkzeugen analysiert werden können.

## 26.4 · Lösungen zu den Aufgaben

Folgende Punkte sollten Sie aus diesem Kapitel unbedingt mitnehmen:

- Durch geschicktes Abprüfen kritischer Bedingungen können Sie Ihr Programm robuster gegenüber Fehleingaben oder anderen Situationen machen, in denen ein regulärer Programmablauf nicht mehr gewährleistet ist.
- Laufzeitfehler werden in Python durch sogenannte Ausnahmen (Exceptions) repräsentiert.
- Anweisungen, bei denen die Gefahr besteht, dass sie eine Ausnahme auslösen könnten, lassen sich in ein **try...except**-Konstrukt verpacken; es „versucht“ die Anweisung auszuführen und bietet die Möglichkeit einer Fehlerbehandlung mit eigenem Programmcode, sollte die Ausnahme ausgelöst („geworfen“ werden).
- Die einfachste Form des **try...except**-Konstrukts lautet: **try: code\_block except: code\_block**; auch kann die abzufangende Ausnahme explizit angegeben werden, sodass für ein- und dasselbe Code-Segment unterschiedliche Fehlerbehandlungs routinen definiert werden können, je nachdem welche Ausnahme geworfen wird.
- Integrierte Entwicklungsumgebungen wie *PyCharm* bieten einige Möglichkeiten der Fehleranalyse während der Entwicklung; dazu gehören Haltepunkte (Stelle, an der die Ausführung des Programmcodes zunächst angehalten wird), Watches (Anzeige des Variableninhalts während das Programm läuft) und die schrittweise Ausführung (wobei die Ausführung der jeweils nächsten Anweisung wird durch den Entwickler explizit ausgelöst wird).

## 26.4 Lösungen zu den Aufgaben

### ■ Aufgabe 26.1

Eine einfache Formulierung des Programms könnte so lauten:

```
zahl1 = input("Zahl 1: ")
op = input("Operator (+, -, *, /): ")
zahl2 = input("Zahl 2: ")

try:
    if op == "+":
        ergebnis = float(zahl1) + float(zahl2)
    else:
        if op == "-":
            ergebnis = float(zahl1) - float(zahl2)
        else:
            if op == "*":
                ergebnis = float(zahl1) * float(zahl2)
            else:
                ergebnis = float(zahl1) / float(zahl2)
    print("Ergebnis von ", zahl1, " ", op, " ", zahl2, ":", ,
          ergebnis, sep = "")
except:
    print("Bei der Berechnung von ", zahl1, " ", op, " ",
          zahl2, " ist ein Fehler aufgetreten.", sep = "")
```

Der Programmabschnitt, der davor geschützt werden soll, auf einen unkontrollierten Fehler zu laufen, wird hier einfach in einem großen **try...except**-Konstrukt verpackt. Tritt bei der Programmausführung tatsächlich ein Fehler auf, so erhält der Benutzer eine allgemeine Fehlermeldung. Eine genauere Beschreibung des Fehlers wäre möglich, wenn die kritischen Programmabschnitte, zum Beispiel also das Umwandeln der Zahleneingaben vom Typ **str** in den Typ **float** jeweils in separaten **try...except**-Konstrukten ausgeführt würden. Der Rest des Programms würde dann jeweils im **except**-Block des vorangegangenen **try...except**-Konstrukts ausgeführt werden, sodass man letztlich eine mehrfache Verschachtelung dieser Konstrukte hätte, die zwar nicht leicht zu lesen ist, aber doch wenigstens dem Benutzer eine genauere Analyse der Fehlerursache bietet.

### ■ Aufgabe 26.2

26

Eine Alternative zur Verwendung des **try...except**-Konstrukts ist – zumindest in diesen Fällen hier – das explizite Auffangen von Fehlern mit Hilfe geeigneter Bedingungen. In der folgenden Formulierung des Programms werden eine Reihe solcher Bedingungen geprüft. Schlägt eine Fehlerbedingung an, wird eine **bool**-Variable **fehler** auf **True** gesetzt; außerdem wird eine **str**-Variable **meldung** mit einer geeigneten Fehlermeldung befüllt. Gegen Ende des Programms wird dann geprüft, ob das Programm fehlerfrei durchgelaufen ist (**fehler == False**); wenn ja, wird die Variable **meldung** mit dem Ergebnis der Berechnung versorgt. Schließlich muss dann nur noch der Inhalt von **meldung** an den Benutzer ausgegeben werden, die dann also entweder das Ergebnis der Berechnung oder eben eine Fehlermeldung enthält.

Beachten Sie hier übrigens den Trick, mit dessen Hilfe wir prüfen, ob die Variablen **zahl1** und **zahl2** nach dem Einlesen mit **input()** wirklich eine Zahl beinhalten. Die **str**-Methode **isdigit()** prüft, ob eine **str**-Variable nur Ziffern beinhaltet. Nun kann der Benutzer natürlich auch ein Dezimaltrennzeichen mit eingegeben haben (also einen Punkt). Deshalb eliminieren wir diesen zunächst mit **replace()**. Außerdem prüfen wir, ob das Dezimaltrennzeichen auch höchstens einmal in dem String vorkommen, denn eine ungültige „Zahl“ hätten wir auch dann, wenn der Dezimaltrenner mehr als einmal in der Zeichenkette auftaucht.

Wie Sie sehen, kann man also auch mit Bedingungen viele Fehler gut abfangen. Das gelingt aber nicht mit allen Fehlern. Bei der Arbeit mit Dateien zum Beispiel ist es schwierig, alle erdenklichen Fehlerursachen mit Bedingungen abzuprüfen (vielleicht noch die Existenz einer Datei, wenn man aus ihr lesen möchte). In diesen Fällen bietet sich dann die Verwendung von **try...except**-Konstrukten an.

## 26.4 · Lösungen zu den Aufgaben

```
zahll1 = input('Zahl 1: ')
op = input('Operator (+,-,*,/): ')
zahl2 = input('Zahl 2: ')

fehler = False

if not zahll1.replace('. ', '').isdigit() and \
    zahll1.count('.') <= 1:
    meldung = zahll1 + ' ist keine gültige Zahl! (ggf. Komma) \
        statt Punkt als Dezimaltrennzeichen' \
        ' verwendet?'
    fehler = True
else:
    if not zahl2.replace('. ', '').isdigit() and \
        zahl2.count('.') <= 1:
        meldung = zahll1 + ' ist keine gültige Zahl! (ggf. Komma) \
            statt Punkt als Dezimaltrennzeichen' \
            ' verwendet?'
        fehler = True
    else:
        if not op in ('+', '- ', '*', '/'):
            meldung = op + ' ist kein gültiger Operator. ' \
                'Bitte +,-,* oder / eingeben!'
        else:
            if op == '+':
                ergebnis = float(zahll1) + float(zahl2)
            else:
                if op == '-':
                    ergebnis = float(zahll1) - float(zahl2)
                else:
                    if op == '*':
                        ergebnis = float(zahll1) * float(zahl2)
                    else:
                        if zahl2 == 0:
                            meldung = 'Achtung: Keine Division' \
                                ' durch Null möglich!'
                            fehler = True
                        else:
                            ergebnis = float(zahll1) / float(
                                zahl2)
if fehler == False:
    meldung = 'Ergebnis: ' + str(zahll1) + ' ' + op + ' ' + \
        str(zahl2) + ' = ' + str(ergebnis)

print(meldung)
```

# JavaScript

## Inhaltsverzeichnis

**Kapitel 27 Einführung – 433**

**Kapitel 28 Was brauche ich zum Programmieren? – 437**

**Kapitel 29 Was muss ich tun, um ein Programm zum Laufen zu bringen? – 441**

**Kapitel 30 Wie stelle ich sicher, dass ich (und andere) mein Programm später noch verstehe? – 451**

**Kapitel 31 Wie speichere ich Daten, um mit ihnen zu arbeiten? – 457**

**Kapitel 32 Wie lasse ich Daten ein- und ausgeben? – 491**

**Kapitel 33 Wie arbeite ich mit Programmfunctionen, um Daten zu bearbeiten und Aktionen auszulösen? – 535**

- Kapitel 34** Wie steuere ich den Programmablauf und lasse das Programm auf Benutzeraktionen und andere Ereignisse reagieren? – 557
- Kapitel 35** Wie wiederhole ich Programmanweisungen effizient? – 571
- Kapitel 36** Wie suche und behebe ich Fehler auf strukturierte Art und Weise? – 587



# Einführung

## Übersicht

In diesem Teil wenden wir uns nun mit JavaScript der zweiten Programmiersprache zu, mit der wir in diesem Buch beschäftigen werden. Auch diese werden wir uns nach dem 9-Fragen-Schema erschließen, ebenso wie wir es mit Python getan haben.

In diesem ersten Kapitel des Teils verschaffen wir uns einen kurzen Überblick über Anwendungsbereiche, Bedeutung und Herkunft von JavaScript.

JavaScript ist die Sprache des Web. Wahrscheinlich kommt keine andere Sprache häufiger zum Einsatz, wenn es um die Programmierung von Web-Frontends, also den modernen Web-Benutzeroberflächen, geht als JavaScript. Kaum eine besucherstarke Seite kommt heutzutage ohne JavaScript aus.

Wenn Sie Formulare auf Webseiten sehen, die Ihre Eingaben validieren, also zum Beispiel prüfen, ob die von Ihnen angegebene E-Mail-Adresse oder Telefonnummer ein gültiges Format hat, ist meist JavaScript im Spiel. Wenn Sie nach Eingabe bereits weniger Buchstaben in ein Suchfeld Vorschläge für mögliche Suchbegriffe angezeigt bekommen, erledigt meist JavaScript im Hintergrund den Job. Wenn eine Webseite mit Animationen arbeitet, zum Beispiel Elemente in Abhängigkeit von Ihren Klicks oder Mausbewegungen ein- oder ausblendet, oder sie besonders hervorhebt, steht im Hintergrund oft ein JavaScript-Programm.

Kein Wunder also, dass JavaScript auf praktischen allen Ranglisten unter den populärsten Programmiersprachen auftaucht (siehe auch ► Kap. 6).

JavaScript hat – anders als man vielleicht beim ersten Lesen denken mag – wenig mit einer anderen populären, aber ungleich schwerer zu lernenden Programmiersprache zu tun: Java. Lediglich einige syntaktische Ähnlichkeiten gibt es zwischen den beiden Namensvettern.

Die Anfänge der Sprache reichen zurück in jene Zeit, als ein Programm namens *Netscape Navigator* der marktdominierende Browser war. Im Jahr 1995, kurz bevor der jahrelang andauernde sogenannte „Browserkrieg“ um die Vorherrschaft in dem damals als Schlüsselmarkt ausgemachten Browserbereich zwischen Netscape und Microsoft entbrannte, veröffentlichte Netscape eine Sprache namens Live-Script. Sie war von Brendan Eich entwickelt worden und wurde wenig später im Zuge einer Kooperation mit Sun Microsystems in JavaScript umbenannt. Nicht gänzlich geklärt ist, ob die Namensgebung damit zu tun hatte, dass JavaScript tatsächlich mit kleinen Java-Anwendungen (sogenannten Java Applets) auf Webseiten zusammenarbeiten sollte, oder ob die Benennung hauptsächlich unter Marketingaspekten erfolgte, um sich ein wenig im Glanze des an Popularität gewinnenden Java zu sonnen; sicher jedoch ist, dass JavaScript schnell zur dominanten Programmiersprache des Web aufstieg und Konkurrenten wie Microsofts VBScript vom Markt verdrängte.

Netscape bemühte sich früh darum, die Sprache standardisieren zu lassen und rief dazu die Standardisierungsorganisation Ecma an (deren Name damals noch als großgeschriebenes Akronym für *European Computer Manufacturers Association* stand). Die brachten in der Folge tatsächlich den Standard ECMA-262 heraus, der eine Sprache namens ECMAScript definiert. JavaScript galt fortan als Imple-

mentierung von ECMAScript, neben anderen Sprachen, die den Standard (ergänzt um eigene Spezifika) ebenfalls implementieren, wie beispielsweise Adobes ActionScript oder Microsofts TypeScript.

Die Diskussion um Standardisierung führt aber im Fall von JavaScript streng genommen in die falsche Richtung, denn tatsächlich ist JavaScript eigentlich alles andere als standardisiert. Das hängt unter vor allem damit zusammen, dass es mehrere populäre Implementierungen von JavaScript gibt. JavaScript-Programme laufen im Web-Browser, werden also von diesem heruntergeladen und dann interpretiert. Jeder Browser, der JavaScript unterstützt, bringt einen eigenen Interpreter mit, eine JavaScript-Engine, und diese Engines unterscheiden sich durchaus zwischen den Herstellern. Microsoft JavaScript-Dialekt namens JScript, der zwar dem ECMA-262-Standard folgt, aber eine Reihe von Spezifika mitbringt, funktioniert deshalb in Nuancen anders als etwa die JavaScript-Variante, die von Googles *Chrome*-Browser interpretiert wird. Schlimmstenfalls können diese Unterschiede dazu führen, dass bestimmte Features einer Webseite in einem Browser funktionieren, in einem anderen jedoch nicht. Der Teufel steckt im Detail. Dem Teufel werden wir freilich in den folgenden Kapiteln kaum begegnen, beschäftigen wir uns doch mit Sprachelementen, die ECMA-262-kompatibel sind und praktisch überall unterstützt werden, wo JavaScript interpretiert wird.

JavaScript ist also eine Sprache, die normalerweise vom Web-Browser, und damit auf der *Client*-Seite interpretiert wird. Sie ist damit in gewissem Sinne das Gegenstück zu PHP, das auf dem Web-Server läuft und die Webseite bereits verändern kann, bevor Sie überhaupt an den Browser geschickt wird (zum Beispiel, indem aus einer Datenbank Datensätze, etwa Produktinformationen, gelesen und auf der Webseite dargestellt werden). Es gibt aber Laufzeitumgebungen wie Node.js, die eine JavaScript-Implementierung bereitstellen, mit der JavaScript auch serverseitig ausgeführt werden kann.

Wir werden uns im Folgenden allerdings auf die klassische Variante konzentrieren, JavaScript nämlich, dass in einer Webseite läuft und so die Webseite dynamisch gestaltbar macht.

Wie bereits im Python-Teil dieses Buches folgen wir wieder den 9 großen Frageblöcken, um uns einen Überblick über die Sprache zu verschaffen.



# Was brauche ich zum Programmieren?

## Inhaltsverzeichnis

- 28.1    Interpreter – 438**
- 28.2    Code-Editoren und Entwick-  
lungsumgebungen – 438**
- 28.3    Hilfe und Dokumentation – 439**
- 28.4    Zusammenfassung – 440**

## Übersicht

In diesem Kapitel werden wir uns mit der Frage beschäftigen, wie JavaScript ausgeführt wird, welche Entwicklungswerkzeuge zur Verfügung stehen und wo man weiterführende Informationen zu JavaScript findet.

Wie Sie sehen werden, benötigen Sie nicht viel, um mit dem Programmieren JavaScript loslegen zu können. Wir werden uns deshalb auch nur mit der „Grundausrüstung“ an Werkzeugen begnügen und nicht mit einer richtigen Integrierten Entwicklungsumgebung arbeiten, wie wir es bei Python getan haben.

In diesem Kapitel werden Sie lernen:

- wie JavaScript-Code interpretiert wird, und was das für die Werkzeuge bedeutet, die Sie brauchen, um JavaScript-Code auszuführen
- wie Sie JavaScript-Code bearbeiten können
- wo Sie bei Bedarf Hilfe im Internet bekommen.

### 28.1 Interpreter

28

Wir hatten bereits in der Einführung zu diesem Teil des Buches gesehen, dass JavaScript eine Programmiersprache ist, die bei der Entwicklung von Webseiten breite Verwendung findet. Damit ist klar, dass die Ausführung von JavaScript-Programmen etwas anders funktionieren muss, als etwa die eines Python-Programms. Denn dem Nutzer einer Webseite kann ja nicht zugemutet werden, erst mühsam einen Interpreter herunterladen, bevor er die Webseite betrachten kann. Und tatsächlich ist das auch gar nicht nötig. Alle modernen Browser bringen nämlich die Fähigkeit, JavaScript zu interpretieren, von Haus aus mit. Deshalb müssen auch Sie als Entwickler nicht extra einen Interpreter installieren. Ihr Browser erledigt den Job!

### 28.2 Code-Editoren und Entwicklungsumgebungen

Auf dem Markt sind eine ganze Reihe integrierte Entwicklungsumgebungen (IDEs) verfügbar, die (unter anderem) JavaScript unterstützen. Dazu gehören zum Beispiel JetBrains‘ *WebStorm* (vom selben Hersteller, der auch das im letzten Teil des Buchs verwendete *PyCharm* anbietet) und *NetBeans* von der Apache Software Foundation. Manche sind kommerziell (wie *WebStorm*), andere kostenfrei erhältlich (wie *NetBeans*).

Wir werden hier allerdings einen anderen Weg einschlagen: Nachdem wir bei Python im letzten Teil dieses Buchs mit einer hochentwickelten IDE gearbeitet haben, werden wir uns jetzt bei JavaScript mit einem ungleich bescheideneren Werkzeug begnügen, nämlich einem reinen Code-Editor. Da unser JavaScript-Code ohnehin in eine Webseite eingebunden und dann im Browser ausgeführt wird, brauchen wir uns nicht mit der Frage zu beschäftigen, wie genau wir den Interpreter in den Code-Editor einbinden können. Wir schreiben einfach den Java-

Script-Code und die Webseite, die ihn verwendet, in unserem Code-Editor und schauen und das Ergebnis dann im Webbrowser an.

Verwendet wird im Folgenden der beliebte Editor *SublimeText*, der nach den Lizenzbedingungen zum Zeitpunkt, als diese Zeilen geschrieben werden, kostenlos während einer zeitlich nicht festgelegten Testphase zum Ausprobieren verwendet werden kann, für die langfristige Nutzung aber den Kauf einer Lizenz voraussetzt. Alternativ können Sie natürlich andere Editoren einsetzen, wie etwa Microsofts *Visual Studio Code* oder *Notepad++*. Letzten Endes wird nur ein beliebiger Editor benötigt, der es erlaubt, den eigentlichen JavaScript-Code und die Webseite, in die er eingebettet wird, zu bearbeiten, alles Übrige erledigen wir im Webbrowser. Auch wenn Sie in der Auswahl eines Editors daher vollkommen frei sind, empfiehlt es sich doch, sich für ein Werkzeug zu entscheiden, das Syntax-Highlighting für JavaScript und HTML unterstützt, entweder von Haus aus oder durch ein entsprechendes Add-in.

Wenn Sie JavaScript erst mal einfach nur ausprobieren wollen und dabei die Mühe, Ihren JavaScript-Code in eine Webseite einbinden zu müssen, weitestgehend vermeiden wollen, dann sind Webdienste wie *JS Bin*, *JS Do* oder *Plunker* für Sie von Interesse. Diese Dienste erlauben es, JavaScript-Code direkt einzugeben und ohne weitere Umstände auszuführen. Sie sehen dann in ein- und demselben Browser-Fenster Ihren Code und das Ergebnis der Ausführung. Im folgenden Kapitel werden wir einen kurzen Blick auf diese Webdienste werfen. Wenn Sie allerdings ernsthaft JavaScript lernen und damit arbeiten wollen, sollten Sie sich für einen Code-Editor oder eine entsprechende IDE entscheiden.

## 28.3 Hilfe und Dokumentation

---

Wie für alle populären Programmiersprachen, finden sich auch für JavaScript unzählige Informationsquellen im Internet.

Eine wirklich offizielle, gut verwendbare Dokumentation für JavaScript existiert indes nicht. ECMA, die den offiziellen Sprachstandard vorgibt, stellt die Sprachspezifikation bereit (für Version 10 von ECMAScript aus dem Juni 2019, erreichbar unter ► <http://www.ecma-international.org/ecma-262/10.0/>). Die ist aber für die praktische Arbeit eher weniger genießbar.

Eine gute, anwendungsorientierte Dokumentation liefert die Mozilla Foundation (► <https://developer.mozilla.org/en-US/docs/Web/JavaScript>), jene Organisation also, die mit ihrem *Firefox*-Browser die Nachfolge von JavaScript-Erfinder Netscape angetreten hat. Hier findet sich unter anderem zu allen Standardfunktionen von JavaScript eine Hilfeseite mit einer allgemeinen Beschreibung, den Argumenten, mit denen die Funktion aufgerufen werden muss, dem Rückgabewert, den sie liefert und einigen Anwendungsbeispielen. Außerdem zeigt die Hilfeseite jeweils auch eine Übersicht, inwiefern und seit welcher Version die JavaScript-Interpreter der wichtigsten Browser wie Internet-Explorer, Edge, *Chrome*, *Safari*, *Firefox* oder *Opera* die Funktion jeweils unterstützen. Schauen Sie sich als Beispiel einmal die Hilfeseite der Funktion `sqrt()` an, die die Quadratwurzel einer Zahl zieht: ► <https://>

[developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Math/sqrt](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/sqrt).

Daneben ist natürlich – wie bei praktisch allen bekannteren Programmiersprachen – für die Lösung konkreter Problem wiederum das Programmierer-Forum *StackOverflow* eine hervorragende Informationsquelle, in dem man für eine Vielzahl häufig (und weniger häufig) auftretender Probleme bereits qualitativ sehr gute Antworten vorfindet und, falls das einmal doch nicht der Fall sein sollte, einen eigenen Frage-Thread eröffnen kann.

Darüber hinaus gibt es natürlich unzählige Artikel, Tutorials, Blogs, Videos und andere Formate im Internet, die – zugeschnitten auf die unterschiedlichsten Erfahrungslevels – jeden nur erdenklichen Aspekt der populären Sprache JavaScript beleuchten.

## 28.4 Zusammenfassung

---

In diesem Kapitel haben wir uns damit beschäftigt, wie JavaScript interpretiert wird und welche Werkzeuge sie benötigten, um JavaScript-Code zu schreiben und auszuführen.

28

Folgende Punkte sollten Sie aus diesem Kapitel unbedingt mitnehmen:

- JavaScript wird vom WebBrowser interpretiert; einen separaten Interpreter oder Compiler benötigen Sie daher nicht.
- Zur Bearbeitung des JavaScript-Codes (und der Webseite, in die er eingebunden wird) genügt ein Texteditor; natürlich stehen aber auch für JavaScript integrierte Entwicklungsumgebungen wie *NetBeans* oder *WebStorm* zur Verfügung.
- Die offizielle Dokumentation von JavaScript ist eher überschaubar. Am besten verwendbar ist noch die Dokumentation der Mozilla Foundation.
- Wie auch bei vielen Programmiersprachen, ist auch im Falle von JavaScript *StackOverflow* ein guter Anlaufpunkt bei Fragen.
- Darüber hinaus existieren eine Unmenge „inoffizielle“ Informationsquellen im Internet, die für alle praktischen Programmierprobleme gute Dienste leisten.



# Was muss ich tun, um ein Programm zum Laufen zu bringen?

## Inhaltsverzeichnis

- 29.1 Einbinden von JavaScript-Code  
in Webseiten – 442**
  - 29.1.1 Das script-Element in HTML – 442
  - 29.1.2 Sicherheitsaspekte – 446
- 29.2 „Hallo Welt“ in JavaScript – 447**
  - 29.2.1 Do-it-yourself: Der (gar nicht so)  
mühsame Weg – 447
  - 29.2.2 Mit etwas Nachhilfe: Die  
Schnelle Umsetzung mit einem  
Webdienst – 449
- 29.3 Zusammenfassung – 450**

## Übersicht

JavaScript-Programme sind typischerweise Bestandteil von Webseiten. Um also ein Programm zum Laufen zu bringen, müssen wir uns als erstes damit beschäftigen, wie wir es in eine Webseite einbinden können. Darum geht es in diesem Kapitel.

In diesem Kapitel werden Sie lernen:

- wie Sie mit dem HTML-Element **script** aus einer Webseite heraus auf ein JavaScript-Skript verweisen können (wenn Sie noch nicht mit HTML vertraut sind, erhalten Sie in diesem Kapitel zugleich eine kleine Einführung in die „Programmiersprache des Web“)
- wie Sie mit Hilfe des **script**-Elements JavaScript-Code direkt in eine Webseite einbetten können.
- wie Sie dank spezieller Webservices JavaScript ausprobieren können, ohne sich um das Einbinden Ihres Skripts in die Webseite kümmern zu müssen
- wie Sie Ihr erstes kleines „Hallo-Welt“-Programm in JavaScript schreiben.

## 29.1 Einbinden von JavaScript-Code in Webseiten

### 29.1.1 Das script-Element in HTML

29

Wie Sie bereits wissen, läuft JavaScript nicht alleine mit Hilfe eines Interpreters, wie es beispielsweise Python tut, sondern wird in eine Webseite eingebettet und mit dieser Webseite zusammen vom Browser interpretiert.

Das bedeutet, dass wir uns zwar nicht mit der Frage beschäftigen müssen, wie ein JavaScript-Programm ausgeführt wird, denn das übernimmt der Browser für uns; sehr wohl allerdings müssen wir uns damit befassen, wie ein JavaScript-Programm in eine Webseite eingebunden wird.

Webseiten bestehen aus Code, der in der deskriptiven Programmiersprachen HTML (*Hypertext Markup Language*) verfasst ist; zur begrifflichen Abgrenzung deskriktiver von anderen Sprachen, blättern Sie nochmal zurück zu ► Abschn. 3.1.

Wenn Sie bereits wissen, wie HTML in den Grundzügen (und mehr werden wir hier auch gar nicht brauchen) funktioniert, dann lesen Sie einfach weiter. Wenn Sie aber denken, Ihre Kenntnisse vertragen eine kurze Auffrischung, dann wenden Sie sich, bevor Sie weiterlesen, zunächst dem Hintergrund-Kasten *HTML (Hypertext Markup Language)* zu.

#### HTML (Hypertext Markup Language)

##### Die Sprache des Web

HTML ist die Sprache, in der Webseiten verfasst werden. Sie wurde Ende der 80er Jahres des 20. Jahrhunderts von dem Physiker Tim Berners-Lee am Europäischen Kernforschungszentrum CERN entwickelt, der deshalb nicht nur von der englischen Königin in den Adelstand erhoben, mit unzähligen weiteren Ehrungen überhäuft und vom Time Magazine auf die Liste der *100 Most Important People of the 20th Century* gesetzt wurde, sondern gemeinhin auch als Vater des World Wide Webs gilt. Heute wird der HTML-Standard vom *World Wide Web Consortium* (W3C) definiert.

## HTML-Elemente und ihre Komponenten

HTML-Dokumente bestehen aus HTML-*Elementen*.

Solche HTML-Elemente besitzen grundsätzlich drei Komponenten:

- **Tags:** Tags sind Symbole, die den Beginn und das Ende eines HTML-Elements markieren; um sie von anderen Bestandteilen des Dokuments abzugrenzen, werden sie von Kleiner- und Größer-Zeichen eingeschlossen, wobei dem Element-Bezeichner beim schließenden Tag ein Querstrich (*/*) vorangestellt wird. Eine Kombination von öffnendem und schließendem Tag könnte damit so aussehen: **<h1>...</h1>**.
- **Inhalt:** Zwischen den Tags kann oftmals ein Inhalt stehen. Das **h1**-Element beispielsweise beschreibt eine Top-Level-Überschrift (*header 1*). Der Text der Überschrift steht dabei zwischen dem öffnendem und dem schließenden Tag, also beispielsweise: **<h1>Einleitung</h1>**.
- **Attribute:** Attribute stehen innerhalb des öffnenden Tags und modifizieren das Verhalten des Tags. Sie bestehen aus einem Attribut-Namen und einem Attribut-Wert. Der Wert des Attributs wird in Anführungszeichen geschrieben und dem Attribut-Namen mit einem Gleichheitszeichen zugeordnet. Das Attribut **dir** beispielsweise steuert die Richtung der Textdarstellung. Seine Ausprägung **rtl** bedeutet *right-to-left*, der Text wird also von rechts nach links dargestellt (rechtsbündig). Eingebunden in unser **h1**-Element sieht das Attribut dann so aus: **<h1 dir="rtl">Einleitung</h1>**.

## Aufbau von HTML-Dokumenten

Viele HTML-Elemente können ineinander verschachtelt werden. Es ergibt sich dann eine hierarchische Struktur, die der Webbrowsere als *Document Object Model (DOM)* einliest und darstellt.

Das wird bereits beim Aufbau eines HTML Dokuments deutlich. HTML-Dokumente bestehen aus einem Kopf (Element **head**) und einem Körper (Element **body**). Im Dokumentenkopf stehen normalerweise übergreifende Informationen zum Dokument, wie etwa der Titel des Dokuments oder Meta-Informationen wie Schlüsselwörter oder eine Beschreibung der Webseite für Suchmaschinen.

Im Dokumentenkörper findet sich der eigentliche Inhalt der Webseite; er ist daher normalerweise der ungleich größere Teil des Dokuments.

Eingebettet sind **head** und **body** in ein umschließendes **html**-Element, das gerne auch als Root-Element (vom englischen *root* = Wurzel) bezeichnet wird, weil es das hierarchisch höchste Element im Document Object Model darstellt.

Eine einfache Website könnte so aussehen:

```
<!DOCTYPE html>
<html>

    <head>
        <title>Test-Seite</title>
    </head>

    <body>
        <h1>Einleitung</h1>
        <p>Hier steht irgendein Text.</p>
    </body>

</html>
```

Das Element **DOCTYPE** ist optional (und ist eines jener besonderen Art von Elementen, die kein schließendes Tag haben), es beschreibt das Dokument als ein HTML-Dokument. Im **body** des Dokuments sehen Sie noch ein weiteres Element, **p**. Es markiert einen Textabsatz und führt normalerweise in der Darstellung dazu, dass über und unter dem Text etwas Abstand gelassen wird und der Text somit als freistehender Absatz erscheint.

Die Tabulator-Einrückungen im Quelltext des HTML-Dokuments sind nur der besseren Lesbarkeit wegen eingefügt worden, sie werden bei der Interpretation des Dokumenteninhalts ebenso ignoriert wie die Groß- und Kleinschreibung der Tags (HTML ist also nicht case-sensitive). Damit ist auch der folgende Code eine gültige Webseite:

```
<!DOCTYPE html><html><head><TITLE>Test-Seite</title></head><body>
<h1>Einleitung</h1><p>Hier steht irgendein Text.</p></BO-
dy></HTML>
```

Fügen Sie diesen Code (am besten mit den Einrückungen) mit Hilfe eines Editors in eine leere Datei ein, speichern Sie sie mit der Endung **.htm** oder **.html** und fertig ist Ihre kleine Webseite, die Sie jetzt mit dem Browser öffnen und betrachten können.

Übrigens: Auch in HTML gibt es natürlich *Kommentare*: Sie besitzen mit **<!--** und **-->** spezielle öffnende und schließende Tags. Kommentare dürfen dabei auch mehrzeilig sein.

Neben dem HTML-Quelltext der Webseiten und Skripts, vor allem den JavaScript-Programmen, mit denen wir uns in diesem Teil des Buches beschäftigen werden, sind *Cascading Style Sheets (CSS)* ein wichtiger Bestandteil moderner Webseiten. Früher war es üblich, Formatierungen mit HTML-Elementen und Attributen direkt im HTML-Quelltext der Webseite festzulegen; zum Beispiel kann ein Text mit dem **align**-Attribut, das unter anderem vom **p**-Element unterstützt wird, auf einfache Weise zentriert werden: **<p align="center"><Hier steht irgendein zentrierter Text</p>**. Wenn nun aber eine Webseite ihr Design grundlegend verändern wollte oder ein „Ausschnitt“ aus einer Webseite auf einer anderen Webseite gezeigt werden sollten, die ein abweichendes Design verwendete, so musste die Formatierung in mühevoller und fehleranfälliger Kleinarbeit im HTML-Code geändert werden.

29

## Cascading Style Sheets (CSS)

Das wurde einfacher mit der Einführung von Cascading Style Sheets. Die Idee hinter Cascading Style Sheets ist es, Struktur und Inhalt einer Webseite einerseits von Design und Formatierung zu trennen, so dass das Design leichter ausgetauscht oder angepasst werden kann, ohne tief in den Teil der Webseite einzugreifen, der Struktur und Inhalte beherbergt. Umgesetzt wird dies dadurch, dass Formatierung und Design in einer separaten Datei (typischerweise mit der Endung **.css**) beschrieben werden und nicht in der HTML-Datei selbst. Im Style Sheet, also in der CSS-Datei, kann zum Beispiel festgelegt werden, dass **h1**-Überschriften immer blau und in Fettsatz dargestellt werden sollen. Wird nun das Design der Webseite geändert und im Zuge dessen die Standard-Überschriftfarbe auf Rot geändert, muss diese Änderung nurmehr an einer einzigen Stelle vorgenommen werden, nämlich im Style Sheet. Automatisch ändern sich dann die Darstellung aller **h1**-Überschriften im gesamten HTML-Dokument. Ein damit verbundener Vorteil liegt natürlich auf der Hand. Dieselbe CSS-Datei kann für viele unterschiedliche Webseiten verwendet werden, eine große Site mit hunderten Unterseiten kann dadurch sehr einfach ihr Design an neue Bedürfnisse anpassen.

Manchmal möchte aber nicht *alle* Überschriften vom Typ **h1** auf eine festgelegte Art formatieren, sondern nur bestimmte. Dazu gibt es das **class**-Attribut, das bei praktisch allen HTML-Elementen verwendet werden kann und das man oft sieht, wenn man den Quelltext einer Webseite betrachtet. Eine Überschrift könnte beispielsweise so ausgezeichnet sein: **<h1 class="rubrik">News</h1>**. Auf diese Attribut-Ausprägung kann nun im CSS-Code Bezug genommen werden. Damit gelten die im CSS hinterlegten Formatierungsvorschriften nur für diese Klasse von **h1**-Elementen, nicht aber für andere **h1**-Elemente. Im CSS könnte zur Formatierung der **h1**-Überschriftselemente folgender Code stehen:

```
h1.rubrik {
    color: red;
}
```

```
h1 {  
    color: blue;  
    font-weight: bold;  
}
```

Hier wird auch deutlich, was der Begriffsbestandteil *cascading* bedeutet. Eigenschaftsausprägungen werden *vererbt*. Die allgemeine **h1**-Formatierungsanweisung im CSS besagt, dass die Textfarbe Blau sein soll. Davon weicht die Formatierung für **h1**-Elemente der Klasse **rubrik** ab. Für sie gilt eine andere Anweisung, nämlich die Einfärbung in Rot. Die Anweisung **font-weight: bold** aus dem CSS des **h1**-Elements dagegen wird nicht durch die speziellen Anweisungen für **h1**-Elemente der Klasse **rubrik** überschrieben, sie gilt deshalb auch dort. Ein **h1**-Element der Klasse **rubrik** wird also, wie alle **h1**-Elemente, ebenfalls in Fettsatz dargestellt.

Der CSS-Standard wird ebenso wie der HTML-Standard vom World Wide Web Consortium verwaltet und weiterentwickelt.

Um ein JavaScript-Programm zum Laufen zu bringen, müssen wir es in eine Webseite einbinden. Das geschieht mit dem **script**-Element, dessen Verwendung Sie im folgenden Beispiel sehen:

```
<!DOCTYPE html>  
<html>  
  
    <head>  
        <title>Test-Seite</title>  
    </head>  
  
    <body>  
        <script type="text/JavaScript">  
            console.log("Das hier wird auf der " +  
                        "Konsole ausgegeben!")  
        </script>  
    </body>  
  
<html>
```

Der Name der Skript-Datei wird als Wert für das Attribut **src** (*source* = Quelle) angegeben.

Im Beispiel haben wir das **script**-Element im **body** der Webseite platziert, wir hätten es aber ebenso im **head** unterbringen können. Allerdings bindet man Skripte meist in den **body** ein, genauer: am Ende des **body**. Das hat einen guten Grund: Denn Skripte werden im HTML-Dokument da ausgeführt, wo sie stehen. Steht Ihr Skript am Ende der Webseite und braucht etwas länger, bis es vollständig ausgeführt ist – kein Problem. Der Rest der Webseite ist zu diesem Zeitpunkt ja bereits geladen und angezeigt. Unschöner wäre es, wenn der Betrachter vor einer leeren Seite warten müsste, während der JavaScript-Code läuft, der am Anfang des HTML-Dokuments eingebunden worden ist und deshalb auch als erstes ausgeführt wird, noch bevor irgendwelche anderen Elemente der Seite geladen werden.

Unseren JavaScript-Programmcode haben wir hier als *Extra-Datei* eingebunden. Das ist auch das übliche Vorgehen. Sie können allerdings (insbesondere kurze) Skripte auch *direkt* in die HTML-Datei einbinden. Das würde dann so aussehen:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Test-Seite</title>
    </head>
    <body>
        <script type="text/JavaScript">
            console.log("Das hier wird auf der Konsole
ausgegeben!")
        </script>
    </body>
<html>
```

Mit dem Attribut **type** des **script**-Elements teilen wir der HTML-Engine des Browsers mit, dass der Code zwischen dem öffnenden und schließenden Tag (in unserem Beispiel nur eine einzige Zeile) als JavaScript-Code verstanden werden soll. De facto ist JavaScript die einzige verwendete Skript-Sprache. Früher hätte man hier allerdings das ein oder andere Mal noch Microsofts VBScript antreffen können, dessen Verwendung in der Praxis freilich recht beschränkt war, weil es von Haus aus nur von Microsofts eigenen Browsern unterstützt wurde und mit JavaScript einen mächtigen Konkurrenten hatte.

**29**

## 29.1.2 Sicherheitsaspekte

---

### ■ Ausführung von Skripten im Browser aktivieren

Zwar benutzen viele (wahrscheinlich sogar die meisten), zumindest der bekannteren Websites in der einen oder anderen Form JavaScript und funktionieren nicht oder nicht richtig, wenn die Ausführung von JavaScript-Code deaktiviert ist. Allerdings gestatten alle modernen Browser, die Ausführung von Skript-Code zu unterbinden. Werfen Sie also einmal einen Blick in die Einstellungen Ihres Browsers und sehen Sie nach, ob dem Genuss Ihres erstes JavaScript-Programms irgendetwas im Wege stehen könnte.

Wenn die Funktionsfähigkeit einer Website an der Ausführbarkeit von JavaScript hängt, möchte man den Benutzer der Seite natürlich darauf hinweisen, dass er JavaScript aktivieren soll. Das lässt sich im HTML-Quelltext der Seite ganz einfach bewerkstelligen, indem man das **noscript**-Element benutzt und ihm als Hinweis eine Nachricht an den Benutzer mitgibt, wie im folgenden Beispiel:

```
<noscript>
    Bitte aktivieren Sie JavaScript, um diese Webseite
    vollumfänglich benutzen zu können.
</noscript>
```

Die Botschaft wird nur angezeigt, wenn die Ausführung von JavaScript tatsächlich in den Sicherheitseinstellungen des Browsers abgeschaltet ist (probieren Sie es aus!).

### ■ Sichtbarkeit des Quellcodes

Anders als PHP werden JavaScript-Programme vom Browser (also client-seitig) interpretiert. Dazu müssen sie zunächst heruntergeladen werden, was im Falle von

PHP-Code nicht notwendig ist, da dessen Interpreter vollständig auf dem Server operiert und den Client für die Ausführung nicht in Anspruch nimmt. Sie können Ihren Quellcode daher nicht vor den Augen des Benutzers verstecken, selbst dann nicht, wenn Sie ihn in eine Extra-Datei auslagern, die sie mit `<script src=...>` von einem Webserver laden. Wie Sie im folgenden Abschnitt sehen werden, ist es im Regelfall ein leichtes, den Code einzusehen. Es gibt zwar Methoden und Tools zur sogenannten *Obfuscation* („Verschleierung“), mit denen der Quelltext praktisch codiert werden kann. Aber selbst diesen Ansätzen sind Grenzen gesetzt, da der Browser den Quelltext ja letztlich wieder decodieren muss, um ihn ausführen zu können. Rechnen Sie also damit, dass Ihre JavaScript-Programme durch die Betrachter Ihrer Seite eingesehen werden können. Das mag zwar faktisch auf Ottonormal-Benutzer nicht zutreffen, aber diejenigen, die über die Kenntnisse verfügen, den JavaScript-Code zu verstehen, werden auch wissen, wie sie an ihn herankommen.

## 29.2 „Hallo Welt“ in JavaScript

### 29.2.1 Do-it-yourself: Der (gar nicht so) mühsame Weg

Das klassische „Hallo Welt“-Programm in JavaScript ist denkbar einfach:

```
console.log('Hallo Welt!')
```

Wenn Sie diese Zeile in einer JavaScript-Datei **hallowelt.js** speichern und dieses Skript dann wie im Beispiel des vorangegangenen Abschnitts gesehen, in ein ansonsten (bis auf die obligatorischen Elemente) leeres HTML-Webseiten-Dokument einbinden, haben Sie diese Aufgabe bereits erfolgreich hinter sich gebracht.

Wenn Sie nun das fertige HTML-Dokument im Browser öffnen, sehen Sie... nichts!

Hat unser einfaches Programm nicht funktioniert? Doch. Es gibt den Text nur einfach nicht im Browser-Fenster aus, sondern in einem anderen Bereich, der so genannten *JavaScript-Konsole*. Wie Sie an die Konsole, die standardmäßig nicht angezeigt wird und eher ein Werkzeug für Entwickler ist, herankommen unterscheidet sich von Browser zu Browser: Unter Windows drücken Sie in Google *Chrome* beispielsweise `<CTRL>+<SHIFT>+<I>`, in Microsoft Edge `<F12>` und in Mozilla *Firefox* `<CTRL>+<SHIFT>+<K>` (gilt jeweils in den deutschsprachigen Versionen der Browser). Auch über die Menüs können Sie die Konsole natürlich öffnen, häufig unter Menüpunkten wie „Weitere Tools“ oder „Entwicklertools“. Wenn Sie mehr als einen Browser installiert haben und in allen Browsern einmal die Konsole öffnen, werden Sie gewisse Ähnlichkeiten sofort erkennen.

Die ersten beiden Reiter/Tabs, mit denen wir hier vor allem arbeiten werden, haben meist Bezeichnungen wie „Elemente“, „Inspektor“ o. ä. und eben „Konsole“. Unter „Elemente“ sehen sie den HTML-Code der aktuell geladenen Seite und können in diesem HTML-Code interaktiv navigieren. Indem Sie ein Element im HTML-Quelltext markieren, wird in der Webseitenansicht automatisch die Darstellung des Elements farblich hervorgehoben. So können Sie leicht nachvoll-

ziehen, welcher Teil des Quelltextes zu welchem Teil der Webseitendarstellung führt. Durch Auf- und Zuklappen der Elemente im Quelltext können Sie zudem den Quelltext kompakter darstellen und die hierarchische Verschachtelung der HTML-Elemente besser nachvollziehen. Bei unserer kleinen Beispelseite mag das noch nicht sonderlich eindrucksvoll sein (► Abb. 29.1), aber öffnen Sie einmal die Elemente-Ansicht, wenn Sie eine andere Website offen haben, etwa Wikipedia oder YouTube! Dann erkennen Sie sehr schön, wie praktisch diese Funktion ist. Umgekehrt können Sie aus „Elemente“ heraus in einen Auswahlmodus wechseln, bei dem Sie ein Element der Webseiten-Ansicht des Browsers anklicken und dann in der „Elemente“-Ansicht die entsprechende Stelle im Quellcode der Seite automatisch hervorgehoben wird. Mehr zu dieser nützlichen Funktion in ► Abschn. 32.6.2.

Auch die JavaScript-Programme, die in andere Dateien ausgelagert sind, lassen sich auf diese Weise betrachten. Klicken Sie im HTML-Code auf einen Verweis auf eine ausgelagerte JavaScript-Datei, öffnet sich der Quelltext sofort in einem anderen Tab.

Übrigens: Sie können den HTML-Quelltext über die Elemente-Ansicht sogar bearbeiten, meist durch entsprechende Optionen im Kontextmenü. Die Änderungen können Sie regelmäßig mit der Tastenkombination <CTRL>+<S> als HTML-Datei speichern.

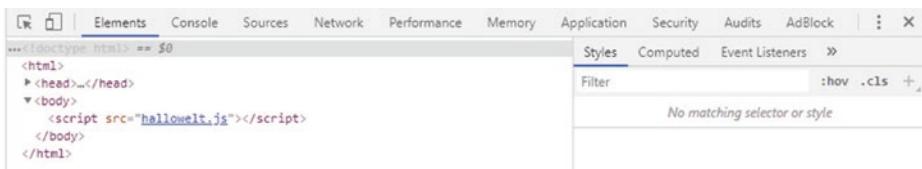
Uns interessiert hier aber für unser „Hallo Welt“-JavaScript-Programm vor allem der zweite Reiter, die Konsole. Öffnen Sie diese, dann sehen Sie sofort, dass unser JavaScript-Code tatsächlich einen Output produziert hat (► Abb. 29.2).

Über die Konsole können wir also Outputs ausgeben, die der Betrachter der Webseite nicht unmittelbar sieht. Das ist vor allem in der Entwicklung nützlich, um Debug-Meldungen ausgeben zu lassen, die die eigentliche Darstellung der Webseite auf diese Weise nicht beeinträchtigen. Aber auch der JavaScript-Interpreter des Browsers nutzt die Konsole, um Fehler- oder Warnmeldungen auszugeben.

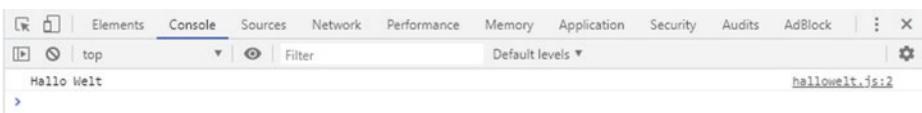
Im Beispiel von Abbildung ► Abb. 29.3 wurde in der Quelltextzeile `console.log("Hallo Welt!")` das erste Anführungszeichen vergessen. Der Interpreter weist uns beim Ausführen des Codes auf den Syntaxfehler hin.

Die Konsole können Sie auch in einem interaktiven Modus verwenden, also Anweisungen direkt in die Konsole eingeben und ausführen. Anders allerdings als in Python hat die Konsole bei JavaScript keinen von der Webseite getrennten Namensraum. Füh-

## 29



► Abb. 29.1 Entwickler-Werkzeuge in Google Chrome



► Abb. 29.2 Output des "Hallo Welt"-Programms in der Konsole

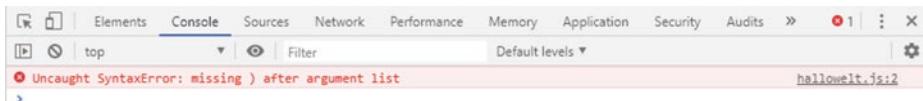


Abb. 29.3 Fehlermeldung in der JavaScript-Konsole von Google Chrome

ren Sie also ein Programm aus und legen dabei Variablen an, stehen diese Variablen im Anschluss in der Konsole auch zur weiteren interaktiven Bearbeitung zur Verfügung.

Die meisten Webbrower unterstützen in der Konsole auch eine Code-Vervollständigung. Wenn Sie zur tippen anfangen, bekommen Sie in einem Popup-Menü sogleich einige Auswahlmöglichkeiten präsentiert, mit deren Hilfe Sie die Tipparbeit abkürzen können.

Wir werden im nächsten Kapitel recht intensiv mit der Konsole arbeiten, weil sie eine einfache Möglichkeit bietet, Informationen auszugeben und im nächsten Kapitel eben nicht Ein- und Ausgaben im Vordergrund stehen, sondern andere Konzepte, denen wir unsere uneingeschränkte Aufmerksamkeit widmen wollen. Im Kapitel ► Kap. 31.7, wo wir uns dann gezielt mit der Ein- und Ausgabe von Daten befassen, werden wir natürlich auch sehen, wie Sie Daten direkt in die Webseite hineinschreiben können.

Wenn Sie nicht warten können, und glauben, ein „Hallo Welt“-Programm, das nicht direkt in die Webseite schreibt, ist kein richtiges „Hallo Welt“-Programm, dann probieren Sie es einmal mit diesem Code:

```
document.write('<p>Hallo Welt</p>');
```

Nach dem Aktualisieren der Ansicht (typischerweise mit F5) sehen Sie tatsächlich den Output im Browser, und zwar nicht in der Konsole, sondern in der eigentlichen Seitenansicht.

## 29.2.2 Mit etwas Nachhilfe: Die Schnelle Umsetzung mit einem Webdienst

Einfach direkt loszuprogrammieren ist in JavaScript gar nicht so einfach, denn zunächst einmal müssen Sie eine Webseite anlegen und Ihr Skript in diese einbinden.

Einfacher geht es mit Webdiensten wie dem Open-Source-Projekt *JS Bin* (► <http://www.jsbin.com>), die es Ihnen erlauben, ohne Umschweife nach Belieben loszulegen. *JS Bin* führt den JavaScript-Code, den Sie in den entsprechenden Fensterbereich eingeben aus, am Ende des HTML-**body** aus (ohne allerdings im HTML-Code einen Verweis auf Ihr Skript anzusegnen). Den HTML-Code können Sie auch manuell bearbeiten. Über die Toggle-Buttons am oberen Bildschirmrand können Sie bestimmen, welche Ansichten dargestellt werden sollen; *Output* ist dabei die fertige Website (► Abb. 29.4).

Tipp: Wenn Sie mit *JS Bin* arbeiten, empfiehlt es sich, das Häkchen *Auto-run JS* am rechten Bildschirmrand abzuwählen und die Ausführung des JavaScripts statt dessen stets über den Button *Run with JS* manuell zu starten. Ansonsten nämlich

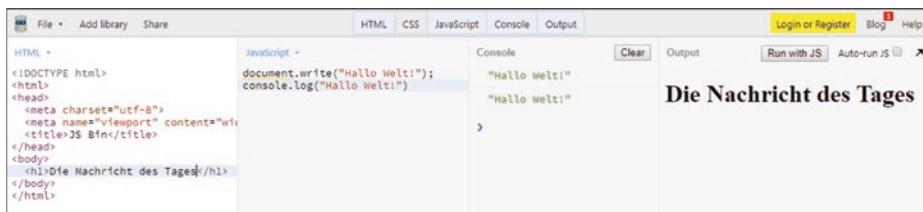


Abb. 29.4 Der JavaScript-Webdienst JS Bin

führt *JS Bin* den aktuellen Stand Ihres JavaScript-Codes bei jedem Tastenanschlag direkt wieder neu aus, was natürlich zu einer unerquicklich langen Liste unnützener Fehlermeldungen in der Konsole führt.

Neben *JS Bin* gibt es noch eine ganze Reihe weiterer Services, die ähnliche Funktionalitäten anbieten, etwa *Plunker* (► <http://www.plnkr.co>) oder *JS.do* (► <http://www.js.do>), wobei der Aufbau jeweils etwas unterschiedlich ist. Allen diesen Diensten ist gemeinsam, dass sie zum Ausprobieren zweifelsfrei gut geeignet sind. Will man sich allerdings ernsthaft mit JavaScript befassen, empfiehlt es sich aber schon, offline mit eigener „Infrastruktur“ (HTML- und JavaScript-Code-Daten) zu arbeiten. Und genau das werden wir im Folgenden auch tun.

### 29.3 Zusammenfassung

## 29

In diesem Kapitel haben wir uns damit beschäftigt, wie Sie ein JavaScript-Programm zum Laufen bringen.

Folgende Punkte sollten Sie aus diesem Kapitel unbedingt mitnehmen:

- JavaScript-Programme laufen typischerweise in Webseiten, müssen also in diese eingebunden werden.
- Das geschieht mit Hilfe des HTML-Elements **script**; diesem kann mit dem Attribut **src** entweder ein Verweis auf eine Skript-Datei mit JavaScript mitgegeben werden, oder aber der JavaScript-Code wird zwischen den **<script>...</script>** Tags direkt in die Webseite eingebettet.
- Es ist gute Praxis, Skripte *am Ende* der Webseite einzubinden, sodass zunächst die Webseite geladen wird und bereits betrachtet werden kann, während ggf. das Skript noch geladen bzw. ausgeführt wird.
- Mit dem HTML- Element **noscript** kann ein Hinweistext angezeigt werden, wenn der Benutzer die Ausführung von JavaScript in seinem Browser ausgeschaltet hat (normalerweise ist JavaScript dank entsprechender Browser-Standardeinstellungen bereits aktiviert).
- JavaScript-Code ist für den Betrachter der Webseite grundsätzlich einsehbar. Sie können Ihren Code also nicht „verstecken“.

Webservices wie *JS Bin* oder *Plunker* erlauben es, JavaScript auszuprobieren, ohne sich um die Einbindung in eine Webseite kümmern zu müssen; das ist praktisch, um schnell etwas zu testen, wer aber ernsthaft mit JavaScript arbeiten will, kommt nicht umhin, sich damit auseinanderzusetzen, wie man Skripte in Webseiten einbindet (was ja nun auch wahrlich nicht kompliziert ist).

# Wie stelle ich sicher, dass ich (und andere) mein Programm später noch verstehe?

## Inhaltsverzeichnis

- 30.1 Gestaltung des Programm-Codes und Namenskonventionen – 452
- 30.2 Kommentare – 454
- 30.3 Zusammenfassung – 455

## Übersicht

Bevor wir nun richtig in die Programmierung mit JavaScript einsteigen, wollen wir uns vorbereitend noch mit einigen wichtigen Fragen der Gestaltung des Programm-Codes und – Sie werden es schon geahnt haben – der Kommentierung des Codes beschäftigen.

In diesem Kapitel werden Sie lernen:

- wie Bezeichner für Variablen, Funktionen und andere Objekte aufgebaut werden
- wie Anweisungen abgeschlossen werden
- welche Style Guides für die Gestaltung Ihres JavaScript-Codes Ihnen zur Orientierung zur Verfügung stehen
- welche Arten von Kommentaren Sie in JavaScript verwenden können.

### 30.1 Gestaltung des Programm-Codes und Namenskonventionen

#### ■ Bezeichner

JavaScript ist *case-sensitive*, unterscheidet also zwischen Groß- und Kleinschreibung. Bezeichner von zum Beispiel Variablen, Objekten und Funktionen dürfen Buchstaben, Ziffern, Unterstriche (\_) und Dollar-Zeichen (\$) enthalten, dürfen aber nicht mit einer Ziffer beginnen. Die Bezeichner sind in JavaScript nach dem Unicode-Schema codiert, was dazu führt, dass auch Umlaute und der Buchstabe ß in Bezeichnern zulässig sind. Da diese Zeichen allerdings in den meisten anderen natürlichen Sprachen nicht vorkommen, ist zu empfehlen, darauf zu verzichten. Würden Sie einmal einen Beispiel-Code in ein Internet-Forum posten, um sich nach Hilfe zu erkundigen, hätte das internationale Publikum unter Umständen Probleme, diese Zeichen zu reproduzieren, wenn es Ihren Code bearbeiten möchte. Denken Sie daran, wie die Situation umgekehrt wäre, wenn ein Isländer eine Variable **húsið1** (also **haus1**) nennen würde, und Sie diesen Code bearbeiten wollten.

Der erste Begriff in einem Bezeichner beginnt in JavaScript oft mit einem Kleinbuchstaben, die folgenden Begriffe dann mit einem Großbuchstaben (auch dann, wenn es sich um Begriffe handelt, die in der natürlichen Sprache eigentlich klein geschrieben werden). Ein typischer JavaScript-Bezeichner wäre zum Beispiel **istKundeGemahnt** oder **starteSpeicherung()**. Diese Art der Schreibweise, die man wegen der „Höcker“ in den Bezeichnern auch *camel case* nennt, mag am Anfang ungewöhnlich sein, lässt Ihr Programm in den Augen von JavaScript-Programmierern eindeutig „javascriptig“ aussehen.

#### ■ Anweisungsende und -umbrüche

Anweisungen können in JavaScript mit einem Semikolon abgeschlossen werden. Sollen mehrere Anweisungen auf einer Zeile stehen, müssen sie durch Semikolons voneinander getrennt sein.

Anders als in manchen anderen Sprachen endet aber nicht jede Anweisung in JavaScript automatisch am Zeilenende. Stattdessen prüft JavaScript ohne

Semikolon am Zeilenende, ob die Anweisung Sinn ergäbe, wenn man sie in der folgenden Zeile fortsetzen würde. Dieses seltsame Feature kann in Einzelfällen zu unerwartetem und scheinbar unerklärlichem Verhalten führen. Deshalb werden wir hier Anweisungen grundsätzlich mit einem Semikolon abschließen – wohlwissend, dass einige Style Guidelines für JavaScript etwas anderes empfehlen. Dass wir Anweisungen mit Semikolon abschließen, gilt nicht für solche Anweisungen, die wir in der JavaScript-Konsole ausführen, wo das Anweisungsende durch das Drücken der <RETURN> oder <ENTER>-Taste klar markiert ist.

Anweisungen können umgebrochen, das heißt, auf der folgenden Zeile fortgesetzt werden. Dabei sind allerdings Zeilenumbrüche *innerhalb von Zeichenketten* unzulässig.

Syntaktisch korrekt ist also beispielsweise folgende umgebrochene Ausgabe-Anweisung für die JavaScript-Konsole:

```
> console.log('Hallo',
  'Welt')
```

Ebenso zulässig wäre folgender Umbruch:

```
> console.log('Hallo'
, 'Welt')
```

Da sich Zeichenketten in JavaScript auch mit dem Plus-Operator verbinden lassen, könnten Sie auch schreiben:

```
> console.log('Hallo'
+ 'Welt')
```

Nicht jedoch zulässig wäre die nächste Ausgabeanweisung, weil sie einen Umbruch innerhalb einer Zeichenkette enthält:

```
> console.log('Hallo
Welt')
```

Möchten Sie unbedingt einen Umbruch innerhalb der Zeichenkette erreichen, können Sie diese mit dem Backslash („\\“) erzeugen, der dabei selbst nicht mit ausgegeben wird:

```
> console.log('Hallo \
Welt')
```

Wenn Sie umbrechen, empfiehlt es sich, die zweite Zeile entsprechend einzurücken. Im Buch sind Programmcode häufig umgebrochen, damit er in die Seitenbreite passt.

Die Programmcodes, die Sie zum Buch downloaden können, unterliegen dieser Beschränkung natürlich nicht und sind deshalb mit weniger Umbrüchen versehen.

### ■ Allgemeiner Code-Stil

Wie für die meisten anderen Programmiersprachen auch, haben sich kluge Entwickler Gedanken darüber gemacht, wie man seinen JavaScript-Code so formatieren sollte, dass er übersichtlich und lesbar ist. Ergebnis sind die Style Guides, von denen es für JavaScript verschiedene gibt, die durchaus zu ein- und demselben Thema ganz unterschiedliche Empfehlungen geben können, etwa zur Frage des Einsatzes von Semikolons beim Abschließen von Anweisungen.

Bekannte JavaScript-Style-Guides sind der *Google JavaScript Style Guide* (► <https://google.github.io/styleguide/jsguide.html>), der von Airbnb (► <https://GitHub.com/airbnb/javascript>) und der *JavaScript Standard Style Guide* (► <https://GitHub.com/standard/standard>), der – anders als sein Name vermuten lässt – keineswegs eine offizielle Rolle einnimmt, wie es zum Beispiel bei Python das *Python Enhancement Proposal (PEP)* Nummer 8 tut.

Für einige Style Guides, darunter den Standard Style Guide, gibt es sogar Software, einen sogenannten *Linter*, der in der Lage ist, den Quellcode vollautomatisch so zu formatieren, dass er den Vorgaben des Style Guides entspricht.

Wir werden uns in diesem Teil des Buches an keinem dieser Style Guides vollumfänglich orientieren. Außer in einem berufsmäßigen Umfeld, wo Sie mit einer Reihe anderer Entwickler zusammenarbeiten, ist eine hundertprozentige Umsetzung eines Style Guides sicherlich auch nicht vonnöten. Es empfiehlt sich dennoch, einige der Style Guides einmal durchzublättern und sich dabei vielleicht das eine oder andere abzuschauen.

## 30.2 Kommentare

JavaScript kennt sowohl ein- als auch mehrzeilige Kommentare. Einzelige Kommentare werden durch *//* eingeleitet und weisen alles, was rechts davon steht, als Kommentar aus, der vom Interpreter nicht verarbeitet wird. Mehrzeilige Kommentare beginnen mit */\** und werden mit *\*/* abgeschlossen (die Sterne sind also immer dem Kommentartext zugewandt).

Beispiele für Kommentare könnten damit so aussehen:

```
/*
Hier folgt ein mehrzeiliger Kommentar, der erklärt,
warum die Zuweisung a = 5 hier notwendig ist
*/
a = 5;
// Diese Zuweisung lässt sich in einem einzeiligen
// Kommentar erläutern
b = 10;
c = 7; // Auch c muss mit einem sinnvollen Wert
        // initialisiert werden
```

Der letzte Kommentar ist ein *Inline-Kommentar*, auch er ist syntaktisch zulässig, aber nicht gerne gesehen, weil schwer zu lesen. Wenn Sie *Inline-Kommentare* verwenden, sollten Sie sicherstellen, dass Sie zwischen dem Ende der Anweisung und dem Kommentarsymbol etwas Platz (mindestens zwei Leerzeichen) lassen, damit sich der Kommentar optisch vom ausführbaren Code abhebt.

### 30.3 Zusammenfassung

---

In diesem Kapitel haben wir uns mit Grundfragen der Gestaltung des JavaScript-Codes sowie der Kommentierung beschäftigt.

Folgende Punkte sollten Sie aus diesem Kapitel unbedingt mitnehmen:

- Bezeichner in JavaScript können Buchstaben, Ziffern, den Unterstrich sowie das Dollar-Zeichen enthalten, aber nicht mit einer Ziffer beginnen.
- Bezeichner sind UTF-8-codiert und dürfen daher auch Umlaute enthalten; von deren Verwendung ist jedoch abzuraten.
- Typischerweise werden Bezeichner klein geschrieben. Bei aus mehreren Begriffen zusammengesetzten Bezeichnern werden die folgenden Begriffe jeweils großgeschrieben.
- Semikolons am Anweisungsende sind zwar nicht zwingend vonnöten, aber zu empfehlen, da JavaScript nicht immer automatisch am Zeilenende mit der Interpretation einer Anweisung stoppt.
- Zeilenumbrüche im Code sind zulässig, solange sie nicht innerhalb einer Zeichenkette erfolgen.
- Zur Gestaltung des Codes, bezüglicher derer man in JavaScript verhältnismäßig frei ist, gibt es zahlreiche Guidelines, die durchaus unterschiedliche Ansichten zu verschiedenen Themen vertreten, darunter den *Google JavaScript Style Guide*, den *JavaScript Standard Style Guide* und den Style Guide von Airbnb.
- JavaScript kennt ein- und mehrzeilige Kommentare.
- Einzelige Kommentare beginnen mit *//* und markieren alles, was rechts davon steht, also Kommentar; sie können auch innerhalb einer Code-Zeile („inline“) eingesetzt werden.
- Mehrzeilige Kommentare stehen zwischen */\** und *\*/*.



# Wie speichere ich Daten, um mit ihnen zu arbeiten?

## Inhaltsverzeichnis

- 31.1 Deklaration von Variablen – 459**
- 31.2 Elementare Datentypen – 460**
  - 31.2.1 Zahlen (number) – 460
  - 31.2.2 Zeichenketten (string) – 462
  - 31.2.3 Wahrheitswerte (boolean) – 467
  - 31.2.4 Spezielle Typen und Werte (null, undefined, NaN) – 467
- 31.3 Konvertieren von Variablen – 469**
  - 31.3.1 Implizite Konvertierung – 469
  - 31.3.2 Explizite Konvertierung – 470
- 31.4 Arrays – 472**
- 31.5 Objekte – 480**
  - 31.5.1 Objektorientierung in JavaScript – 480
  - 31.5.2 Objekte direkt erzeugen – 481
  - 31.5.3 Auf Eigenschaften von Objekten zugreifen – 482

- 31.5.4 Objekte mit Hilfe des Object-Konstruktors erzeugen – 483
  - 31.5.5 Objekte mit Hilfe von Konstruktor-Funktionen erzeugen – 484
  - 31.5.6 JSON – 485
- 31.6 Lösungen zu den Aufgaben – 486**
- 31.7 Zusammenfassung – 489**

## Übersicht

Nun ist es an der Zeit, dass wir uns damit beschäftigen, wie wir in JavaScript mit Variablen und Objekten arbeiten, um Daten zur Verarbeitung temporär speichern zu können. Darum geht es in diesem Kapitel.

In diesem Kapitel werden Sie lernen:

- welche die elementaren Datentypen von JavaScript sind
- wie Sie mit elementaren Datentypen arbeiten (sie erzeugen, zuweisen, konvertieren und vieles mehr)
- wie Sie Felder (Arrays) aus beliebigen Datentypen erschaffen und mit ihnen arbeiten können,
- was die elementaren Datentypen von echten JavaScript-Objekten unterscheidet
- inwiefern Objektorientierung in JavaScript anders funktioniert als in anderen objektorientierten Sprachen
- wie Objekte in JavaScript erzeugt und bearbeitet werden
- was JSON ist, warum es so wichtig ist, und wie Sie JavaScript-Objekte in JSON verwandeln können, und umgekehrt.

### 31.1 Deklaration von Variablen

Variablen *sollten* in JavaScript deklariert, also vor der ersten Verwendung angemeldet werden. Es ist gute Praxis, Variablen zu deklarieren, obwohl eine einfache Wertzuweisung regelmäßig genügt, um eine Variable zu erzeugen und eine formale Deklaration strenggenommen nicht erforderlich ist. Eine Fehlermeldung erhält man allerdings, wenn man auf eine Variable zuzugreifen versucht, die weder deklariert noch durch Zuweisung zuvor mit einem Wert versehen wurde.

Ein häufiges Problem, wenn Variablen-deklarationen nicht zwingend erforderlich sind, ist, dass man im Programm durch Vertippen aus Versehen eine neue Variable erzeugt. Will man etwa einer existierenden Variablen **betragRechnung** einen Wert zuweisen, schreibt aber in der Zuweisung unbeabsichtigt **betragRecnung**, so bekommt man spätestens dann ein Problem, wenn man später mit der ursprünglichen Variable **betragRechnung** weiterarbeitet, die aber den vermeintlich ihr zugewiesenen Wert nie erhalten hat. Stattdessen hat die neu geschaffene Variable **betragRecnung** den Wert „abbekommen“.

Man kann solcherlei Schwierigkeiten vermeiden, indem man JavaScript im sogenannten *strikten Modus* betreibt; dann nämlich führen undeklärte Variablen zu einer Fehlermeldung. Der strikte Modus lässt sich für ein ganzes Skript (oder aber eine einzelne Funktion) einschalten, indem man eine spezielle Anweisung als erste Anweisung in das Skript (oder die Funktion) einführt:

```
'use strict';
```

Probieren Sie es aus, und weisen Sie in einem solchen Skript einen Wert einer Variablen zu, die Sie zuvor nicht deklariert haben; Sie erhalten sofort eine Fehlermeldung.

dung. Jetzt entfernen Sie die `'use strict'`-Anweisung und führen das Skript erneut aus – kein Problem, die Variable wird einfach bei der ersten Zuweisung erzeugt.

Die formale Deklaration von Variablen geschieht in JavaScript mit Hilfe des Schlüsselwortes `var`, auf das ein oder – durch Kommata separiert – mehrere Variablen-Bezeichner folgen. Die Variablen können, müssen aber nicht bei der Deklaration bereits mit einem Wert initialisiert werden.

```
var x = 0.5, y, z = 'Eine Nachricht';
```

Es fällt auf, dass die Variablen bei der Deklaration ohne Typangabe erzeugt werden. JavaScript erkennt anhand des bei der Deklaration oder auch später zugewiesenen Werts automatisch, um welchen Typ es sich handeln muss (schwache Typisierung). Natürlich können wir, und das ist manchmal sogar zwingend notwendig, die Typen von Variablen durch Konvertierung explizit festlegen. Damit beschäftigen wir uns in Abschnitt ▶ Abschn. 31.3. Der Typ einer Variablen kann auch geändert werden, indem der Variablen einfach ein Wert eines anderen Typs zugewiesen wird. JavaScript passt dann den Typ der Variablen automatisch so an, dass sie den neuen Wert aufnehmen kann (dynamische Typisierung).

Das Schlüsselwort `var` kann mehrfach im Programm auftreten und muss nicht zwingend am Anfang des gesamten Programmquelltextes stehen. Es ist allerdings gute Praxis, Variablen-deklarationen am Anfang zusammenzuführen, um den Überblick über die angemeldeten Variablen zu behalten.

## 31.2 Elementare Datentypen

### 31

JavaScript besitzt eine Reihe elementarer Datentypen, darunter Zahlen (**number**), Zeichenketten (**string**) und Wahrheitswerte (**boolean**). Dieses (und einige spezielle weitere) Typen sind in dem Sinne *elementar*, dass sie selbst *keine Objekte* sind, sondern sogenannte *primitives*. Variablen aller übrigen Typen sind in JavaScript Objekte und besitzen daher auch Eigenschaften und Methoden, um sie zu bearbeiten. Die elementaren Datentypen tun das nicht, verhalten sich aber dennoch manchmal – wie wir noch sehen werden – auf beinahe wundersame Weise so, als wären sie „richtige“ Objekte.

### 31.2.1 Zahlen (**number**)

#### ■ Ganze Zahlen und Fließkommazahlen

**number** ist der Datentyp, der Zahlen aufnimmt, und zwar ganz unabhängig davon, ob es sich dabei um ganze Zahlen oder gebrochene, das heißt Fließkommazahlen, handelt. Zum Speichern einer Zahl reserviert JavaScript immer 64 Bit, also 8 Bytes. Der Wertebereich an Zahlen, den eine **number**-Variable aufnehmen kann, hängt deshalb davon ab, wie hoch die Genauigkeit sein soll, wieviel Speicher also für die Ablage der Nachkommastellen benötigt wird. Umgekehrt hängt die mögliche Ge-

nauigkeit von der Höhe des Ganzzahl-Anteil vor dem Komma ab: Ist dieser sehr groß, bleibt weniger Speicherplatz für die Nachkommastellen.

Das Dezimaltrennzeichen ist in JavaScript, wie in praktisch allen Programmiersprachen, der Punkt.

### ■ **Infinity**

Ein besonderer Wert, den eine **number**-Variable annehmen kann, ist **Infinity**, also unendlich. Teilen Sie in JavaScript eine Zahl durch den Wert **0**, so erhalten Sie – anders als in vielen anderen Sprachen – keine Fehlermeldung, sondern den Wert **Infinity** bzw. **-Infinity**. Öffnen Sie die JavaScript-Konsole und geben Sie folgenden Code ein (das Größer-Zeichen stellt hier die Eingabeaufforderung der Konsole dar und darf deshalb nicht mit eingegeben werden):

```
> 1/0
Infinity

> Infinity+1
Infinity
```

### ■ **Operatoren**

Natürlich können Sie mit Zahlen die üblichen Rechenoperationen ausführen, darunter die vier Grundrechenarten. Darüber hinaus steht Ihnen mit **%** ein *Modulo*-Operator zur Verfügung, der den ganzzahligen Rest einer Division liefert. Alle diese Operatoren sind *binäre* Operatoren, die zwei Werte zu einem neuen Wert verarbeiten. Daneben existieren aber auch eine Reihe von *unären* Operatoren, die auf einen einzelnen Wert angewendet werden. Besonders interessant in diesem Zusammenhang sind die häufigen in JavaScript-Programm zu findenden *Inkrement*-**(++)** und der *Dekrement-Operatoren* **(--)**, die eine **number**-Variable um den Wert **1** erhöhen bzw. verringern und zwar unabhängig davon, ob die Variable einen von **0** verschiedenen Nachkommaanteil besitzt, oder nicht. So ist zum Beispiel **variable++** eine kompaktere Schreibweise für die Zuweisung **variable = variable + 1**.

### ■ **Methoden von number-Objekten**

**number**-Variablen besitzen (scheinbar) aufrufbare Methoden, wenngleich auch in überschaubarer Anzahl. „Scheinbar“ deshalb, weil eine **number**-Variable eigentlich ja ein *primitive*, also eine Variable von einem elementaren Datentyp und damit eben *kein* Objekt ist, bei dem wir das Vorhandensein von Eigenschaften und Methoden erwarten würden. In Abschnitt ► Abschn. 31.3.2 werden wir sehen, wie dieses scheinbar widersprüchliche Verhalten zustande kommt.

Unter den Methoden, die für **number** zur Verfügung stehen, ist hier beispielsweise **toExponential()** zu nennen, eine Methode, die die Zahl in eine Exponentialschreibweise umwandelt und als Zeichenkette zurückgibt. Die Variable selbst wird dabei nicht verändert.

```
> zahl = 50000
> wissenschaftlich = zahl.toExponential()
> wissenschaftlich
"5e+4"
> typeof zahl
"number"
> typeof wissenschaftlich
"string"
```

Hier sehen Sie zunächst, dass wir auf die Methoden (und Eigenschaften) von Objekten mit Hilfe des Punkt-Operators zugreifen können (genau wie in Python), und dass der Aufruf einer Methode immer die Angabe der Klammern erfordert, selbst dann, wenn der Methode gar keine Argumente übergeben werden.

Mit **typeof objektinstanz** lernen Sie gleich noch einen weiteren nützlichen Operator kennen, der den Typ einer Variablen als String zurückgibt. Wie Sie sehen, ist die **typeof()**-Methode scheinbar eine globale Methode, die an keinem Objekt hängt.

Eine weitere praktische Methode neben **toExponential()**, die auf **number** angewandt werden kann, ist **toFixed(stellen)**, die die Zahl auf die angegebene Anzahl von Nachkommastellen runden, und das Ergebnis wiederum als String zurückgibt:

```
> zahl = 3.14159
> zahl.toFixed(3)
"3.142"
```

Auch konstante Werte (sogenannte *Literale*) verhalten sich in JavaScript wie Objekte, dementsprechend also auch konstante Zahlwerte. Um auf die Methoden dieser Objekte zugreifen zu können, müssen Sie allerdings den Wert in runden Klammern schreiben, wie im folgenden Beispiel, das Sie leicht in der Konsole ausprobieren können:

```
> (3.14159).toFixed(3)
"3.142"
```

### 31.1 [3 min]

Zeigen Sie, dass in JavaScript unendlich plus eins immer noch unendlich ist.

## 31.2.2 Zeichenketten (string)

### ■ Zeichenketten zuweisen

Zeichenketten werden durch einfache oder doppelte Anführungszeichen abgegrenzt:

```
var nachricht = "Hallo Welt", nachricht2 = 'Nochmal hallo Welt';
console.log(nachricht, nachricht2);
```

Dadurch, dass in JavaScript, ähnlich wie in Python, zwei unterschiedliche Zeichenkettenbegrenzer zur Verfügung stehen, kann jeweils der eine von beiden zur Abgrenzung des Strings, der andere innerhalb des Strings für „Zitate“ verwendet werden:

```
var nachricht = "Daraufhin rief der Nerd laut: 'Hallo Welt'";
```

### ■ Escaping

Manchmal möchte man aber innerhalb eines Strings das Zeichen, das man benutzt, um Anfang und Ende der Zeichenkette zu markieren, als Bestandteil des Strings darstellen.

In diesem Fall muss vom *Escaping* gebraucht gemacht werden. Dabei wird, wie wir bereits an früherer Stelle gesehen haben, dem Zeichen, das innerhalb des Strings eine andere Funktion haben soll, als es sie üblicherweise hätte, ein Backslash (\) vorangestellt.

In unserem Beispiel könnten wir das doppelte Anführungszeichen folgendermaßen escapen:

```
var nachricht = "Daraufhin rief der Nerd laut: \"Hallo Welt\"";
```

Durch die Voranstellung des Backslashes wird das Anführungszeichen nicht mehr als das Zeichen interpretiert, das es in JavaScript-Programmen eigentlich ist, nämlich ein *Begrenzer* für Zeichenketten. Stattdessen wird es als *Bestandteil* der Zeichenkette betrachtet. Das letzte Anführungszeichen ist nicht escaped und markiert deshalb tatsächlich das Ende.

Das Escapen kann allerdings nicht nur dazu verwendet werden, die normale Steuerungsfunktion bestimmter Zeichen abzuschalten, um sie stattdessen als Bestandteil der Zeichenkette zu berücksichtigen. Escapen kann umgekehrt auch dazu dienen, Zeichen, die eigentlich im String ganz unauffällige Buchstaben wären, eine andere Funktion zu verleihen und sie dadurch in eine Steueranweisung verwandeln. Betrachten Sie das folgende Beispiel:

```
> console.log('Manche Dinge muss man \nmit etwas \t Abstand
  betrachten. ');
Manche Dinge muss man
mit etwas      Abstand betrachten.
```

Auch hier sehen wir zwei Backslashes, die zum Escapen verwendet werden: Einmal wird der Buchstabe **n** escaped; **\n** führt dazu, dass an dieser Stelle ein Zeilenumbruch in den String eingefügt wird. **\t** führt zum Einfügen eines Tabulatorssprungs.

In beiden Fällen wird ein Buchstabe, der sonst keine besondere Funktion hätte und ein normaler Bestandteil des Strings wäre, in eine Steueranweisung umgewandelt.

Was aber, wenn Sie mit einem Backslash ein Zeichen escapen, dass gar keine spezielle Steuerfunktion besitzt, zum Beispiel das **i**. Die gute Nachricht ist, es passt gar nichts:

```
> console.log("Escape das \i, bringt's aber nie.");
Escape das i, bringt's aber nie.
```

Der Backslash wird allerdings nicht dargestellt, er ist eben eine spezielle Steueranweisung, die dazu führt, dass das nächste Zeichen escaped wird.

Das bringt uns logischerweise zur nächsten Frage, nämlich, wie sich denn dann ein Backslash in einem String darstellen lässt. Immerhin verwenden insbesondere Programmierer, die unter Windows arbeiten, den Backslash als Trennzeichen bei Pfadangaben. Des Rätsels Lösung ist einfach: Der Backslash selbst wird einfach escaped:

```
> console.log('Die Datei liegt in C:\\Windows\\\\System.');
Die Datei liegt in C:\\Windows\\\\System.
```

### ■ Zugriff auf einzelne Zeichen

Auf die einzelnen Zeichen eines Strings kann in JavaScript zugegriffen werden wie auf die Elemente eines (Nur-Lese-)Arrays. Die Indizierung von Arrays in JavaScript beginnt mit dem Index **0** für das erste Zeichen. Der Index wird beim Zugriff in eckige Klammern gesetzt:

```
> var nachricht = 'Hallo Welt ';
> nachricht[1];
"a"
> typeof(nachricht[1]);
"string"
```

Wie sich leicht überprüfen lässt, ist ein Zeichen aus einem String wiederum ein String. Einen speziellen Datentyp für einzelne Zeichen gibt es in JavaScript nicht.

Das Array mit den Zeichen des Strings ist nur lesbar, kann also nicht beschrieben werden. Sie können daher *nicht* mit einer Anweisung wie **nachricht[2] = 'z'** ein Zeichen in der Zeichenkette ersetzen. Dazu müssen Sie mit den Methoden, die für Variablen des Typs **string** zur Verfügung stehen, arbeiten, was wir weiter unten tun werden.

Mit den Details der Arbeit mit Arrays werden wir uns in Abschnitt ▶ Abschn. 31.4 noch eingehender beschäftigen.

### ■ Strings miteinander verketten

Obwohl mit Strings nicht wie mit Zahlen gerechnet werden kann, unterstützen auch **string**-Variablen den Plus-Operator (**+**). Im Zusammenhang mit Strings wird er dazu verwendet, mehrere Strings mit einander zu verketten:

```
var nachricht = 'Hallo ' + ' ' + 'Welt';
console.log(nachricht);
```

Wie Sie sehen, spielt es dabei keine Rolle, ob einfache oder doppelte Anführungszeichen für die Begrenzung der Strings verwendet werden.

Anders als das Plus können die übrigen arithmetischen Operatoren auf Strings nicht angewendet werden. Tut man dies dennoch, so erhält man zwar keine Fehlermeldung, aber den Rückgabewert **NaN** (*not a number*), den wir uns in Abschnitt ► Abschn. 31.2.4 noch genauer anschauen werden. Er signalisiert, dass wir einen Operator, der für die Arbeit mit Zahlen ausgelegt ist, auf etwas angewendet haben, das keine Zahl ist (probieren Sie es aus!).

### ■ Methoden und Eigenschaften von string

Wenn Sie in der JavaScript-Konsole eine **string**-Variable anlegen und dann ihren Namen, gefolgt von einem Punkt-Operator eingeben, öffnet sich das Popup-Menü mit der Liste der Methoden und Eigenschaften, die für Strings zur Verfügung stehen.

Eine besonders wichtige Eigenschaft ist **length**, die Länge des Strings:

```
> var nachricht = "Hallo Welt";
> nachricht.length;
10
```

Wenn Sie also auf die einzelnen Zeichen des Strings zugreifen wollen, können Sie mit **nachricht[nachricht.length-1]** das letzte Zeichen des Strings greifen (weil die Indizierung bei 0 beginnt!).

Statt über das Array der Zeichen auf ein einzelnes Zeichen zuzugreifen, könnte man auch die die Methode **charAt(index)** des String verwenden:

```
> var nachricht = 'Hallo Welt ';
> nachricht.charAt(nachricht.length-1);
"t"
```

Neben **charAt()** bietet **String** noch eine Reihe weiterer praktischer Methoden::

- **indexOf(suchstring, abPosition)**: Sucht im String, für den die Methode aufgerufen wird den **suehstring**, und zwar frühestens ab Index **abPosition**; dabei ist **abPosition** ein optionales Argument, kann also auch weggelassen werden, was dazu führt, dass ab String-Anfang gesucht wird. Als Rückgabewert liefert **indexOf** den Index des Beginns von **suehstring** innerhalb der durchsuchten

Zeichenkette oder **-1**, falls **suchstring** nicht in der durchsuchten Zeichenkette gefunden wurde:

```
> nachricht.indexOf("el");
7
> nachricht.indexOf("tel");
-1
```

- **replace(suchen, ersetzenDurch)**: Durchsucht den String nach der Zeichenfolge **suchen** und ersetzt jedes Vorkommen durch die Zeichenkette **ersetzenDurch**:

```
> nachricht.replace("Hallo", "Moin");
"Moin Welt"
```

- **toUpperCase()**, **toLowerCase()**: Wandelt den String in Groß- bzw. Kleinbuchstaben um:

```
> nachricht.toUpperCase();
"HALLO WELT"
> nachricht.toLowerCase();
"hallo welt"
```

31

- **trim()**, **trimLeft**, **trimRight()**: Entfernt führende und abschließende, nur führende bzw. nur abschließende Leerzeichen aus einem String:

```
> nachricht = "    " + nachricht + "      "
"    Hallo Welt      "
> nachricht.trim();
"Hello Welt"
> nachricht.trimLeft();
"Hello Welt      "
> nachricht.trimRight();
"    Hallo Welt"
```

Beachten Sie bitte bei der Anwendung dieser Methoden, dass die Variable, für die sie aufgerufen werden, keine Veränderung erfährt, sondern lediglich ein neuer, veränderter String durch die Methoden erzeugt und zurückgegeben wird. Sie müssen also selbst dafür Sorge tragen, diesen Rückgabewert aufzufangen.

### 31.2.3 Wahrheitswerte (boolean)

Der dritte wichtige, elementare Datentyp in JavaScript sind Wahrheitswerte. Sie können nur die Ausprägungskonstanten **true** und **false** (Kleinschreibung beachten!) annehmen und werden intern mit den Werten **1** für **true** und **0** für **false** codiert. Das wird deutlich, wenn man mit den Konstanten rechnet:

```
> true * 5  
5  
> false - 1  
-1
```

### 31.2.4 Spezielle Typen und Werte (null, undefined, NaN)

#### ■ null

JavaScript kennt zwei spezielle Werte, die signalisieren können, dass eine Variable *keinen Wert* hat: **undefined** und **null**.

Besitzt eine Variable bewusst keinen echten Wert – zum Beispiel die Antwort auf eine Auswahlfrage in einem Fragebogen, die der Befragte bei der Beantwortung bewusst ausgelassen hat – wird der (Quasi-) „Wert“ **null** verwendet:

```
frage1Alter = null;
```

**null** ist also eine spezielle Wertkonstante, die anzeigt, dass eine Variable bewusst „leer“ ist, also derzeit keinen „echten“ Wert hält. Weist man einer Variablen, die zuvor einen „echten“ Wert besaß, **null** zu, so ändert sie ihren Objekt-Typ auf das allgemeine **object**:

```
> var zahl = 5;  
> typeof(zahl);  
"number"  
> zahl = null;  
> zahl;  
"object"
```

Lassen Sie sich dadurch nicht verwirren: **null** ist eine Variable von einem gleichnamigen elementaren Datentyp. Dass **typeof()** trotzdem **"object"** liefert, ist streng genommen falsch und röhrt von der historischen Implementierung der Funktion **typeof()** her.

### ■ undefined

**undefined** ähnelt **null** in der Hinsicht, dass auch dieser Wert signalisiert, dass eine Variable keinen echten Wert besitzt. Allerdings bedeutet **undefined** nicht etwa „leer“, sondern eher „unberührt“ oder „bislang einfach noch nicht mit einem anderen Wert versehen“.

Variablen tragen nach der Initialisierung den Wert **undefined**, wie sich leicht überprüfen lässt:

```
> var wert;
> console.log(wert);
undefined
```

**undefined** wird in JavaScript nicht nur als „Inhalt“ derzeit noch nicht initialisierter Variablen zu verwendet, sondern auch in anderen Zusammenhängen, wo etwas „fehlt“, ohne, dass sich jemand bewusst dazu entschlossen hat, einen „leeren“ Inhalt zu verwenden, beispielsweise (wie wir an späterer Stelle noch sehen werden) als Rückgabewert von Funktionen, die keinen echten Wert zurückgeben.

Technisch gesehen ist **undefined**, ebenso wie **null**, eine richtige Variable, und zwar vom gleichnamigen elementaren Typ, von dem es aber eben nur diese eine Variable gibt:

```
> typeof(undefined)
"undefined"
```

31

### ■ NaN

**NaN** ist die Abkürzung für *Not a Number* und drückt als spezieller Wert aus, dass eine numerische Variable keinen gültigen Zahlenwert beinhaltet, obwohl sie das eigentlich sollte.

Häufig wird **NaN** als Ergebnis von unzulässigen Rechenoperationen geliefert, wie etwa dem Ziehen der Quadratwurzel aus einer negativen Zahl:

```
> Math.sqrt(-1)
NaN
```

Interessanterweise ist **NaN** nicht etwa eine richtige Variable von einem eigenständigen Typ wie es bei **undefined** der Fall ist, sondern einfach nur ein spezieller Wert von numerischer Variablen, wie man am folgenden Code sieht:

```
> var x = Math.sqrt(-10);
> typeof(x)
"number"
```

### 31.3 · Konvertieren von Variablen

Mit dem Wert **NaN** kann man nicht rechnen, jede darauf angewendete Rechenoperation liefert wiederum **NaN**:

```
> NaN + 3;  
NaN
```

Mit **isNaN(ausdruck)** steht eine spezielle Funktion zur Verfügung, um zu prüfen, ob ein Ausdruck (Variable, Rechenvorschrift, Funktionsaufruf o.ä.) den Wert **NaN** hat:

```
> isNaN(Math.sqrt(-100))  
true  
> isNaN(Math.sqrt(100))  
false
```

## 31.3 Konvertieren von Variablen

### 31.3.1 Implizite Konvertierung

Ein Beispiel impliziter, also nicht ausdrücklich durch den Entwickler angewiesener Typkonvertierung haben wir in ► Abschn. 31.2.3 bereits kennengelernt, nämlich den Umstand, dass sich **boolean**-Variablen bei Berechnungen wie die Werte **1** (für **true**) und **0** (für **false**) verhalten. In diesem Fall hat JavaScript also den Typ automatisch umgewandelt, als es geboten erschien.

Eine in der Praxis wichtige Rolle spielen implizite Typkonvertierungen im Zusammenhang mit Strings.

```
> x=3; y='4'; z='5';  
> x * y  
12  
> y * z  
20
```

Wie Sie sehen, führt JavaScript die Berechnungen anstandslos durch, obwohl die Variablen **y** und **z** eigentlich Strings sind. Für die Berechnung werden die Werte implizit in Zahlen (Typ **number**) umgewandelt.

Dass dies hier so gut funktioniert liegt vor allem daran, dass der angewendete (Rechen-)Operator bei der Arbeit mit Strings keinerlei Bedeutung besitzt. Ganz anders dagegen stellt sich die Situation dar, wenn wir eine Addition ausführen:

```
> x + y
"34"
> y + z
"45"
```

In diesem Fall geht JavaScript davon aus, dass der „Ziel-Typ“ der Operation **string** sein soll. Deshalb wird der Operator **+** als String-Verkettungsoperator interpretiert und der Wert der **number**-Variable **x** zum Zwecke der Verkettung in einen String konvertiert; **y** und **z** sind ja bereits von Haus aus Strings, bedürfen also keiner weiteren Konvertierung.

Bei der Arbeit mit Strings, die Zahlen enthalten, ist also Vorsicht angebracht.

### 31.2 [5 min]

Welches Ergebnis haben die folgenden Operationen? Wenn Sie sich unsicher sind, probieren Sie es aus und erklären Sie das Ergebnis:

- a 'ab' + 'def'
- b '98' + '5'
- c '98' + 5
- d '98' + '5.3'
- e '98' \* 5
- f '98' \* false
- g '98' \* 'false'

#### 31.3.2 Explizite Konvertierung

## 31

Will man die beiden in den **string**-Variablen **y** und **z** des letzten Abschnitts enthaltenen Zahlen als **number**-Werte betrachten und sie addieren, genügt es nicht, die beiden Variablen mit dem Plus-Operator zu verknüpfen, weil dies zu einer String-Verkettung führt. Hier bedarf es also einer *expliziten* Konvertierung, die Werte vor der Verknüpfung mit dem Plus-Operator in Zahlen umwandelt. Dazu steht in JavaScript die Funktion **Number(nicht\_zahlwertwert)** zur Verfügung:

```
> y='4'; z='5';
> Number(y) + Number(z);
9
```

**Number()** ist eine Funktion, die einen elementaren **number**-Wert zurückgibt. Zugleich ist **Number()** aber auch die Konstruktor-Funktion des Objekts **Number**. Mit einer Zuweisung der Form **variable = new Number()** (eine Notation, die wir uns in ► Abschn. 31.5.4 und ► Abschn. 31.5.5 genauer anschauen werden) können Sie ein Objekt der Klasse **Number** erzeugen. Diese Objekte beherbergen all' jene Eigenschaften und Methoden, mit denen wir im Zusammenhang mit **number**-Variablen bereits gearbeitet haben. Und genau so kommen auch die eigentlich elementaren Datentypen wie **number** zu ihren Eigenschaften und Methoden: Greifen

### 31.3 · Konvertieren von Variablen

wir nämlich auf eine Eigenschaft oder Methode des elementaren **number**-Datentyps zu, eines Datentyps, der *primitive* ist, somit eigentlich gar kein Objekt darstellt und daher weder Methoden noch Eigenschaften haben dürfte, wandelt JavaScript im Hintergrund die Variable kurzerhand in ein Objekt des Typs **Number** um. Für dieses Objekt stehen die betreffenden Eigenschaften und Methoden dann zur Verfügung. Nach getaner Arbeit entsorgt die automatische Garbage Collection von JavaScript das nunmehr wieder ungenutzte Objekt, zurück bleibt die ursprüngliche elementare Variable. Durch diesen „Trick“ gelingt es JavaScript, auch elementare Datentypen wie richtige Objekte mit Eigenschaften und Methoden aussehen zu lassen. Soweit ein kurzer Blick „unter die Haube“ von JavaScript.

Entsprechende Funktionen existieren mit **String(nichtStringwert)** und **Boolean(nichtBooleanwert)** auch für die anderen beiden elementaren Datentypen. Im folgenden Beispiel wandeln wir den String "true" in eine echte **boolean**-Variable um.

```
> wahr = 'true';
> typeof(wahr);
"string"
> wahr = Boolean(wahr);
> typeof(wahr);
"boolean"
```

Die Funktion **Boolean()** bleibt übrigens, wie Sie leicht ausprobieren können, vollkommen unbeeindruckt, wenn Sie die üblichen Regeln der Case Sensitivity ignorieren und zum Beispiel '**True**' oder '**TRUE**' in Ihren String schreiben; die Konvertierung funktioniert dennoch einwandfrei. Das liegt daran, dass **Boolean()** alles als **true** auswertet, was nicht **0**, **null**, **undefined** oder **NaN** ist. Damit liefert etwa auch **Boolean('hallo')** den Wert **true** zurück. Während der Vielfraß **Boolean()** alles irgendwie verwertet, was man ihm gibt, ist **Number()** ungleich wählerischer: Ruft man **Number()** mit einem Argument auf, das nicht in eine Zahl umwandelbar ist, verweigert die Funktion mit der Rückgabe **NaN** den Dienst.

Insbesondere für den Konvertierung zwischen Strings und Zahlen gibt es neben **Number()** und **String()** noch einige Spezialfunktionen. **parseInt(string)** und **parseFloat(string)** verarbeiten eine Zeichenkette zu einer Zahl und leisten damit auf den ersten Blick dasselbe wie **Number()**; die Besonderheit von **parseInt()** und **parseFloat()** besteht allerdings darin, dass sie auch Zeichenketten verarbeiten, die nicht ausschließlich numerisch sind, solange, wie der numerische Teil am Anfang steht:

```
> parseInt('3 Zwillinge sind einer zuviel');
3
> parseInt('3.1415926535 lautet die Zahl Pi');
3
```

Am zweiten Beispiel sehen Sie zudem, dass `parseInt()` – wie der Name bereits suggeriert – lediglich den Ganzzahl-Anteil der am Anfang des Strings gefundenen Zahl verarbeitet. Wird zu Beginn der Zeichenketten nichts gefunden, was als Zahl interpretiert werden kann, wird `NaN` zurückgegeben.

In der Gegenrichtung der Konvertierung – von der Zahl zum String – stehen ebenfalls einige Spezialfunktionen zur Verfügung, mit deren Hilfe sich die Darstellung der Zahl als String besser steuern lässt. Die Methode `toString(zahlensystem)` von `number` (eigentlich des `Number`-Objekts, wie wir ja mittlerweile wissen), erlaubt Ihnen, die Basis des Zahlensystems anzugeben, in das Sie konvertieren wollen; `zahlensystem=2` würde also zu einer Binär-, `zahlensystem=16` zu einer Hexadezimal-Darstellung führen:

```
> zahl = 156
> zahl.toString(2)
"10011100"
> zahl.toString(16)
"9c"
```

Mit den Methoden `toFixed(dezimalstellen)` und `toExponential(dezimalstellen)` des `Number`-Objekts können Sie die Zahl der Dezimalstellen in einer herkömmlichen Darstellung sowie einer wissenschaftlichen Darstellung beeinflussen:

```
> zahl = 156.27813
> zahl.toFixed(2)
"156.27"
> zahl.toExponential(3)
"1.562e+2"
```

31

Der Exponent wird in der Exponentialdarstellung dabei so gewählt, dass stets *eine* Stelle vor dem Komma gezeigt wird.

## 31.4 Arrays

### ■ Arrays erzeugen und auf einzelne Elemente zugreifen

Arrays in JavaScript ähneln dem, was man in manchen anderen Sprachen (Darunter auch Python, vgl. ► Abschn. 21.6.1) eher als Liste kennt. Nicht nur können sie nämlich Elemente jeglichen Typs aufnehmen, die Elemente können dabei auch von unterschiedlichen Typen sein. Sogar Arrays selbst können Elemente anderer Arrays sein. Auf diese Weise lassen sich mehrdimensionale Arrays erschaffen, die JavaScript von Haus aus eigentlich nicht kennt.

Am einfachsten erzeugen lässt sich ein Array durch direkte Angabe seiner Elemente in Form eines *Array-Literals*. Die Elemente werden dabei in eckige Klammern gesetzt:

### 31.4 · Arrays

```
> primzahlen = [1,3,5,7,11,13]
```

Genauso können aber die Objekte eben auch unterschiedliche Typen besitzen:

```
> mehrereTypen = [false, 'Ulrike', 28.3]
```

Der Zugriff auf die einzelnen Elemente erfolgt über einen numerischen Index, der – wie in vielen anderen Sprachen – auch in JavaScript bei **0** beginnt und in eckigen Klammern angegeben wird. Wollten wir also auf das zweite Element des soeben erzeugten Arrays zugreifen ('Ulrike'), so würden wir das folgendermaßen bewerkstelligen:

```
> mehrereTypen[1]
"Ulrike"
```

Entsprechend würde **mehrereTypen[0]** das erste Element des Arrays, **false**, liefern.

Die Elemente von Arrays können auch leer sein; genauer gesagt, sie können *frei gelassen* werden und nehmen damit den Wert **undefined** an:

```
> mehrereTypen = [false, , 'Ulrike', 28.3, ]
> mehrereTypen
[false, empty, 'Ulrike', 28.3, empty]
```

Zeigt man sich den Inhalt des Arrays an, so wird – zumindest in der JavaScript-Konsole von Google *Chrome* – **empty** als Inhalt der freigelassenen Array-Elemente angezeigt, um deutlich zu machen, dass die Elemente „leer“ sind. Durch direktes Anzeigen dieser Elemente kann man sicher allerdings leicht davon überzeugen, dass der Inhalt tatsächlich **undefined** ist:

```
> mehrereTypen[1]
undefined
```

Arrays sind keine elementaren Datentypen, keine *primitives*, sondern *Objekte*. Deshalb können Arrays nicht nur durch Zuweisung eines Array-Literals erzeugt werden, sondern auch durch Aufruf der Konstruktorfunktion des Array-Typs, **Array()**. Das oben verwendete Array ließe sich daher auch so aufbauen:

```
> mehrereTypen = new Array(false, undefined, 'Ulrike', 28.3,
undefined)
```

Dabei müssen die freibleibenden Array-Elemente explizit mit dem Wert **undefined** belegt werden, „Leer-Lassen“ führt zu einer Fehlermeldung. Das Schlüsselwort **new** haben wir bereits in ► Abschn. 31.3.2 gesehen; in ► Abschn. 31.5.4 und ► Abschn. 31.5.5 werden wir uns mit seiner Bedeutung genauer beschäftigen.

Übrigens: Gibt man als Argument des Konstruktors **Array()** nur eine einzige, positive ganze Zahl an, dann wird nicht etwa ein Array mit nur einem Element, nämlich der angegebenen Zahl erzeugt, sondern ein komplett leeres Array mit der angegebenen Zahl als Zahl der Elemente.

### ■ Mehrere Elemente aus Arrays selektieren

Arrays stellen als Objekte eine ganze Reihe nützlicher Eigenschaften und Methoden bereit. Im Folgenden werden wir einige dieser Eigenschaften und Methoden nutzen, um intensiver mit Arrays zu arbeiten. Beginnen wir mit der Selektion von Elementen aus Arrays.

Wir haben bereits gesehen, wie man durch Angabe des Element-Index auf ein Array-Element zugreifen kann. Was aber, wenn mehrere Elemente auf einmal selektiert werden sollen? JavaScript kennt – anders als beispielsweise Python – keinen Operator, um einen ganzen Bereich von Indexwerten auf einmal zu „greifen“. Dafür besitzt das **Array**-Objekt allerdings die Methode **slice(von, bis)**, mit deren Hilfe sich dasselbe erreichen lässt. Dabei werden alle Indexwerte von Indexwert **von** bis *vor* den Indexwert **bis** zurückgeliefert, ohne jedoch das Element mit dem Indexwert **bis** selbst. Betrachten Sie dazu das folgende Beispiel:

```
> primzahlen = [1,3,5,7,11,13]
> primzahlen.slice(2,4)
[5, 7]
```

## 31

Selektiert werden also die Elemente mit den Indexwerten 2 und 3 (und damit, weil die Indizierung bei 0 beginnt, das dritte und vierte Element im Array, also 5 und 7).

Das **bis**-Argument kann auch entfallen. In diesem Fall wird der gesamte Rest des Arrays ab der Index-Position **von** zurückgegeben:

```
> primzahlen.slice(3)
[7, 11, 13]
```

Auch negative Werte sind für die **von**- und **bis**-Argumente von **slice()** möglich. In diesem Fall wird von hinten selektiert, wobei das letzte Element des Arrays den **Indexwert -1** trägt:

```
> primzahlen.slice(-3,-1)
[7, 11]
```

Beachten Sie bitte, dass auch hier bis *vor* das Element mit dem als zweites angegebenen Index selektiert wird.

## ■ Länge von Arrays feststellen und ändern

Mit der Eigenschaft **length** lässt sich die Länge von Arrays ermitteln:

```
> mehrereTypen = new Array(false, undefined, 'Ulrike', 28.3,
  undefined)
> mehrereTypen.length
5
```

Diese Eigenschaft kann auch verändert werden; wir können also ein Array kürzen, indem wir seiner Länge einen neuen, kleineren Wert zuweisen:

```
> mehrereTypen.length = 3
[false, empty, "Ulrike"]
```

„Verlängern“ wir es danach wieder, erscheinen nicht die zuvor „weggekürzten“ Elemente wieder, sondern die neuen Elemente werden mit **undefined** (in der Google-Chrome-Darstellung: **empty**) aufgefüllt:

```
> mehrereTypen.length = 5
[false, empty, "Ulrike", empty x 2]
```

Auch ein Array-Literal ist natürlich ein **Array**-Objekt, und so können wir auch bei einem Array-Literal auf die Objekt-Eigenschaften und -methoden von Arrays zugreifen. Anders als etwa bei **number** oder **string** muss dabei das Literal *nicht* in runde Klammern gesetzt werden:

```
> [1,3,5,7,11,13].length
6
```

## ■ Elemente in Arrays ändern, zu Arrays hinzufügen oder aus Arrays löschen

Bisher haben wir – mit Ausnahme der Änderung der Array-Länge – mit Arrays nur *lesend* gearbeitet. Sie können natürlich mit der Selektionstechnik, die wir bereits kennengelernt haben, den Wert eines Array-Element auch *ändern*:

```
> primzahlen = [1,3,5,7,11,13]
> primzahlen[2] = 'Lücke in Primzahlenreihe'
> primzahlen
[1, 3, "Lücke in Primzahlenreihe", 7, 11, 13]
```

*Löschen* können Sie Elemente aus einem Array mit Hilfe der Methode **splice(von, anzahl)** des Array-Objekts. Gelöscht werden dabei **anzahl** Elemente ab (und einschließlich) dem Element an Index-Position **von**. Die Methode **splice()** liefert als Rückgabewert ein Array der gelöschten Elemente:

```
> primzahlen.splice(2,3)
["Lücke in Primzahlenreihe", 7, 11]
> primzahlen
[1, 3, 13]
```

Anders als die Methoden, die für die primitives **number** und **string** zur Verfügung gestellt werden, ändert **splice()** das Original-Array, für das es aufgerufen wird.

Das Argument **anzahl** ist dabei übrigens optional: Bleibt **anzahl** unbelegt, so löscht **splice** den Rest des Arrays ab und einschließlich Index-Position **von**.

**splice()** kann aber nicht nur zum Löschen, sondern auch zum Einfügen von Elementen genutzt werden. Hinter dem zweiten Argument (**anzahl**) kann nämlich noch eine beliebige Menge weiterer Werte folgen, die *hinter* dem Element mit der Indexposition **von** eingefügt werden; die durch das Argument **anzahl** festgelegte Zahl von Elementen wird dennoch vorher gelöscht:

```
> primzahlen = [1,3,5,7,11,13]
> primzahlen.splice(1, 2, 'Lücke 1', 'Lücke 2')
[3, 5]
> primzahlen
[1, "Lücke 1", "Lücke 2", 7, 11, 13]
```

Wollen Sie also nur einfügen, nicht aber löschen, setzen sie das Argument **anzahl** auf **0**:

```
31
> primzahlen.splice(3, 0, 'Lücke 3')
[]
> primzahlen
[1, "Lücke 1", "Lücke 2", "Lücke 3", 7, 11, 13]
```

**splice()** gibt, da keine Elemente aus dem Array gelöscht wurden, ein leeres Array als Funktionswert zurück.

Neben **splice()** besitzt das Array-Objekt noch weitere Methoden, mit deren Hilfe sich dem Array Elemente hinzufügen oder aus Array entfernen lassen.

Die Methode **push()** hängt ein Element hinten an das Array an und gibt die neue Länge des Arrays als Funktionswert zurück; **pop()** löscht das letzte Element aus dem Array und gibt den Wert des gelöschten Elements zurück:

```
> primzahlen.push(57)
8
> primzahlen
[1, "Lücke 1", "Lücke 2", "Lücke 3", 7, 11, 13, 57]
> primzahlen.pop()
57
> primzahlen
[1, "Lücke 1", "Lücke 2", "Lücke 3", 7, 11, 13]
```

Die Methoden **shift()** und **unshift()** funktionieren ganz ähnlich wie **push()** und **pop()**, arbeiten allerdings mit dem Anfang des Arrays statt mit dessen Ende (probieren Sie es aus!).

### ■ Arrays verknüpfen

Zwei Arrays lassen sich mit Hilfe der Methode **concat(*anderesArray*)** bequem zusammenfügen. Dabei werden die Elemente des Array **anderesArray** hinter das letzte Element des Arrays gesetzt, dessen **concat()**-Methode aufgerufen wird. Verändert wird das Array durch den Aufruf seiner **concat()** allerdings nicht; stattdessen wird das neue, zusammengefügte Array einfach als Funktionswert zurückgegeben und kann in einer Variablen aufgefangen werden:

```
> primzahlen = [1,3,5,7,11,13]
> geradeZahlen = [2,4,6,8,10]
> zahlen = primzahlen.concat(geradeZahlen)
> zahlen
[1, 3, 5, 7, 11, 13, 2, 4, 6, 8, 10]
```

### ■ Arrays sortieren

Mit der Methode **sort()** des Array-Objekts können Sie die Elemente eines Arrays gemäß ihrer Werte *alphabetisch* aufsteigend sortieren:

```
> primzahlen = [1,3,5,7,11,13]
> primzahlen.sort()
> primzahlen
[1, 11, 13, 3, 5, 7]
```

Ihnen ist sicher aufgefallen dass **sort()** die Elemente des Arrays als Strings betrachtet und deshalb zum Beispiel 3 *hinter* 13 platziert, eine Reihenfolge, die sich bei numerischer Sortierung nicht ergäbe. Ist dieses Verhalten nicht gewünscht, lässt es sich ändern. Die Methode **sort()** besitzt nämlich ein optionales Argument, mit dem man eine Funktion angeben kann, die als Argumente zwei Werte (nennen wir sie **x** und **y**) übergeben bekommt und einen positiven Wert immer dann zurück liefert, wenn **x** in der Sortierreihenfolge vor **y** stehen soll und einen negativen Wert im umgekehrten Fall. Indem man also eine Vergleichsvorschrift angibt, die für zwei beliebige Werte entscheidet, welcher von beiden in der Reihenfolge zuerst erscheinen soll und welcher als zweites, kann man das Verhalten der **sort()**-Funktion feinsteuern. Um eine numerische Sortierung zu erreichen, könnten wir uns eine Hilfsfunktion **groesser(x,y)** schreiben, die einen positiven Wert zurückgibt, wenn **x > y**, und sonst einen negativen Wert. Der Ausdruck **(x-y)>0** ist dabei ein logischer Ausdruck, ein Ausdruck, also, der den Werte **true** oder den Wert **false** annimmt, je nachdem, wie der Vergleich von **x** und **y** ausfällt.

Damit würde unsere numerische Sortierung dann so aussehen:

```
> groesser = function(x, y) {
  return (x - y) > 0;
}
> primzahlen.sort(groesser)
> primzahlen
[1, 3, 5, 7, 11, 13]
```

Mit der Definition von Funktionen werden wir uns im übernächsten Kapitel eingehender beschäftigen.

Um die Werte in absteigender Folge zu sortieren, können wir die Methode `reverse()` des Array-Objekts benutzen. Sie dreht die Elemente eines Arrays einfach herum. Wenden wir diese Methode auf das zuvor mit `sort()` sortierte Array an, erhalten wir eine absteigende Sortierung:

```
> primzahlen.sort(groesser).reverse()
```

Die Notation mit den beiden Punkt-Operatoren mutet möglicherweise seltsam an, ist aber eigentlich sehr logisch: Der Ausdruck `primzahlen.sort(groesser)` liefert ein Array-Objekt zurück, das wiederum eine `reverse()`-Funktion besitzt. Diese wird mit der üblichen Punkt-Notation aufgerufen. Natürlich können Sie die gesamte Operation auch in zwei Schritte zerlegen.

Sowohl `sort()` als auch `reverse()` geben nicht nur das Ergebnis der jeweiligen Operation als Funktionswert zurück, sondern ändern auch das Array, für das sie aufgerufen werden. In diesem letzten Punkt ähneln sie `splice()` und unterscheiden sich von `concat()`.

## 31

### ■ Arrays als Strings darstellen

Mit `join()` und `toString()` stellt das Array-Objekt zwei Methoden bereit, um seine Elemente in einem String zusammenzufassen. Das ursprüngliche Array wird von beiden Methoden nicht angetastet.

`join()` und die von jedem JavaScript-Objekt angebotene `toString()` Methode haben grundsätzlich die gleiche Wirkung, `join()` ist jedoch insofern flexibler, als dass sich mit einem optionalen Argument das Separatorzeichen, das in dem erzeugten String zwischen den einzelnen Array-Elementen steht, angeben lässt, während `toString()` stur das Komma als Separatorzeichen verwendet:

```
> primzahlen = [1,3,5,7,11,13]
> primzahlen.toString()
"1,3,5,7,11,13"
> primzahlen.join()
"1,3,5,7,11,13"
> primzahlen.join("-")
"1-3-5-7-11-13"
```

### 31.4 · Arrays

Verwechslungsgefahr besteht bei der **join()**-Methode mit der Methode **concat()**: Anders als der Name **join()** möglicherweise suggeriert werden hier nicht *zwei verschiedene Arrays* miteinander verbunden, sondern die *verschiedenen Elemente eines Array*.

Übrigens: Wenn Sie zwei Arrays „addieren“, erhalten Sie ebenfalls einen String, der alle Elemente beider Arrays umfasst; Vorsicht ist aber geboten, denn zwischen den beiden Arrays, also zwischen dem letzten Element des ersten und dem ersten Element des zweiten Arrays wird kein Separator eingefügt (daher im Beispiel die „Zahl“ 132):

```
> primzahlen = [1,3,5,7,11,13]
> geradeZahlen = [2,4,6,8,10]
> primzahlen + geradeZahlen
"1,3,5,7,11,132,4,6,8,10"
```

#### ■ Strings zu Arrays zerlegen

Genauso, wie man mit **join()** und **toString()** Arrays in Strings umwandeln kann, ist es möglich, einen String zu zerlegen und die einzelnen Teile zu Elementen eines Arrays zu machen. Betrachten Sie im folgenden Beispiel den String **freunde**, der eine kommaseparierte Namensliste enthält:

```
> freunde = "Thomas,Markus,Ulrike,Sven"
```

Mit Hilfe der Methode **split()**, kann der String nun in die einzelnen Namen zerlegt und mit diesen ein Array „gefüttert“ werden. Das Argument von **split()** ist dabei das Separatorzeichen, das die einzelnen Teile des Strings voneinander trennt, in unserem Beispiel also das Komma:

```
> freundeArray = freunde.split(",")
> freundeArray
["Thomas", "Markus", "Ulrike", "Sven"]
```

#### ■ Wiederholung: Zugriff auf String-Zeichen in Array-Notation

In JavaScript sind Strings, wie wir in ► Abschn. 31.2.2 gesehen haben, ein elementarer Datentyp, ein *primitive*, und damit kein Objekt. Wie Sie sich erinnern, kann auf ihre einzelnen Zeichen dennoch zugegriffen werden wie auf die Elemente eines Arrays:

```
> var nachricht = "Hallo Welt!"
> nachricht[1]
"a"
> nachricht[nachricht.length-1]
"!"
```

Mit der letzten Anwendung greifen wir auf das letzte Zeichen im String zu (bedenken Sie, dass die Indizierung bei 0 beginnt!). Ändern können Sie die einzelnen Zeichen eines Strings mit der Array-Notation allerdings nicht. Zwar führt ein solcher Versuch nicht zu einer Fehlermeldung, aber die versuchte Änderung wird im String nicht wirksam. Strings sind gewissermaßen „Nur-Lese-Arrays“.

### 31.2 [5 min]

Erzeugen Sie einen String mit dem Wert **'Hallo Welt'**. Selektieren Sie das fünfte und das siebte Zeichen aus diesem Zeichenkette. Entfernen Sie danach diese Zeichen aus dem String.

## 31.5 Objekte

### 31.5.1 Objektorientierung in JavaScript

Objektorientierung in JavaScript funktioniert etwas anders als in den meisten übrigen objektorientierten Sprachen. Das zentrale Grundkonzept der objektorientierten Programmierung, die *Klassen*, kennt JavaScript von Haus aus nicht. JavaScript verfolgt stattdessen einen Ansatz, bei dem Objekte auf Basis sogenannter *Prototypen* erzeugt werden. „Prototypen“ und „Klassen“ mögen zwar zunächst nicht sonderlich verschieden klingen, haben doch auch die Klassen in der objektorientierten Programmierung eine Prototyp-Funktion für die davon abgeleiteten Objekte (diese sind ja gewissermaßen nach dem „Prototyp“ der Klasse aufgebaut). Tatsächlich aber unterscheiden sich die beiden Ansätze durchaus erheblich.

Zwei Beispiele: Anders als in typischen objektorientierten Sprachen mit ihren Klassen, können in JavaScript einzelnen Objekten (in der normalen objektorientierten Terminologie würde man sagen: den Instanzen von Klassen) Eigenschaften und Methoden *direkt hinzugefügt* werden, die nicht in der in der Typdefinition enthalten sind. Ohne also weitere Klassen von einer Basisklasse abzuleiten, können sich unterschiedliche Objektinstanzen der Basisklasse in ihren Methoden und Eigenschaften voneinander unterscheiden. In anderen objektorientierten Sprachen hätten alle Objekte desselben Typs (das heißt, derselben Klasse) denselben Aufbau in Gestalt von verfügbaren Methoden und Eigenschaften (letztere natürlich durchaus mit unterschiedlichen Werten belegt). Ein weiterer Unterschied zwischen dem prototypischen Ansatz von JavaScript und dem klassen-basierten Ansatz anderer objektorientierter Sprachen besteht darin, dass in JavaScript der Aufbau der Klassen nicht explizit beschrieben wird; es gibt (oder besser: gab, bis ECMAScript-Version 2015, die aber an dem prototypischen Ansatz nichts ändert) also keine Syntaxstruktur, die eine Klasse abstrakt als Sammlung von Methoden und Eigenschaften beschreibt, wie wir es etwa in Python gesehen haben. Stattdessen ist die Typdefinition in JavaScript komplett in den Konstruktor des Typs und seinen Bezug auf die Prototypen anderer Objekte, von denen er Methoden „erbt“, verpackt (die Vererbung von Eigenschaften macht bei einem Prototyp-Ansatz eigentlich keinen Sinn,

es sei denn man wünscht im Sinne der objektorientierten Programmierung *statische* Eigenschaften, die also für alle Objektinstanzen den gleichen Wert besitzen).

Wenn Ihnen der letzte Absatz sehr technisch und der überfliegenden Lektüre eher schwer zugänglich vorkommt, keine Sorge: Wir werden in den folgenden Abschnitten die meisten Untiefen des JavaScript-eigenen Prototyp-Ansatz weiträumig umschiffen. Ihre Behandlung würde die Idee des Buches, die *wichtigsten* Grundkonzepte der Sprache *schnell* zu erlernen, um sie *anwenden* zu können, deutlich übersteigen.

Insbesondere verzichten wir hier – anders als im Python-Teil des Buchs – darauf, uns mit Vererbung zu beschäftigen und damit Objekte von anderen Objekten bzw. deren Prototypen abzuleiten. Die meisten Anwendungsszenarien werden Sie gut meistern können, auch ohne JavaScripts (vor allem wenn man aus einer „klassischen“ objektorientierten Denkweise kommt) ungewöhnlichen, aber sehr flexiblen Ansatz der Objektorientierung vollends bis ins letzte Detail verinnerlicht zu haben. Es ist vermutlich nicht unfair, anzunehmen, dass es Ihnen hier nicht anders geht als vielen, die JavaScript durchaus erfolgreich auch im professionellen Umfeld einsetzen.

### 31.5.2 Objekte direkt erzeugen

Der einfachste Weg, ein Objekt in JavaScript zu erzeugen, ist eine Variablen-deklaration, bei der einer (Objekt-)Variablen nicht nur ein Wert, sondern gleich (eine oder häufiger) mehrere Eigenschaften zugewiesen werden. Um dies zu demonstrieren, greifen wir das bereits mehrfach bemühte Beispiel einer Produktdefinition auf, die ein Produkt durch seine Bezeichnung und seinen Preis beschreibt.

Ein Objekt mit ebendiesen Eigenschaften lässt sich in JavaScript leicht durch eine Zuweisung wie die folgende erschaffen:

```
var produkt = {  
    bezeichnung: "Gartenschaufel, Edelstahl",  
    preis: 10.99  
}
```

Wie Sie sehen, wird hier eine Variable namens **produkt** deklariert und ihr im Zuge der Deklaration direkt etwas zugewiesen. Bei dem, was zugewiesen wird, handelt es sich um die Eigenschaften des Produkts. Die geschweiften Klammern machen deutlich, dass es sich bei **produkt** um ein selbst definiertes Objekt handelt. Die in den geschweiften Klammern angegebenen Eigenschaften bestehen jeweils aus einem Bezeichner, in unserem Beispiel **bezeichnung** und **preis**, und einem Wert, mit dem die Eigenschaft hinter dem Doppelpunkt belegt wird. Mehrere Eigenschaften werden innerhalb der Objektdeklaration durch Kommata getrennt.

Die Eigenschaften des Produkts können auch mit Variablen belegt werden (deren momentaner Wert damit der Eigenschaft zugewiesen wird); insbesondere können die Werte von Eigenschaften wiederum andere Objekte sein.

In der Praxis eher selten anzutreffen, aber syntaktisch zulässig ist es auch, die Bezeichner der Eigenschaften in (einfache oder doppelte) Anführungszeichen zu setzen: Das erlaubt es Ihnen, sogar Leerzeichen in die Eigenschaftsbezeichner aufzunehmen, zum Beispiel:

```
var produkt = {
    'bezeichnung des produkts': 'Gartenschaufel, Edelstahl',
    preis: 10.99
}
```

Auf diese Weise können Sie Eigenschaften mit Bezeichnern versehen, die sonst in JavaScript unzulässig wären, wie etwa **#hashtag** (unzulässig wegen erstem Zeichen).

Auch ansonsten reservierte Schlüsselwörter wie **var** könnten Sie als Eigenschaftsbezeichner verwenden. Auch das ist aber wegen der erwartbar schlechtbaren Lesbarkeit und der höheren Fehleranfälligkeit des Codes eher unüblich.

Wie bei „freistehenden“ Variablen auch, entscheidet JavaScript selbst über den Datentyp, den die Eigenschaften haben müssen; in unserem Beispiel wird **bezeichnung** vom Typ **string**, **preis** vom Typ **number** sein.

### 31.5.3 Auf Eigenschaften von Objekten zugreifen

Objekte sind also letztlich nichts weiteres als *assoziative Arrays*: Felder von Eigenschaften, die aus Schlüssel-Wert-Pärchen bestehen. Was aber ist mit Methoden?

31

Im Sinne der objektorientierten Programmierung verstehen wir Objekte als Konstrukte, die Eigenschaften (*Attribute*) und *Methoden*, also Funktionen, umfassen, mit deren Hilfe mit den Eigenschaften gearbeitet werden kann. Die Methoden aber können ja nun nicht zugleich Eigenschaften sein. Wie also kann ein Objekt in JavaScript praktisch ein assoziatives Array sein, dass *nur* Eigenschaften besitzt? Der Trick besteht darin, dass in JavaScript auch Funktionen Objekte sind, und zwar Objekte vom Typ **function**. Wir hatten aber im letzten Abschnitt gesehen, dass der Wert einer Objekt-Eigenschaft wiederum ein Objekt sein kann und damit eben auch eine Funktion.

Wenn Objekte also letztlich eine Art assoziatives Array sind, dann lässt sich auf ihre Eigenschaften (und damit auch Methoden) durch Angabe des Schlüssels, also des Eigenschaftsbezeichners, zugreifen. Der muss zu diesem Zweck in Anführungszeichen und eckigen Klammern angegeben werden. Nachdem Sie die Objektdeklaration aus dem letzten Abschnitt in der JavaScript-Konsole ausgeführt haben, können Sie nun leicht auf die **preis**-Eigenschaft des Objekts **produkt** zugreifen:

```
> produkt['preis'];
10.99
```

Der Schlüssel darf dabei natürlich auch selbst eine Variable sein:

```
> eigenschaft = 'preis';
> produkt[eigenschaft]
10.99
```

Diese Notation macht den Charakter von JavaScript-Objekten als assoziative Arrays deutlich.

In der Praxis wesentlich häufiger anzutreffen ist aber die Zugriffsweise, die auch aus vielen anderen objektorientierten Sprachen bekannt ist, nämlich mit Hilfe des Punkt-Operators:

```
> produkt.preis
10.99
```

Diese Art des Zugriffs funktioniert nur dann, wenn die Bezeichner Ihrer Objekt-Eigenschaften zulässige JavaScript Bezeichner sind, also nicht etwa Leerzeichen enthalten oder mit einem anderen Sonderzeichen als Unterstrich und Dollarzeichen beginnen. Es ist allerdings auf jeden Fall zu empfehlen, stets Bezeichner zu wählen, die den üblichen Regeln genügen.

### 31.5.4 Objekte mit Hilfe des Object-Konstruktors erzeugen

In ► Abschn. 31.5.2 haben wir ein Objekt dadurch erzeugt, dass wir einer Variablen bei Ihrer Deklaration eine Reihe von Schlüssel-Wert-Pärchen, nämlich die Eigenschaften des Objekts zugewiesen haben. In diesem Abschnitt werden wir eine zweite Art kennenlernen, ein Objekt zu erzeugen.

Dabei machen wir uns die Eigenschaft zu Nutze, dass in der JavaScript-Objekt-hierarchie alle Objekte vom grundlegenden Typ **object** abgeleitet sind. Daher erzeugen wir zunächst eine Variable vom Typ **object**, indem wir die Konstruktorfunktion aufrufen (mehr zu JavaScript-Konstruktoren dann im folgenden Abschnitt). Beachten Sie dabei bitte die Großschreibung der Konstruktorfunktion:

```
> var produkt = new Object();
> typeof(produkt);
"object"
```

Damit haben wir ein leeres Objekt erschaffen. Wenn Sie **produkt**. in die Konsole eingeben, sehen Sie an dem sich nun öffnenden Popup-Fenster, dass unser vermeintlich leeres Objekt bereits eine ganze Reihe Eigenschaften und Methoden enthält, nämlich die, die der Typ **object** standardmäßig besitzt.

Wir aber wollen dem Objekt ja unsere Produkt-Eigenschaften **bezeichnung** und **preis** mitgeben. Das geschieht nun durch einfache Zuweisung:

```
> produkt.bezeichnung = 'Gartenschaufel, Edelstahl';
> produkt.preis = 10.99;
```

Sie können leicht überprüfen, dass unser **produkt**-Objekt tatsächlich die Eigenschaften **bezeichnung** und **preis** mit den entsprechenden Werten besitzt:

```
> produkt.bezeichnung
"Gartenschaufel, Edelstahl"
> produkt.preis
10.99
```

### 31.5.5 Objekte mit Hilfe von Konstruktor-Funktionen erzeugen

Der letzte – und wahrscheinlich wichtigste – Weg, Objekte zu generieren, besteht darin, eine Konstruktor-Funktion zu schreiben, die das Objekt erzeugt. Auch im vorangegangenen Abschnitt haben wir eine Konstruktor-Funktion verwendet, und zwar den Konstruktor **Object()**. In diesem Abschnitt nun werden wir uns nun selbst einen Konstruktor bauen, um damit einen ganz eigenen Typ von Objekt zu erschaffen.

Unser **Produkt**-Objekt soll zwei Eigenschaften, **bezeichnung** und **preis**, besitzen; diese sollten direkt beim Erzeugen des Objekts angebbar sein. Das erreichen wir mit einer Konstruktor-Funktion wie der folgenden:

```
function Produkt(preis, bezeichnung) {
    this.preis = preis;
    this.bezeichnung = bezeichnung;
}
```

Dem Konstruktor werden als Argumente zwei Werte übergeben, die dann Eigenschaften des neu geschaffenen Objekts zugewiesen werden; dabei kommt das Schlüsselwort **this** zum Einsatz. Es verweist auf das aktuelle Objekt, in dessen Kontext es verwendet wird. In unserem Beispiel also auf unser gerade entstehendes **Produkt**-Objekt.

Damit ist die Definition unseres sehr einfachen Typs auch schon fertig. Wir können nun eine Instanz dieses Typs erzeugen, indem wir die gerade entwickelte Konstruktorfunktion aufrufen:

```
> gartenschaufel = new Produkt(10.99, 'Gartenschaufel, Edelstahl');
```

Beachten Sie dabei das Schlüsselwort **new**. Es sorgt dafür, dass ein neues Objekt erzeugt wird. Wird die Konstruktorfunktion ohne dieses Schlüsselwort aufgerufen, wird kein neues Objekt angelegt, sondern einfach nur **undefined** zurückgegeben.

### 31.5.6 JSON

Auch wenn Sie nicht mit JavaScript arbeiten, stehen die Chancen gut, dass Ihnen irgendwann einmal das Datenformat JSON begegnen wird. Es findet beispielsweise häufig Anwendung bei Internet-APIs, über die Informationen von Web-Services abgefragt werden. Solche Schnittstellen geben nicht selten ihre Ergebnisse im JSON-Format an die aufrufende Anwendung zurück. Neben XML ist es das zweite bedeutende Datenaustauschformat im Internet.

JSON ist die Abkürzung für *JavaScript Object Notation*, und so verwundert es sicher alleine des Namens wegen schon nicht, dass wir zum Abschluss unserer Betrachtung von Objekten in JavaScript unsere Aufmerksamkeit noch auf dieses populäre Datenaustauschformat richten.

Das JSON-Format ist uns bereits begegnet, und zwar in ► Abschn. 31.5.2, wo wir Objekte als Literale direkt erzeugt haben. Die dort verwendete Notation mit ihren Schlüssel-Wert-Pärchen, die kommasepariert in geschweifte Klammern gestellt werden, ist nämlich nichts anderes als eine Schreibweise im JSON-Format.

Betrachten Sie den folgenden Auszug aus einem JSON-Datensatz:

```
{
    'kunde01': {
        'Vorname': 'Paul',
        'Nachname': 'Mayer',
        'Adresse': {
            'Strasse': 'Hörnchenweg',
            'Hausnummer': 58,
            'Postleitzahl': 81249,
            'Stadt': 'München'
        }
    },
    'kunde02': {
        'Vorname': 'Lydia',
        'Nachname': 'Welti',
        'Adresse': {
            'Strasse': 'Escherweg',
            'Hausnummer': 82,
            'Postleitzahl': 53119,
            'Stadt': 'Bonn'
        }
    }
}
```

Dieses JSON-Objekt (begrenzt durch die äußersten geschweiften Klammern), umfasst zwei (Unter-)Objekte, **kunde01** und **kunde02**, die durch verschiedene Felder charakterisiert sind. Eines der Felder, die Adresse, ist dabei selbst wieder ein Objekt, das aus verschiedenen Feldern zusammengesetzt ist.

Die Feldnamen, also die Schlüssel der Schlüssel-Wert-Pärchen, werden in JSON in Anführungszeichen gesetzt, was – wie bereits erwähnt – auch in JavaScript syntaktisch zulässig ist.

Mit Hilfe der Funktion **JSON.stringify(Objekt)** können Sie ein beliebiges Objekt in einen JSON-String konvertieren, was auch als *Serialisierung* bezeichnet wird; umgekehrt erlaubt **JSON.parse(string)** einen JSON-String (den man zum Beispiel als Resultat eines Web-API-Aufrufs zurückgegeben bekommt), in ein echtes JavaScript-Objekt zu verwandeln.

Obwohl die Herkunft von JSON eng mit JavaScript verknüpft ist, bieten heute praktische alle gängigen Programmiersprachen Funktionen zur Arbeit mit Daten im JSON-Format an, was die Popularität von JSON als Datenaustauschformat unterstreicht. Diese Popularität röhrt sicherlich auch daher, dass JSON es erlaubt, hierarchische Objektstrukturen mit geringem Aufwand auf eine syntaktisch unkomplizierte Art und Weise abzubilden, die selbst ohne JSON-Kenntnisse gut gelesen werden kann.

### 31.3 [5 min]

Erzeugen Sie ein Array aus Objekten, die jeweils den Namen und die Altersangabe (in Jahren) Ihrer engeren Familienmitglieder enthalten. Greifen Sie dann auf den Namen der zweiten Personen in diesem Array zu und lassen ihn sich anzeigen. Konvertieren Sie schließlich dieses Array in einen JSON-String.

### 31.4 [10 min]

Erzeugen Sie zwei Objekte, die jeweils eine Kundenadresse enthalten. Arbeiten Sie dabei einmal mit der direkten Erzeugung von Objekten durch Zuweisung von Schlüssel-Wert-Pärchen zu einer Variable, einmal mit dem allgemeinen Objekt-Konstruktork **Object()**.

## 31.6 Lösungen zu den Aufgaben

### Aufgabe 31.1

```
> Infinity + 1
Infinity
> Infinity + 1 == Infinity
true
```

Hier benutzen wir den speziellen Wert **Infinity**, mit dem wir selbstverständlich auch rechnen können. Anhand der zweiten Eingabe sehen Sie übrigens, dass unendlich plus eins wiederum gleich unendlich ist (das doppelte Gleichheitszeichen ist der

Vergleichsoperator). Der Vergleich ergibt also eine wahre Aussage, dementsprechend wird der Wert **true** zurückgeliefert.

### ■ Aufgabe 31.2

- a. **'abc' + 'def' = 'abcdef'**. Strings werden durch den Plus-Operator miteinander verkettet.
- a. **'98' + '5' = '985'**. Der Plus-Operator hängt auch hier die beiden Strings aneinander. Dass diese zufälligerweise Zahlen beinhalten, spielt dabei keine Rolle.
- b. **'98' + 5 = '985'**. Hier ist der zweite Summand eine echte Zahl. Vorrang bei der Abarbeitung des Ausdrucks hat aber das Plus als *String-Verkettungsoperator*. Deshalb wandelt JavaScript die Zahl **5** implizit in einen String um, um sie mit dem String **'98'** verketten zu können.
- c. **'98' + '5.3' = '985.3'**. Auch, wenn die Strings gebrochene Zahlen enthalten, führt der Plus-Operator zu einer String-Verkettung.
- d. **'98' \* 5 = 490**. Der Multiplikationsoperator hat für Strings keine Bedeutung. Weil deshalb eine Operation mit Zeichenketten hier nicht in Frage kommt, konvertiert JavaScript implizit den String **'98'** in eine Zahl, um so doch noch eine sinnvolle Operation durchführen zu können.
- e. **'98' \* false = 0**. Die Konstante **false** wird intern mit dem Wert **0** gewertet. Da die Multiplikation für Strings keinen Sinn macht, versucht JavaScript eine Operation mit Zahlen und wandelt zu diesem Zweck den String **'98'** in eine Zahl um.
- f. **'98' \* 'false' = NaN**. Hier kapituliert auch JavaScript. Die Multiplikation ist offensichtlich eine Operation mit Zahlen als Operanden, allerdings bekommt JavaScript hier zwei Strings serviert. Das Ergebnis ist *not a number*. Keineswegs werden hier beide Strings in Zahlenwerte konvertiert, obwohl das bei **'98' \* '5'** durchaus geschehen würde (probieren Sie es aus!).

Wie Sie sehen, ist es nicht ganz einfach, JavaScripts implizite Konvertierungen vorherzusehen, ohne das vollständige Regelwerk detailliert zu kennen. Es ist daher am besten, sich nicht auf die implizite Konvertierung zu verlassen, sondern sicherzustellen, dass überall dort, wo es vonnöten sein könnte, eine explizite Konvertierung vorgenommen wird.

### ■ Aufgabe 31.3

Strings verhalten sich im *lesenden Zugriff* wie Arrays, dementsprechend können wir die Zeichen in Array-Notation herausselektieren:

```
> nachricht = 'Hallo Welt'  
> nachricht[5]  
"e"  
> nachricht[7]  
"t"
```

Bzgl. des bearbeitenden Zugriffs verhalten sich String allerdings nicht wie Arrays. Ein Einsatz der Array-Methode **splice()**, um die Elemente zu löschen, scheidet daher aus. Ein Weg, den gewünschten Effekt dennoch zu erzielen, besteht darin, sich den String „zusammenzustückeln“, indem man mit **slice()**, die entsprechenden Teilstücke so zuschneidet, dass die Zeichen an den Index-Positionen 5 und 7 herausfallen. Beachten Sie dabei, dass **slice()** immer bis *vor* den angegebenen zweiten Index selektiert:

```
> nachricht = nachricht.slice(0,5) + nachricht.slice(6,7) +
nachricht.slice(8,nachricht.length)
```

### ■ Aufgabe 31.4

Wir erzeugen ein Array, dessen Elemente Objekte sind, die wir in der üblichen Objekt-Notation als Schlüssel-Wert-Pärchen hinterlegen:

```
> familie = [{name: 'Mark', alter: 28}, {name: 'Petra', alter:
54}, {name: 'Ulrich', alter: 57}]
```

Den Namen der zweiten Person erhalten wir dadurch, dass wir zunächst das Personen-Objekt anhand seines Index aus dem Array selektieren (**familie[1]**) und so dann auf dessen Eigenschaft **name** zugreifen, was wir selbstverständlich in der üblichen Punkt-Notation tun:

```
> familie[1].name
"Petra"
```

31

Bei der Umwandlung in einen JSON-String hilft die Methode **stringify()** des globalen **JSON**-Objekts.

```
> JSON.stringify(familie)
"[{"name":"Mark","alter":28}, {"name":"Petra","alter":54},
 {"name":"Ulrich","alter":57}]"
```

### ■ Aufgabe 31.5

Direkte Erzeugung in Objekt-Notation als kommaseparierte Liste von Schlüssel-Wert-Pärchen:

```
> var kundel = { Kundennummer: 14527, Name: 'Müller', Vorname:
'Hans-Peter', Strasse: 'Hummelweg 1', PLZ : '20257', Stadt:
'Hamburg' }
```

```
> var kunde2 = { Kundennummer: 19321, Name: 'Schmidt', Vorname:  
'Christian', Strasse: 'Bahnhofsplatz 21', PLZ : '70173', Stadt:  
'Stuttgart' }
```

Erzeugung dadurch, dass wir zunächst vom allgemeinen **Object()**-Konstruktor ein „leeres“ Objekt erzeugen lassen und dieses dann mit unsere Eigenschaften ausstatten:

```
> var kundel = new Object()  
> kundel.Kundennummer = 14527  
> kundel.Name = 'Müller'  
> kundel.Vorname = 'Hans-Peter'  
> kundel.Strasse = 'Hummelweg 1'  
> kundel.PLZ = '2025'  
> kundel.Stadt = 'Hamburg'
```

Das Vorgehen für **kunde2** ist analog.

## 31.7 Zusammenfassung

---

In diese Kapitel haben wir uns mit elementaren Variablen und Objekten in JavaScript beschäftigt.

Folgende Punkte sollten Sie aus diesem Kapitel unbedingt mitnehmen:

- Variablen müssen in JavaScript nicht zwingend deklariert werden, allerdings ist es gute Praxis, das zu tun.
- Die wesentlichen elementaren Datentypen sind **number** (Zahlen, sowohl ganze als auch Fließkommazahlen), **string** (Zeichenketten) und **boolean** (Wahrheitswerte).
- Strings können in einfache oder doppelte Anführungszeichen eingeschlossen werden; auf einzelne Zeichen eines Strings kann in einfacher Array-Notation (**zeichenkette[zeichenindex]**) zugegriffen werden, allerdings nur lesend.
- Für die elementaren Datentypen, die selbst keine Objekte sind, existieren jeweils Objektprototypen gleichen Namens (allerdings mit großem Anfangsbuchstaben, also beispielsweise **Number**), die nützliche Eigenschaften und Methoden bereitstellen; wenn erforderlich, konvertiert JavaScript automatisch im Hintergrund die elementaren Datentypen vorübergehend in Objekte des entsprechenden Typs, sodass Sie auf die Eigenschaften und Methoden zugreifen können, als wären es Eigenschaften bzw. Methoden der elementaren Datentypen selbst.
- Der spezielle Wert **undefined** kommt dann zum Einsatz, wenn eine Variable noch keinen definierten Wert hat (oder eine Funktion keinen echten Rückgabewert liefert), **null** dagegen immer dann, wenn eine bewusste Entscheidung dafür getroffen wird, dass eine Variable „leer“ bleiben soll.
- JavaScript konvertiert bereits implizit, wo nötig.

- Vorsicht ist geboten bei der Konvertierung von Strings: der Plus-Operator dient bei Strings der Verknüpfung zweier Zeichenketten, hier kann es zu unerwünschten Effekten kommen, wenn Zahlen, die in Zeichenketten enthalten sind, numerisch addiert werden sollen, aber zuvor nicht explizit in den **number**-Typ konvertiert werden.
- Eine explizite Konvertierung der elementaren Datentypen untereinander kann mit Hilfe der Konstruktoren der zu den elementaren Typen gehörenden Objekt-Typen erreicht werden, also beispielsweise mit **Number(zeichenkette)**.
- Arrays sind Listen, die Variablen (auch Objekte) unterschiedlicher Datentypen aufnehmen können; auf die einzelnen Elemente des Arrays wird in der Notation **array[elementindex]** zugegriffen, wobei die Indizierung bei **0** beginnt.
- JavaScript kennt in seiner Kerndefinition keine Klassen; Objektorientierung wird mit Hilfe sogenannter Prototypen realisiert, nach deren Ebenbild Objekte dann erzeugt werden können; den Instanzen von Objekten können weitere Eigenschaften und Methoden hinzugefügt werden, die der Prototyp, dessen Bauweise das Objekt folgt, nicht besaß.
- Erzeugt werden können Objekte direkt, indem einer Variable in geschweiften Klammern eine kommaseparierte Liste von Elementen/Eigenschaften in Form von Schlüssel-Wert-Pärchen (Objekt-Notation) zugewiesen werden, also beispielsweise: **objekt = { eigenschaft1: wert1, eigenschaft2: wert2 }**. In ähnlicher Weise kann zunächst ein „leeres“ Objekt mit dem Objekt-Konstruktor **Object()** durch eine Zuweisung der Form **objekt = new Object()** erzeugt werden und in diesem dann sukzessive Eigenschaften angelegt werden, durch Zuweisungen der Form **objekt.eigenschaft = wert**.
- Daneben können Sie Objekte erzeugen, indem Sie den Konstruktor des jeweiligen Objekttyps aufrufen, zum Beispiel **objekt = MeinObjekt()**.
- Auf die Elemente/Eigenschaften von Objekten können Sie mit Hilfe des Punktoperators in der Notation **objekt.eigenschaft** zugreifen oder wie bei einem (assoziativen) Feld in einer Arraynotation der Form **objekt['eigenschaft']**.
- Die *JavaScript Object Notation (JSON)* ist ein gängiges Austauschformat für Daten im Internet. Sie entspricht exakt der Notation, die auch bei der direkten Erzeugung von Objekten als in geschweiften Klammern angegebene, kommaseparierte Liste von Schlüssel-Wert-Pärchen verwendet wird. Jedes JSON-Dokument lässt sich daher mit **JSON.parse(jsonDokument)** in eine JavaScript-Objekt verwandeln und umgekehrt jedes JavaScript-Objekt mit **JSON.stringify(objekt)** als JSON-Dokument darstellen.



# Wie lasse ich Daten ein- und ausgeben?

## Inhaltsverzeichnis

- 32.1 Überblick über die Ein- und Ausgabe in JavaScript – 493**
- 32.2 Ausgabe über die Konsole – 494**
- 32.3 Ein- und Ausgaben über Dialogboxen – 497**
- 32.4 Ausgabe in ein HTML-Dokument / Webseite – 498**
  - 32.4.1 HTML-Code in die Webseite schreiben – 498**
  - 32.4.2 Das Document Object Model (DOM) – 500**
  - 32.4.3 DOM-Knoten über ihre Eigenschaften selektieren – 501**
  - 32.4.4 DOM-Knoten über die hierarchische Struktur des Dokuments selektieren – 503**

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann [https://doi.org/10.1007/978-3-658-29850-0\\_32](https://doi.org/10.1007/978-3-658-29850-0_32).

- 32.4.5    HTML-Elemente ändern – 506
- 32.4.6    HTML-Elemente hinzufügen  
und löschen – 508
- 32.5    Eingabe mit Formularen – 510**
- 32.5.1    Formulare in HTML – 510
- 32.5.2    Formulare aus JavaScript heraus  
ansteuern – 513
- 32.6    Beispiel: Einfacher  
Taschenrechner – 517**
- 32.6.1    Die Web-Oberfläche – 518
- 32.6.2    Die CSS-Designanweisungen – 521
- 32.6.3    Der JavaScript-Code – 524
- 32.7    Beispiel: Color Picker – 525**
- 32.7.1    Die Web-Oberfläche – 525
- 32.7.2    Der JavaScript-Code – 528
- 32.8    Zusammenfassung – 530**
- 32.9    Lösungen zu  
den Aufgaben – 531**

## Übersicht

JavaScript ist die Sprache des Web. Kein Wunder also, dass sich auch bei der Ein- und Ausgabe von Daten fast alles um die Interaktion mit der Webseite dreht, innerhalb derer Ihr JavaScript-Programm läuft.

JavaScript bietet viele Möglichkeiten, Informationen vom Benutzer der Webseite entgegenzunehmen und die Webseite zur Ausgabe von Informationen zu verändern. Die Interaktion mit der umgebenden Webseite wird dadurch möglich, dass JavaScript über das sogenannte *Document Object Model (DOM)* der Webseite einen Zugriff auf deren einzelne Elemente erlaubt.

Bevor wir uns aber der Nutzung des DOM zuwenden, wollen wir noch einen kurzen Blick auf die Ausgabe mit der JavaScript-Konsole, die wir bereits häufig verwendet haben, werfen, und uns die Arbeit mit Dialogboxen ansehen.

In diesem Kapitel werden Sie folgendes lernen:

- wie Sie Objekte in der Konsole ausgeben können
- wie Sie mit Template Literals und String-Ersetzungen bequem Ausgabe-Zeichenketten unter Verwendung von Variablen aufbauen
- wie Sie über Dialogboxen Informationen ausgeben und vom Benutzer Entscheidungen abfragen
- wie Sie mit JavaScript auf einfache Weise in den HTML-Code einer Webseite hinschreiben können
- wie das Document Object Model (DOM) aufgebaut ist
- wie Sie auf Basis des DOM einzelne Elemente einer Webseite selektieren und verändern können
- wie Sie Formulare benutzen, um Eingaben vom Benutzer entgegenzunehmen.

## 32.1 Überblick über die Ein- und Ausgabe in JavaScript

Die einfachste Art, mit JavaScript Daten auszugeben, ist die Ausgabe in der Konsole. Von dieser Möglichkeit haben wir im letzten Kapitel ausgiebig Gebrauch gemacht, vor allem, weil die Konsole eine Interaktivität bietet, die beim Ausprobieren neuer Sprachkonzepte nützlich ist. Man gibt eine Anweisung ein und erhält sofort das Ergebnis angezeigt.

Wenn Sie echte JavaScript-Anwendungen schreiben, werden Sie aber typischerweise nicht mit der Konsole arbeiten wollen, die ja ein Entwicklerwerkzeug somit für den Betrachter Ihrer Website normalerweise unsichtbar ist. Stattdessen soll Ihre JavaScript-Anwendung mit der Webseite interagieren und Informationen *dort* ausgegeben, sodass der Betrachter sie auch wahrnehmen kann.

In diesem Kapitel werden wir uns – nachdem wir nochmal kurz einen genaueren Blick auf die bereits bekannte Konsolenausgabe geworfen haben – ansehen, wie sie über *Dialogfelder* Daten aus- (und auch ein-)geben lassen können. Danach aber wenden wir uns dem Kernbereich der Ein- und Ausgabe zu, der Arbeit mit der Webseite, die das Skript aufruft. Dabei werden wir sehen, wie einzelne Elemente der Webseite über das sogenannte *Document Object Model (DOM)* von Ihrem JavaScript-Programm aus verändert werden können. Insbesondere werden wir uns da-

bei auch mit *Formularen* beschäftigen, die für die Interaktivität von Websites besonders nützlich sind, weil sie es dem Benutzer erlauben, Text und andere Informationen direkt einzugeben. Die Arbeit mit Formularen ist nicht ohne Grund ein wichtiges Anwendungsgebiet von JavaScript.

Anders als bei anderen Programmiersprachen werden wir uns bei JavaScript nicht mit der Arbeit mit Dateien befassen, denn aus Sicherheitsgründen hat JavaScript normalerweise keinen Zugang zum Dateisystem des lokalen Computers.

Abschließen werden wir das Kapitel mit zwei kleinen Beispielanwendungen, einem einfachen Taschenrechner und einem Color Picker, mit dem man die in HTML üblichen hexadezimalen Farbcodes bequem erzeugen kann.

## 32.2 Ausgabe über die Konsole

Im letzten Kapitel haben wir bereits häufig mit der Methode `log()` des `console`-Objekts gearbeitet, um Daten schnell und problemlos in die Konsole auszugeben. In diesem Abschnitt schauen wir uns das nochmal etwas genauer an und beschäftigen uns insbesondere mit der Frage, wie eine aus mehreren unterschiedlichen Teilen zusammengesetzte Ausgabe mit `console.log()` bewerkstelligt werden kann. Die hier für `console.log()` vorgestellten Ansätze, sind übertragbar und können vielerorts verwendet werden, wo mit Strings gearbeitet wird.

### ■ Liste mehrerer Objekte ausgeben

Wollen Sie mehrere Werte/Objekte hintereinander ausgeben, übergeben Sie sie `console.log()` einfach als kommaseparierte Liste:

```
> console.log('Heute ist: ', Date(), '. Eine Zufallszahl
zwischen 0 und 10 lautet: ', Math.round(Math.random()*10,0))
Heute ist: Thu Oct 03 2019 13:24:53 GMT+0200 (Mitteleuropäische
Sommerzeit) . Eine Zufallszahl zwischen 0 und 10 lautet: 4
```

32

`console.log()` gibt einfach stur hintereinander die als Argumente übergebenen Objekte aus. Die Ausgaben der unterschiedlichen Objekte werden dabei durch ein Leerzeichen getrennt.

### ■ Mehrere Objekte als verketteten String ausgeben

Wollen Sie dieses Leerzeichen vermeiden, müssen Sie die Objekte zunächst zu einem String zusammenbauen, wobei Sie das Auftreten von Leerzeichen wie gewünscht steuern, und dann den vollständigen String ausgeben lassen:

```
> ausgabe = 'Heute ist: ' + Date().toString() + '. Eine Zu-
fallszahl zwischen 0 und 10 lautet: ' + Math.round(Math.
random()*10,0).toString()
> console.log(ausgabe)
```

```
Heute ist: Thu Oct 03 2019 13:30:19 GMT+0200 (Mitteleuropäische Sommerzeit). Eine Zufallszahl zwischen 0 und 10 lautet: 9
```

## ■ Template Literals

Eine andere Methode, dasselbe zur erreichen, besteht darin, mit einem sogenannten *Template Literal* zu arbeiten. Template Literals sind Zeichenketten, die Platzhalter beinhalten. Anders als herkömmliche String werden sie in *back ticks* (`) eingeschlossen. Innerhalb eines Template Literals können dann Platzhalter eingefügt werden, die den Wert von Variablen oder anderen Ausdrücken darstellen. Platzhalter sind dadurch gekennzeichnet, dass sie mit einem Dollarzeichen (\$) beginnen und den Ausdruck, den sie repräsentieren, in geschweiften Klammern einschließen, wie im folgenden Beispiel:

```
> zufallszahl = Math.round(Math.random()*10,0)
> ausgabe = `Eine Zufallszahl zwischen 0 und 10 lautet: ${zufallszahl}.`
> console.log(ausgabe)
Eine Zufallszahl zwischen 0 und 10 lautet: 6.
```

Auf diese Weise müssen Sie Ihren String nicht mühsam zusammenbauen und dabei darauf achten, dass bei jedem Teilstring die Anführungszeichen richtig gesetzt und die Teilstrings selbst alle mit Plus-Operatoren verbunden sind. Es genügt, einfach einen langen String zu schreiben, in den Sie alles, was sie an Variablen oder anderen Ausdrücken darstellen wollen, einfach als Platzhalter mit hineinschreiben.

Wichtig dabei ist: Die Werte der Variablen werden zu dem Zeitpunkt fixiert, wenn das Template Literal erzeugt wird. Spätere Änderungen an den Werten der Variablen spiegeln sich dann nicht mehr im Template Literal wider. Dazu das folgende Beispiel:

```
> einWert = 2
> ausgabe = `Aktueller Wert: ${einWert}`
> einWert = 3
> console.log(ausgabe)
Aktueller Wert: 2
```

Ein sehr praktisches Feature von Template Literals ist übrigens, dass sie sich – anders als herkömmliche String – über mehrere Zeilen erstrecken können, wie dieses Beispiel zeigt:

```
> zweiZeilen = `Erste Zeile
Zweite Zeile`
> console.log(zweiZeilen)
Erste Zeile
Zweite Zeile
```

In einem normalen String hätten wir dazu mit der Escape-Sequenz `\n` arbeiten müssen:

```
> console.log('Erste Zeile\nZweite Zeile')
Erste Zeile
Zweite Zeile
```

### ■ String-Ersetzungen

Ein weiterer Weg, einen zusammengesetzten Output zu generieren, ist die Verwendung von String-Ersetzungen, die von `console.log()` und weiteren JavaScript-Funktionen unterstützt werden. Auch dabei wird mit Platzhaltern gearbeitet. Ein einfaches Beispiel verdeutlicht die Funktionsweise:

```
> pi = 3.14159
> console.log('Der Wert der Zahl Pi lautet: %f', pi)
3.14159
```

Der Platzhalter setzt sich hier zusammen aus dem Prozentzeichen und einer Formatierungsanweisung; `f` weist die Ausgabe an, die ausgegebene Zahl als Fließkommazahl darzustellen. Womit der Platzhalter schließlich gefüllt wird, richtet sich nach dem weiteren Argument der Funktion `console.log()`. In unserem Beispiel ist das erste Argument nach dem String unsere Variable `pi`, also wird diese für den ersten Platzhalter eingesetzt, der im String zu finden ist. Enthielte der String weitere Platzhalter, würden deren Ersetzungen als weitere Argumente hinter `pi` angefügt werden.

Analog kann ein Wert mit `%s` ein String, mit `%d` und `%i` als Ganzzahl ausgegeben werden:

```
> console.log('Der Wert der Zahl Pi lautet: %', pi)
3
```

32

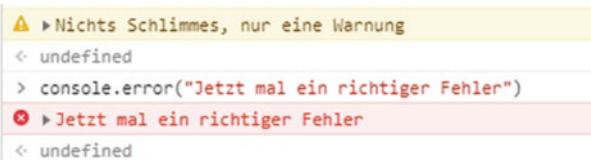
### ■ Ausgabe von Warnungen und Fehlern

Bisher haben wir Ausgaben in die Konsole stets mit `console.log()` vorgenommen. Sie können allerdings auch Warn- und Fehlermeldungen ausgeben, die dann farblich hervorgehoben und mit entsprechenden Icons versehen werden:

```
> console.warn('Nichts Schlimmes, nur eine Warnung')
> console.error('Jetzt mal ein richtiger Fehler')
```

Das Ergebnis dieser Ausgaben sehen Sie in Abb. 32.1.

■ Abb. 32.1 Selbst erzeugte Warn- und Fehlermeldungen



### 32.3 Ein- und Ausgaben über Dialogboxen

JavaScript bietet die Möglichkeit, den Browser zu veranlassen, Meldungen als kleine Dialogfenster anzuzeigen.

Mit Hilfe der Funktion **alert(nachricht)** platzieren Sie Ihre Botschaft prominent beim Benutzer:

```
> alert('Eine Wichtige Nachricht')
```

Es öffnet sich eine Dialogbox wie in □ Abb. 32.2 zu sehen.

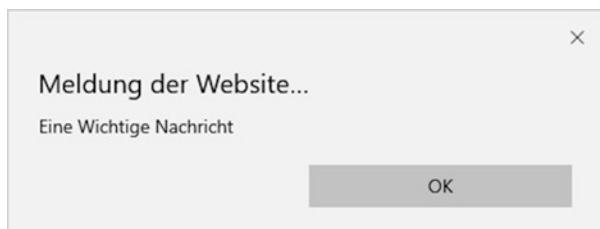
Möchte man sich vom Benutzer eine Aktion bestätigen lassen, verwendet man die Funktion **confirm(nachricht)**, die eine Meldung anzeigt und dem Benutzer die Buttons **Okay** und **Abbrechen** in einem Dialog anbietet, wie er für das Beispiel unten in □ Abb. 32.3 dargestellt ist. **confirm()** liefert den Wert **true** zurück, wenn der Benutzer auf **Okay** und **false**, wenn der Benutzer **Abbrechen** gewählt hat.

```
> aktion = confirm('Wollen Sie JavaScript wirklich weiterlernen? ')
> console.log(aktion)
true
```

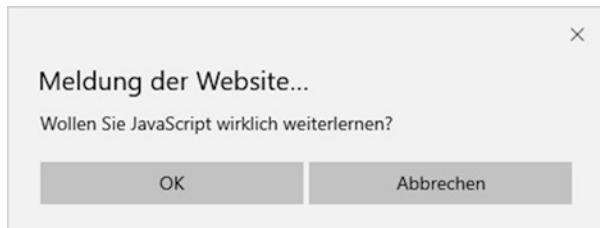
Eine weitere Möglichkeit, eine Eingabe vom Benutzer entgegenzunehmen, bietet die Funktion **prompt(nachricht)**. Sie erzeugt ein Dialogfenster, in das der Benutzer eine beliebige Eingabe machen kann, die **prompt()** als String zurückliefert (auch dann, wenn der Benutzer eine Zahl eingibt!).

Das folgende Beispiel, in dem wir auch die im vorangegangenen Abschnitt eingeführten Template Literals verwenden, dürfte bekannt vorkommen, es handelt sich um die mittlerweile bereits mehrfach bemühte Umrechnung eines Temperaturwertes von Grad Celsius in Kelvin:

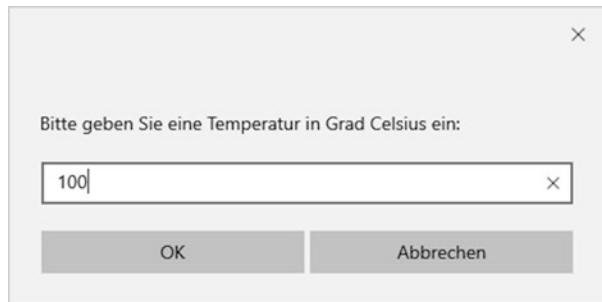
□ Abb. 32.2 `alert()`-Box in Microsoft Edge



□ Abb. 32.3 Bestätigungsdialog mit `confirm()` in Microsoft Edge



■ Abb. 32.4 `prompt()`-Eingabedialog in Microsoft Edge



```
celsius = prompt('Bitte geben Sie eine Temperatur in Grad Celsius ein: ');
console.log(`\$ {celsius} Grad Celsius sind \$ {Number(celsius)} +
273.15 Kelvin.`);
```

Der Eingabedialog, den `prompt()` erzeugt, ist in ■ Abb. 32.4 dargestellt.

Übrigens: Probieren Sie dieses Beispiel einmal aus, ohne den Aufruf des `Number()`-Konstruktors. Was passiert? Können Sie sich das Ergebnis erklären? Wenn nein, blättern Sie nochmal einige Seiten zurück zu ► Abschn. 31.3.1.

## 32.4 Ausgabe in ein HTML-Dokument / Webseite

### 32.4.1 HTML-Code in die Webseite schreiben

Nun wollen wir uns aber tatsächlich der wichtigsten Form der Ausgabe zuwenden, dem Verändern der Webseite, in die das JavaScript-Programm eingebettet ist.

32

Die einfachste Möglichkeit, eine Webseite aus JavaScript heraus zu verändern, ist die Methode `document.write(html)`. Sie schreibt einfach an der Stelle, an der das Skript in der Website eingebettet ist, den als Argument `html` übergebenen String in die Webseite. Der String, der Text ebenso wie HTML-Anweisungen (Tags) enthalten darf, wird in den Quelltext der HTML-Seite eingefügt, als wäre er vom Entwickler der Seite ursprünglich dort hineingeschrieben worden.

Um das zu verdeutlichen, gehen wir von einer (sehr minimalistischen) Webseite aus, deren Quelltext so aussieht:

```
<!DOCTYPE html>
<html>

<head>
    <title>Test-Seite</title>
    <noscript>Bitte JavaScript aktivieren!</noscript>
</head>
```

```
<body>
    <h1>Unsere Test-Seite</h1>
    <p id="ausgabe"></p>
    <script src="skript.js"></script>
</body>

<html>
```

Der Body dieser Webseite enthält nichts außer einer Überschrift (*header*, **<h1>...</h1>**), einem leeren Absatz (*paragraph*, **<p>...</p>**) und der Skript-Einbindung.

Das in die Webseite eingebettete JavaScript-Programm **skript.js** sieht wie folgt aus:

```
var zufall = Math.round(Math.random()*100, 0);
document.write(`<p>Eine Zufallszahl zwischen 0 und 100: ${zufall}.</p>`);
```

Wie Sie sehen, verwenden wir hier die aus ► Abschn. 32.2 bekannten Template Literals, um eine einfache Darstellung des in die Webseite zu schreibenden HTML-Strings zu erreichen. Stattdessen hätten wir natürlich auch **document.write("<p>Eine Zufallszahl zwischen 0 und 100: ", zufall, ".</p>")**; schreiben können, was ein wenig unübersichtlicher ist.

Wenn Sie die Darstellung der Webseite im Browser nun aktualisieren, wird der JavaScript-Code immer wieder ausgeführt und dabei jedes Mal eine neue Zufallszahl gezogen. Die Webseite, die im Browser gezeigt wird, nachdem unser Skript mit Hilfe von **document.write()** hineingeschrieben hat, sieht dann de facto so aus:

```
<!DOCTYPE html>
<html>

    <head>
        <title>Test-Seite</title>
        <noscript>Bitte JavaScript aktivieren!</noscript>
    </head>

    <body>
        <h1>Unsere Test-Seite</h1>
        <p id="ausgabe"></p>
        <p>Eine Zufallszahl zwischen 0 und 100: 62.</p>
    </body>

<html>
```

Unser Skript modifiziert die Webseite, indem es ein HTML-Element hinzufügt. Damit ergibt sich praktisch eine „neue“ Webseite, die dann im Browser dargestellt wird. Den Quelltext der Seite können Sie sich natürlich im Browser ansehen.

Nun werden Sie natürlich nicht immer an der aktuellen Stelle im Skript etwas Neues ausgeben, sondern vielleicht auch bereits vorhandene Elemente der Webseite ändern wollen. So könnten wir zum Beispiel die Überschrift der Seite ändern wollen. Dafür ist unser Skript aber „zu weit unten“ in der Webseite eingebunden, an die Überschrift kommen wir so nicht mehr heran. Es müsste also einen Weg geben, *beliebige* Elemente der Webseite von überall her zu verändern. Diesen Weg gibt es tatsächlich. Er führt über das *Document Object Model* (DOM) der Webseite, mit dem wir uns im nächsten Abschnitt beschäftigen werden.

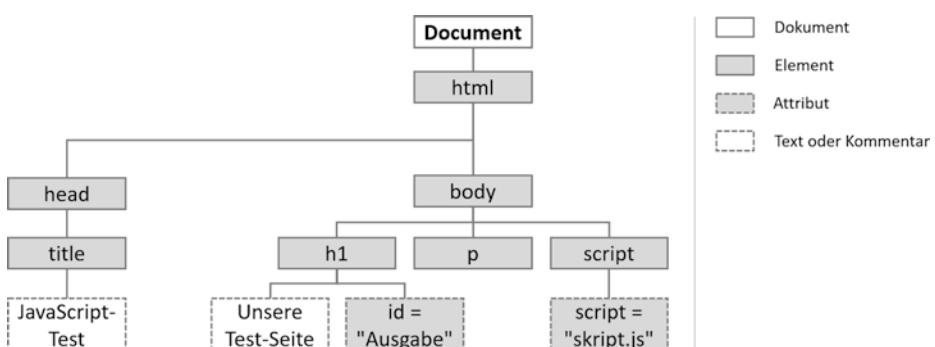
### 32.4.2 Das Document Object Model (DOM)

In der HTML-Wiederholung in ► Abschn. 29.1.1 hatten wir gesehen, dass der Webbrowsert die Webseite, also die HTML-Datei, einliest und intern in eine Darstellung überführt, die als *Document Object Model* oder kurz *DOM* bezeichnet wird. Es handelt sich dabei um eine hierarchische Darstellung der Dokumentenstruktur. Die Knoten in der Struktur sind entweder

- das Dokument selbst (höchster Knoten),
- die einzelnen HTML-Elemente wie **title**, **body**, **h1** oder **p**,
- eventueller Text, der den Elementen zugeordnet ist (die Elemente **title** und **h1** unserer Beispielwebsite aus dem vorangegangenen Abschnitt besitzen unmittelbar zugehörigen Text), oder
- die Attribute der HTML-Elemente wie das **src**-Attribut des **script**-Elements.

Bildet man die hierarchische Struktur dieser Elemente grafisch ab, ergibt sich eine Darstellung des Document Object Model der Webseite wie in □ Abb. 32.5.

Die einzelnen Knoten des Document Object-Models werden in JavaScript durch entsprechende Objekte repräsentiert. Mit Hilfe dieser Objekte können wir die entsprechenden Knoten bearbeiten und auf diese Weise etwa den Text ändern, der in der **h1**-Überschrift der Webseite dargestellt wird. Dazu müssen wir nur das richtige Element zu fassen bekommen. Ganz einfach ist das nicht, schließlich



□ Abb. 32.5 Das Document Object Model (DOM) unserer Beispielwebseite

könnte es von jedem Elementtyp (zum Beispiel eben von **h1**) mehrere Vorkommnisse im Dokument geben. Die Kunst besteht also darin, das eine Element „anzuvisieren“, das wir tatsächlich bearbeiten wollen. Genau damit beschäftigen wir uns in den folgenden Abschnitten.

Übrigens: Der höchste Knoten im Document Object Model, der das Gesamtdokument repräsentiert, wird in JavaScript durch ein Objekt abgebildet, mit dem wir im vorangegangenen Abschnitt bereits gearbeitet haben: **document**. Wie Sie sich erinnern, hatten wir dessen Methode **write()** dazu verwendet, um an der Stelle, an der das JavaScript-Programm eingebettet ist, HTML-Code in die Webseite zu schreiben.

### 32.4.3 DOM-Knoten über ihre Eigenschaften selektieren

#### ■ Ein HTML-Element über seine ID selektieren

Die einfachste Art, ein HTML-Element (also einen Knoten des Document Object Model der Webseite) zu erfassen, besteht darin, es über sein **id**-Attribut anzusprechen. In unserem Beispiel-HTML-Dokument aus ► Abschn. 32.4.1 sehen Sie ein Paragraph-Element (**p**) mit folgendem Code, das wir bislang noch nicht verwendet hatten:

```
<p id="ausgabe"></p>
```

Das Element trägt keinen Text oder weitere HTML-Elemente, aber es besitzt ein Attribut **id**, über das wir es ansteuern können. Dazu wird die Funktion **getElementById(id)** des **document**-Objekts verwendet. Die Anweisung

```
var ausgabefeld = document.getElementById('ausgabe');
```

erzeugt ein Objekt **ausgabefeld**, das mit dem **p**-Element unserer Webseite korrespondiert und über das wir das Element auf der Webseite bearbeiten können.

Dass **ausgabefeld** ein richtiges Objekt mit Eigenschaften und Methoden ist, erkennen Sie schnell, wenn Sie wie gewohnt in der JavaScript-Konsole des Browsers **ausgabefeld**. (also Objekt-Bezeichner gefolgt vom Punkt-Operator) eingeben und sich das Popup-Fenster mit den Eigenschaften und Methoden des Objekts öffnet.

Die angebotenen Eigenschaften und Methoden sind natürlich abhängig von der Art des Objekts. In unserem Beispiel haben wir es mit einem **HTMLParagraph**-Objekt zu tun. Analog gibt es eine ganze Reihe anderer Objekt-Typen für die verschiedenen HTML-Elementtypen. Jeder dieser Objekt-Typen mag seine spezifischen, für die jeweilige Art von HTML-Element relevanten Eigenschaften und Methoden mitbringen. Gemeinsam ist allen jedoch, dass sie vom Objekt-Typ **HTMLElement** abgeleitet sind und deshalb bestimmte Eigenschaften und Methoden teilen.

Alle HTML-Elemente können ein **id**-Attribut besitzen, dessen Wert – wie bei HTML-Attributen üblich – in Anführungszeichen geschrieben werden. Sicher gestellt werden muss bei der Arbeit mit den **id**-Attributen lediglich, dass jede **ID** nur einmal im Dokument vorkommt, sodass sie tatsächlich zur eindeutigen Identifizierung des jeweiligen Elements verwendet werden kann.

### ■ Ein HTML-Element über seinen Typ selektieren

HTML-Elemente können nicht nur über ihre eindeutige ID gegriffen werden, sondern auch über ihren Typ. Dazu findet die Methode **document.getElementsByTagName(typ)** Anwendung. Zu beachten ist bei diesem Vorgehen jedoch, dass natürlich ein HTML-Dokument mehr als ein Element von jedem Typ haben kann. Dementsprechend gibt **document.getElementsByTagName()** auch ein *Array* mit allen gefundenen Elementen zurück. Bei genauerem Hinsehen erkennen Sie, dass die Methode, anders als **getElementById()** auch ein Plural-s im Namen trägt – nicht ohne Grund!

Bauen wir, um das zu verdeutlichen, unser Skript aus ► Abschn. 32.4.1 so um, dass *alle p*-Elemente gegriffen werden:

```
var zufall = Math.round(Math.random()*100, 0);
document.write(`<p>Eine Zufallszahl zwischen 0 und 100: ${zufall}.</p>`);

var pElemente = document.getElementsByTagName('p');
console.log(pElemente.length);
```

Wenn Sie die JavaScript-Konsole öffnen, sehen Sie, dass als Länge des **pElemente**-Arrays **2** ausgegeben wird. Warum aber zwei Elemente, enthält unser HTML-Dokument nicht nur ein einzelnes **p**-Element? Woher kommt dann das zweite Element? Dieses zweite Element ist dasjenige, das wir durch die **document.write()**-Anweisung in unserem eigenen Skript erzeugen.

Mit diesen Elementen können wir nun arbeiten; zum Beispiel können wir mit der Eigenschaft **innerText** den Inhalt des Text-Elements, das im Document Object Model direkt an dem durch unser Skript erzeugten HTML-Element hängt, anzeigen. Dieses Element ist das zweite im Array, also dasjenige mit dem Index 1:

```
> pElemente[1].innerText
"Eine Zufallszahl zwischen 0 und 100: 62."
```

### ■ Ein HTML-Element über seine CSS-Klasse selektieren

In ähnlicher Weise wie über den Elementtyp lassen sich HTML-Elemente auch über den Wert ihres **class**-Attributs greifen. Wie Sie sich an die HTML-Wiederholung aus ► Abschn. 29.1.1 erinnern werden, können HTML-Elemente mit Hilfe ihres **class**-Attributs zu Gruppen zusammengefasst werden, für die dann

in einer Cascading-Style-Sheets-Datei (CSS) Formatierungs- und Darstellungseinstellungen vorgenommen werden können. Auf diese Weise können Formatierungs- und Darstellungsanweisungen definiert werden, die nicht auf alle Elemente eines Typs (zum Beispiel auf alle **p**-Elemente), sondern nur auf einige davon, angewendet werden.

Mit Hilfe der Methode **document.getElementsByClassName(klasse)** erhalten Sie ein Array aller HTML-Element(objekte), deren **class**-Attribut **klasse** entspricht.

Unser einfaches Beispiel arbeitet nicht mit CSS, daher besitzt keines der HTML-Elemente das **class**-Attribut.

#### 32.4.4 DOM-Knoten über die hierarchische Struktur des Dokuments selektieren

Die hierarchische Struktur des Document Object Model kann man sich zunutze machen, um ausgehend von einem Knoten andere mit ihm in Verbindung stehende Knoten zu erfassen. Dazu bieten alle Knoten-Objekte eine Reihe von vordefinierten Eigenschaften.

##### ■ Kind-Elemente eines Knotens

Das folgende Beispiel zeigt, wie wir ausgehend vom **body**-Element des HTML-Dokuments alle DOM-Knoten, die sich direkt auf der Ebene *darunter* befinden, greifen können:

```
var bodyElements = document.getElementsByTagName('body');
var bodyChildren = bodyElements[0].childNodes;
for(var i = 0; i < bodyChildren.length - 1; i++) {
    console.log('Knoten-Nr.', i)
    console.log('Knoten-Name:', bodyChildren[i].nodeName);
    console.log('Knoten-Typ:', bodyChildren[i].nodeType, '\n');
}
```

Dazu wird zunächst das **body**-Element selektieren; genauer gesagt: wir greifen mit Hilfe der **document**-Methode **getElementsByName()** alle **body**-Elemente. Natürlich gibt es in unserer Webseite davon nur eines. Trotzdem liefert **getElementsByName()** immer ein Array zurück. Auf dessen erstes Element greifen wir dann in der nächsten Zeile zu. Dabei machen wir uns dessen **childNodes**-Eigenschaft zunutze. **childNodes** ist eines der Nur-Lese-Arrays, die jeder DOM-Knoten von Haus aus besitzt, und enthält die „Kindknoten“, also jene Knoten, die hierarchisch unmittelbar auf der Ebene unter **body** hängen, deren Eltern-Element (*parent*) also **body** ist.

In einer **for**-Schleife (mit der wir uns im Detail in ► Abschn. 35.1 beschäftigen) gehen wir alle Kind-Elemente durch und geben zwei ihrer Eigenschaften in die Ja-

vaScript-Konsole aus, ihren Namen und ihren Typ. Der Output in die Konsole würde dann bei unserer Beispieldatei so aussehen:

```
Knoten-Nr. 0
Knoten-Name: #text
Knoten-Typ: 3

Knoten-Nr. 1
Knoten-Name: H1
Knoten-Typ: 1

Knoten-Nr. 2
Knoten-Name: #text
Knoten-Typ: 3

Knoten-Nr. 3
Knoten-Name: P
Knoten-Typ: 1

Knoten-Nr. 4
Knoten-Name: #text
Knoten-Typ: 3

Knoten-Nr. 5
Knoten-Name: SCRIPT
Knoten-Typ: 1
```

**32** Der Name des Knoten entspricht dem Bezeichner des HTML-Elements; bei Textknoten finden wir **#text** als Namen. Wie Sie sehen, hängen mehrere Textknoten an **body**. In unserer Beispieldatei sind im Body lediglich andere HTML-Elemente vorhanden, vor und hinter denen aber jeweils ein Text stehen könnte. Diese Texte sind in unserer Beispieldatei nur scheinbar leer. Tatsächlich enthalten sie aber Tabulatorenrückrungen, um die hierarchische Struktur des HTML-Quelltextes optisch besser zur Geltung zu bringen. Diese Tabulatorzeichen werden durch die im obigen Output zu sehenden Textknoten repräsentiert; auf ihre Darstellung haben wir der Übersicht halber bei unserem Document Object Model in Abb. 32.5 verzichtet.

Text ist natürlich auch in der **h1**-Überschrift des Dokuments enthalten („Unsere Test-Seite“), der aber steckt in einem Textknoten, der hierarchisch *eine Ebene tiefer* hängt: Er hängt nicht am **body**-Element der Webseite, sondern am **h1**-Element und wird deshalb mit unserem Durchlaufen der Kindknoten des **body**-Elements nicht erfasst.

### ■ Attribute abfragen

Gleiches gilt für die *Attribute* des **p**-Elements sowie des **script**-Elements. Sie hängen hierarchisch an dem **p**- bzw. **script**-Element und sind damit keine unmittelbaren Kinder des **body**-Elements (eher seine Enkel, sozusagen).

Bei den Attributen gibt es allerdings eine Besonderheit: Sie sind nicht im Array **childNodes** als eigene Knoten enthalten. Das können Sie leicht überprüfen, indem Sie **bodyChildren[5].childNodes** in die Konsole eingeben (der 5. Kindknoten von **body** ist das **script**-Element). **childNodes** ist leer! Das ist zunächst einmal insoweit verständlich als dass hierarchisch unter dem **script**-Element weder weitere HTML-Elemente noch Textknoten hängen. Aber es hängt das Attribut **src** (der Name der Skript-Datei) am **script**-Element, und auch dieses Attribut ist natürlich ein Knoten im Document Object Model unsere Webseite. Trotzdem ist dieser Knoten in **childNodes** nicht enthalten. Attribute werden nämlich anders abgebildet und zwar in Gestalt des Array-Objekts **attributes**. **bodyChildren[5].attributes[0]** repräsentiert damit das erste Attribut unseres **script**-Elements, also **src**. Auf den Namen des Attributes und seinen Wert kann mit den Eigenschaften **nodeName** und **nodeValue** zugegriffen werden; in unserem Beispiel also etwa mit **bodyChildren[5].attributes[0].nodeName** (das würde dann "src" liefern).

Es gibt noch eine zweite Möglichkeit, auf die Attribute eines HTML-Elements zuzugreifen. Die Attribute sind nämlich zugleich Eigenschaften des Element-Objekts. Weil **bodyChildren[5]** unser **script**-Element ist, können wir also mit **bodyChildren[5].src** direkt auf den *Wert* seines **src**-Attribut zugreifen (der Name steckt ja bereits im Eigenschaftsbezeichner). Wie in vielen Fällen, ist der Wert der **src**-Eigenschaft des Objekts **bodyChildren[5]** eine einfache Zeichenkette.

#### ■ Unterschiedliche Knoten-Typen

Wenden wir uns nochmal dem obigen Output zu. Zwei Dinge fallen noch auf: Zum einen, dass die Knoten in der Reihenfolge im Array abgebildet sind, in der sie auch im Dokument vorkommen; das ist praktisch, wenn man die Kindknoten von **body** von oben nach unten durchlauen will. Zum anderen, dass es offenbar zwei Knotentypen, nämlich 1 und 3 gibt. Knoten vom Typ 1 sind die HTML-Elemente, Knoten des Typs 3 sind die Textknoten. Attribute haben den Knotentyp 2; so hat auch das **src**-Attribut unseres **script**-Elements eine **nodeType**-Eigenschaft: **bodyChildren[5].attributes[0].nodeType**.

#### ■ Eltern-Element eines Knoten

Neben **childNodes** gibt es noch zur Navigation innerhalb der DOM-Struktur noch ein weiteres wichtiges Objekt, das eine Eigenschaft jedes DOM-Knotens ist: **parentNode**. **parentNode** ist das Gegenstück zu **childNodes**, es gibt den Knoten an, dem der aktuelle Knoten (derjenige, dessen **parentNode**-Eigenschaft wir abfragen), hierarchisch unmittelbar untergeordnet ist. Anders als **childNodes** ist **parentNode** kein Array, sondern ein einzelner Knoten, denn jeder Kindknoten hat nur genau einen Elternknoten, unter dem er hängt.

### ?

#### 32.1 [5 min]

Angenommen, Sie haben in JavaScript ein Objekt **elem**, das ein HTML-Element einer Webseite repräsentiert. Wie können Sie auf die Geschwister-Elemente von **elem** zugreifen (einschließlich **elem** selbst), also alle HTML-Elemente, die auf der gleichen hierarchischen Ebene wie **elem** im HTML-Dokument stehen?

### 32.4.5 HTML-Elemente ändern

#### ■ HTML-Code direkt in ein Objekt „injizieren“

Alle HTML-Element-Objekte besitzen in JavaScript die Objekt-Eigenschaft **innerHTML**. Sie ist eine Zeichenkette, die den gesamten HTML-Code enthält, der hierarchisch unter dem jeweiligen HTML-Element hängt.

Schauen wir uns das am Beispiel unserer Webseite aus ► Abschn. 32.4.1 einmal an. Deren body-Element hat folgende **innerHTML**-Eigenschaft:

```
> document.body.innerHTML
"
<h1>Unsere Test-Seite</h1>
<p id="ausgabe"></p>
<script src="skript.js"></script>
<p>Eine Zufallszahl zwischen 0 und 100: 31.</p>
"
```

Diese Eigenschaft können wir natürlich auch bearbeiten und so HTML-Code gewissermaßen in ein Objekt „einspritzen“.

Angenommen, wir wollten unsere Zufallszahl nicht in einem eigenen Paragraph-Element (**p**) ausgeben, das vom Skript einfach an der Stelle der Website eingefügt wird, wo es abläuft, sondern wir wollten die Zahl in das vorhandene Paragraph-Element mit der ID **ausgabe** schreiben. Unser Skript **skript.js** müsste dazu lediglich das Paragraph-Element bei seiner ID fassen und ihm dann die Ausgabe mit der Zufallszahl in dieses Element „injizieren“. Damit könnte unser Skript **skript.js** dann so aussehen:

```
zufall = Math.round(Math.random()*100, 0);
pAusgabe = document.getElementById("ausgabe");
pAusgabe.innerHTML = 'Eine Zufallszahl zwischen 0 und 100: ' +
    zufall + '.';
```

32

Vielleicht möchten wir aber auch unsere Ausgabe in Fettsatz hervorheben und sie zu diesem Zweck in ein **strong**-Element einbetten. Dazu müssten wir die letzte Zeile unseres Skripts lediglich folgendermaßen anpassen:

```
pAusgabe.innerHTML =
'<strong>Eine Zufallszahl zwischen 0 und 100: ' +
    zufall + '</strong>';
```

Wenn Sie sich jetzt den Quelltext der im Browser angezeigten Seite ansehen, oder aber einfach mit **document.body.innerHTML** den HTML-Inhalt des **body**-Elements, werden Sie darin folgendes finden:

```
"  
    <h1>Unsere Test-Seite</h1>  
    <p id="ausgabe"><strong>Eine Zufallszahl zwischen 0 und  
    100: 33.</strong></p>  
    <script src="skript.js"></script>  
"
```

Wie Sie sehen, haben wir also tatsächlich erfolgreich einen HTML-Code-Schnipsel ins Innere des **p**-Elements **ausgabe** „injiziert“.

#### ■ Attribute von Elementen ändern

Neben der Verwendung des **b**-Elements gibt es eine weitere Möglichkeit, Text in Fettsatz ausgeben zu lassen, und das ist mit Hilfe der CSS-Eigenschaft **font-weight**. Setzt man diese auf **bold**, wird der Text ebenfalls in Fettsatz dargestellt.

Die CSS-Eigenschaften eines HTML-Elements können entweder über eine separate CSS-Datei beschrieben werden (ein Weg, den wir in einem der beiden Beispiele, mit denen wir dieses Kapitel beschließen, gehen werden), oder aber direkt mit Hilfe des **style**-Attributs eines HTML-Elements gesetzt werden. Wollten wir also den gesamten Inhalt unseres Paragraph-Elements **ausgabe** in Fettsatz darstellen, müssten wir das öffnende HTML-Tag folgendermaßen anpassen:

```
<p id="ausgabe" style="font-face: bold;">
```

Der Namen einer Eigenschaft und ihr Wert werden in CSS stets durch einen Doppelpunkt voneinander getrennt. Das abschließende Semikolon hätten wir streng genommen nicht benötigt. Allerdings ist es möglich, in das **style**-Attribut mehrere CSS-Eigenschaftzuweisungen unterzubringen, die dann mit Semikolons voneinander getrennt werden müssen. Es schadet also nicht, schon mal ein Semikolon am Ende des **style**-Attributs stehen zu haben.

Wie aber können wir nun mit Hilfe unseres JavaScript-Codes das **style**-Attribut setzen? Die Herangehensweise dürfte Ihnen nach dem vorangegangenen Abschnitt nicht mehr überraschend vorkommen: Wir wissen ja bereits, dass das HTML-Elementobjekt (das wir uns im Skript oben mit **getElementById** an die Variable **pAusgabe** gebunden haben) für jedes Standard-Attribut eine passende Eigenschaft hat. Und die können wir natürlich auch zuweisen. Dabei ist zu beachten, dass **style** ein Objekt mit zahlreichen Eigenschaften ist, die die CSS-Eigenschaften widerspiegeln. Die Namen der Eigenschaften gleichen den CSS-Eigenschaften, wobei der CSS-typische Bindestrich entfällt und durch Großschreibung ersetzt wird. Aus **font-weight** wird so **fontWeight**:

```
pAusgabe.style.fontWeight = 'bold';
```

Wenn Sie jetzt – damit wir die Wirkung auch beobachten können – die „Injektion“ des HTML-Codes wieder zurücksetzen auf

```
zufall = Math.round(Math.random()*100, 0);
pAusgabe = document.getElementById("ausgabe");
pAusgabe.innerHTML = 'Eine Zufallszahl zwischen 0 und 100: ' +
    zufall + '.';
```

und die Webseite im Browser neu laden, werden Sie feststellen, dass die Operation erfolgreich war und die Ausgabe tatsächlich in Fettsatz dargestellt wird.

Nun ist das **style**-Attribut insofern ein spezieller Fall, weil der **style**-Eigenschaft des HTML-Elementobjekts in JavaScript nicht einfach ein Wert wie "**font-face: bold;**" zugewiesen werden kann, sondern stattdessen mit den einzelnen Eigenschaften des **style**-Objekts gearbeitet werden muss.

Zahlreiche Attribute aber besitzen im HTML-Element-Objekt in JavaScript eine korrespondierende Eigenschaft, die direkt zugewiesen werden kann. Das **align**-Attribut etwa, dass die Textrichtung bestimmt, könnte mit der Anweisung

```
pAusgabe.align = 'right';
```

so gesetzt werden, dass der Text innerhalb des Absatzes rechtsbündig dargestellt wird (probieren Sie es aus!). Anders als **style** ist die korrespondierende Eigenschaft des HTML-Elementobjekts (in unserem Fall die **align**-Eigenschaft von **pAusgabe**) eben nicht wiederum ein Objekt, sondern eine einfache Zeichenkette, die sich leicht zuweisen lässt. Die Anweisung führt also dazu, dass der Code des **p**-Elements nun folgendermaßen aussieht:

```
<p id="ausgabe" align="right">
```

32

### 32.2 [10 min]

Erzeugen Sie eine einfache Webseite mit einem **p**-Element (**paragraph**), in dem ein Text steht. Ändern Sie aus einem JavaScript-Programm heraus die Schrift-Hintergrundfarbe (CSS-Eigenschaft **background-color**) so, dass der Text hellgelb (Farb-Anteile rot: 255, grün: 255, blau: 204) hervorgehoben ist.

#### 32.4.6 HTML-Elemente hinzufügen und löschen

##### ■ HTML-Elemente hinzufügen

Im letzten Abschnitt haben wir mit Hilfe der **innerHTML**-Eigenschaft von HTML-Elementobjekten HTML-Code direkt in die Elemente „eingespritzt“ und konnten auf diese Weise auch neue HTML-Objekte als Kind-Elemente der bearbeiteten Elemente Eltern-Elemente erzeugen, indem wir ihren vollständigen HTML-Code einfach in das Eltern-Element hineingeschrieben haben.

In diesem Abschnitt widmen wir uns einer weiteren Methode, HTML-Elemente zu erzeugen, und zwar dadurch, dass wir entsprechende HTML-Elementobjekte in JavaScript generieren und diese dann an der gewünschten Stelle in das Dokument „ehängen“.

Neue HTML-Elementobjekte lassen sich in JavaScript leicht mit Hilfe der Funktion `createElement()` des `document`-Objekts erzeugen. Als Argument erwartet diese Funktion den Bezeichner des HTML-Elementtyps, von dem eine Instanz erzeugt werden soll.

Um also ein beispielsweise eine neue Unterüberschrift (HTML-Elementtyp `h2`) zu erzeugen, genügt die folgende Anweisung:

```
var ueberschrift2 = document.createElement('h2');
```

Nun können wir die Eigenschaften des neuen Elements wie gewünscht bearbeiten, beispielsweise, indem wir mit Hilfe des `align`-Attributs den Überschriftentext rechtsbündig machen:

```
ueberschrift2.align = 'right';
```

Einen Text benötigt die Überschrift natürlich auch. Den könnten wir mit der bereits bekannten `innerHTML`-Eigenschaft unseres Elementobjekts setzen, oder aber auch die Eigenschaft `innerText` verwenden. `innerText` repräsentiert normalerweise den gesamten Text, der in Textknoten an dem Element selbst oder seinen Kindern hängt; das wird deutlich, wenn Sie einmal die `innerText`-Eigenschaft des `body`-Elements unserer Webseite betrachten. Wir können aber `innerText` natürlich auch dazu nutzen, unserem neu erzeugten Element (das ja gar keine Kind-Elemente besitzt) einen Text mitzugeben:

```
ueberschrift2.innerText = 'Ein weiterer spannender Abschnitt';
```

Nachdem wir unser neues HTML-Elementobjekt ausreichend konfiguriert haben, müssen wir es noch an der Stelle, an der wir es haben wollen, in die Webseite einbinden. Das geht am einfachsten, indem wir das angedachte Eltern-Element greifen und diesem dann das neue Kind-Element mit Hilfe der Elementobjekt-Methode `appendChild(neuesKind-Element)` hinzufügen:

```
var bodyElem = document.getElementsByTagName('body')[0];
bodyElem.appendChild(ueberschrift2);
```

Alternativ können Sie, wenn sie das neue Element nicht einfach an die vorhandene Kind-Elemente hinten anhängen wollen, die Position auch genauer steuern, indem Sie das Kind-Element mit der Elementobjekt-Methode `insertBefore(neues-`

**Kind-Element, nachfolgerKind-Element)** platzieren. Auf diese Weise könnten wir unsere neue Überschrift vor das Paragraph-Element **ausgabe**, das wir in unserem Programm durch das Objekt **pAusgabe** repräsentieren, einfügen:

```
bodyElem.insertBefore(ueberschrift2, pAusgabe);
```

### 32.3 [5 min]

Entwickeln Sie ein Stück JavaScript-Code, das Sie in der JavaScript-Konsole ausführen können und das dem HTML-**body** einer Webseite den in fettener Schrift formatierten Text „Hier ist Schluss.“ hinzufügt (am Ende der Webseite).

Probieren Sie es aus: Nehmen Sie eine Webseite wie *wikipedia.de*, öffnen Sie die Entwickertools im Browser und führen Sie Ihren Code aus. Sie werden sehen, dass die Webseite um Ihr HTML-Element ergänzt wurde!

#### ■ HTML-Elemente löschen

Das Löschen von HTML-Elementen aus der Webseite lässt sich bequem mit der Methode **remove()**, die alle HTML-Knotenobjekte in JavaScript von Haus aus mitbringen, bewerkstelligen. Um also etwa das Paragraph-Element **ausgabe** zu löschen, müssen wir lediglich **remove()** für das Objekt **pAusgabe** aufrufen:

```
pAusgabe.remove()
```

Das Element verschwindet sofort von der Webseite. Ebenso wie beim Hinzufügen und Ändern von Objekten generiert der Browser die Darstellung neu, ohne dass der Benutzer oder wir als Entwickler etwas tun müssten.

## 32

### 32.5 Eingabe mit Formularen

Im Folgenden werden wir uns damit beschäftigen, wie JavaScript verwendet werden kann, um die Eingaben aus HTML-Formularen zu validieren oder anderweitig zu verarbeiten. Der nächste Abschnitt bietet zunächst einen kurzen Überblick über die Funktionsweise von Formularen in HTML. Wenn Sie damit bereits vertraut sind, können Sie – ohne etwas zu verpassen – direkt zum übernächsten Abschnitt wechseln.

#### 32.5.1 Formulare in HTML

Mit Ausnahme der Dialogboxen haben wir uns bislang vor allem damit befasst, wie Daten in Webseiten *ausgegeben* werden können, nicht jedoch damit, wie der Benutzer Daten *eingeben* kann.

Genau das ist der Zweck von HTML-Formularen: Sie dienen dazu, Daten vom Benutzer entgegen zu nehmen. Die entgegengenommenen Daten werden häufig an den Webserver, der die Seite bereitstellt, geschickt und dort verarbeitet. Dabei kommt oft die Programmiersprache PHP zum Einsatz, die genau zu diesem Zweck entwickelt worden ist, teilweise aber auch serverseitig ablaufende JavaScript-Programme. In solchen Client-Server-Situationen fällt JavaScript meist die Aufgabe zu, die Eingaben des Benutzers auf der Client-Seite vor dem Abschicken an den Webserver zu validieren, das heißt, auf etwaige Fehleingaben hin zu prüfen, ggf. das Versenden der Daten aufzuhalten und den Benutzer auf die Fehleingaben hinzuweisen.

Die Daten müssen aber nicht zwingend an einen Webserver versandt werden. Tatsächlich können die Formulareingaben natürlich auch als Input für eine JavaScript-Anwendung verwendet werden; diese Anwendung wäre dann kein Validierungsmechanismus, sondern gewissermaßen der endgültige Empfänger der Daten. Die Daten würden nur zu dem Zweck eingegeben, durch das JavaScript-Programm verarbeitet zu werden. Die beiden Beispiele, mit denen dieses Kapitel schließt, ein Taschenrechner und ein Color Picker, fallen genau in die Kategorie von Zusammenspiel zwischen Formular und JavaScript.

Bevor wir uns allerdings mit diesen Anwendungen beschäftigen, schauen wir uns zunächst an, wie Formulare in HTML aufgebaut werden.

Formulare werden stets durch das HTML-Element **form** erzeugt. Damit entsteht ein zunächst noch leeres Formular. Innerhalb des **form** können dann beliebig viele unterschiedliche **input**-Elemente stehen, die verschiedene Eingabemöglichkeiten abbilden. Welchen Typ von Eingabe das jeweilige Element darstellt, wird über das **input**-Attribut **type** gesteuert. Betrachten Sie als Beispiel das folgende einfache (Login-)Formular:

```
<form action="http://www.meinenettewebseite.com/login.php"
      method="POST">
    Benutzername: <br>
    <input type="entry" value="" name="username"><br>
    Passwort: <br>
    <input type="password" value="" name="password"><br>
    <input type="submit" value="Login">
</form>
```

In diesem Beispiel erzeugen wir ein Formular mit insgesamt drei Eingabeelementen:

- einem Element vom Typ **entry** für den Benutzernamen; dabei handelt es sich um ein einfaches Texteingabefeld,
- einem Element vom Typ **password**, was letztlich eine spezielle Variante des Typs **entry** ist, bei der die eingegebenen Zeichen durch Sternchen maskiert werden, und
- einem Element vom Typ **submit**, einem speziellen Button, der den Formularinhalt an den Server absendet.

Neben diesen gibt es eine ganze Reihe weiterer Typen von input-Elementen, zum Beispiel **button** („normale“ Buttons, der **submit**-Typ hat ja als Button eine ganz spezielle Funktion), **radio** (Radiobuttons), **checkbox** (Checkboxen), **range** (Schieberegler) oder **textarea** (für mehrzeilige Texteingaben). Daneben gibt es Eingabeelemente, die die Auswahl von Datumsangaben (**date**), Farben (**color**) oder hochzuladenden Dateien (**file**) erlauben, sowie eine ganze Reihe weiterer Elemente, die andere Modi der Eingabe unterstützen.

Alle Elemente haben ein **value**-Attribut, das ihren aktuellen Wert enthält; bei den Texteingabefeldern ist das der aktuell im Eingabefeld befindliche Text, beim **submit**-Button ist es die Beschriftung des Buttons. Dieses Attribut ist wichtig, weil sich darüber die Eingabe des Benutzers, die ja letztlich verarbeitet werden soll, ermitteln lässt. Neben **value** können die Eingabeelemente auch ein **name**-Attribut besitzen (wie im Übrigen alle HTML-Elemente!). Das ist hilfreich, wenn man serverseitig auf die Eingabewerte zugreifen möchte. Arbeiten wir dagegen nur auf Client-Seite mit JavaScript, können wir die Elemente natürlich auch wie gewohnt über ein **id**-Attribut ansteuern, das im obigen Beispiel der Einfachheit halber weggelassen worden ist. Der **name** dient auch dazu, diejenigen Elemente zu gruppieren, die logisch zusammen ausgewertet werden müssen; das gilt vor allem für die einzelnen Auswahloptionen bei einer Gruppe von Radiobuttons, bei denen ja jeweils nur eine Option auswählbar ist (das werden wir uns in einem Beispiel weiter unten genauer ansehen).

Das Verhalten der Eingabeelemente kann mit einer Reihe weiterer Attribute feiner gesteuert werden; das Attribut **required** verhindert beispielsweise, dass das Formular abgesendet wird, wenn das betreffende Feld nicht gefüllt ist, das Attribut **readonly**, dass der Benutzer Änderungen am aktuellen Wert des Eingabeelements vornimmt. Beide Attribute sind vom Typ **boolean**, ihre möglichen Werte daher **true** und **false**. Statt etwa **required="true"** sieht man aber in der Praxis oft einfach nur **required**. Die schiere Existenz des Attributs wird bereits als **true** ausgewertet und würde in diesem Fall genügen, das betreffende Eingabeelement zur einer Pflichtangabe zu machen.

Neben diesen Standardattributen können die verschiedenen Eingabeelemente noch weitere Typ-spezifische Attribute besitzen. Das **range**-Eingabeelement, das einen Schieberegler darstellt, besitzt zum Beispiel die Attribute **min** und **max**, die die Grenzen des Bereichs beschreiben, innerhalb dessen mit Hilfe des Reglers ein Wert ausgewählt werden kann (welcher dann wiederum im **value**-Attribut abgefragt werden kann); Checkboxen haben mit **checked** ein **boolean**-Attribut, das festlegt, ob die Checkbox derzeit markiert sein soll oder nicht.

Bislang noch nicht besprochen haben wir die Attribute des **form**-Elements selbst. Das Attribut **action** legt fest, welche Adresse aufgerufen werden soll (in der Regel ein PHP-Skript), um die eingelesenen Daten an den Server zu übergeben, wenn der Benutzer das Absenden des Formulars über den **submit**-Button auslöst. Die **method** bestimmt, nach welchem Modus die Daten über das *Hypertext Transfer Protocol* (HTTP) übertragen werden sollen. Der Standardwert dieses Attributes ist **GET**, gerade dann aber, wenn sensible Daten wie Passwörter übertragen werden sollen, wird üblicherweise **POST** verwendet.

Die Attribute **action** und **method** des **form**-Elements, sowie den **submit**-Button, mit dem die Daten abgesendet werden können, werden allerdings nur benötigt, wenn die eingegebenen Daten auch tatsächlich an einen Webserver übertragen werden sollen. Das Color-Picker-Beispiel am Ende des Kapitels kommt vollständig ohne jedwede Form von Buttons aus, das Taschenrechner-Beispiel benutzt zwar Buttons, aber keinen **submit**-Button. Beide Anwendungen verarbeiten die Daten direkt in einem client-seitigen JavaScript-Programm und können deshalb auf Vorehrungen zum Versenden der Daten verzichten.

### 32.5.2 Formulare aus JavaScript heraus ansteuern

Aus unseren JavaScript-Programmen heraus wollen wir regelmäßig mit den Daten, die der Benutzer in ein Formular eingegeben hat, arbeiten. Dazu müssen wir zum einen aus JavaScript heraus auf die Formularelemente zugreifen, um an die darin vorhandenen Daten zu gelangen, zum anderen aber auch einen Weg finden, diesen Zugriff (und die sich daran anschließende Verarbeitung der Daten) mit einer Aktion des Benutzers zu verknüpfen. Schließlich sind wir im Bereich der Weboberflächen ja auch in einer *ereignisgesteuerten* Umgebung unterwegs, in der der Benutzer Ereignisse auslöst (zum Beispiel, indem er auf einen Button klickt) und unsere JavaScript-Anwendung darauf reagiert.

Um die Vorgehensweise bei der Ereignissteuerung und dem Zugriff auf die Formulardaten kennenzulernen, betrachten wir ein mittlerweile wohlbekanntes Beispiel, die Temperaturumrechnung zwischen Grad Celsius und Kelvin.

Diese Anwendung könnte eine Oberfläche besitzen, in der der Benutzer eine Temperatur eingibt und entscheidet, ob er diese Temperatur in Grad Celsius (dann hatte eine Kelvin-Temperatur eingegeben) oder in Kelvin (dann war die eingegebene Temperatur eine Celsius-Temperatur angegeben) umrechnen möchte. Ein Klick auf einen Button startet die Umrechnung und gibt das Ergebnis aus.

Der HTML-Code einer solchen Oberfläche, wie sie auch in Abb. 32.6 gezeigt ist, sieht so aus:

## Temperatur-Umrechnung Kelvin <=> Grad Celsius

Temperatur zu Umrechnung:  Celsius

Umrechnen in:

- Grad Celsius
- Kelvin

Abb. 32.6 Kelvin-Celsius-Umrechnung mit HTML-Formular

```

<!DOCTYPE html>
<html>

<head>
    <title>Temperatur-Umrechnung</title>
    <noscript>Bitte JavaScript aktivieren!</noscript>
</head>

<body>
<script src="kelvincelsius.js"></script>

<h1>Temperatur-Umrechnung Kelvin <=> Grad Celsius </h1>
<form>
    <p>Temperatur zu Umrechnung:
        <input id="temp"
               type="text" value=""
               size="5">
        <span id="einheitLabel"> Kelvin</span>
    </p>
    Umrechnen in:<br>
    <p><input type="radio" name="richtung" checked
               onchange="aendern(' Kelvin')">Grad Celsius</p>
    <p><input type="radio" name="richtung"
               onchange="aendern(' Celsius')">Kelvin</p>
    <p></p>
    <input type="button" value="Umrechnen"
           onclick="umrechnen()">
</form>
</body>

</html>

```

## 32

Das Formular umfasst eine Texteingabe mit der ID **temp** für *Temperatur*, zwei Radiobuttons (**richtung**) zur Festlegung der Umrechnungsrichtung und einen Button, der die Umrechnung auslöst.

Auch sehen Sie ein HTML-Element, das wir bisher noch nicht kennengelernt haben, nämlich **span**. **span** hat keine besondere Funktion, aber es hilft uns, eine Textanzeige mit einer eigenen ID zu versehen, um ihn aus unserem JavaScript-Programm heraus ansprechen zu können. Auf unserem **span**-Element **einheitLabel** stellen wir nämlich die Einheit dar, in der der Benutzer die Temperatur eingibt. Jedes Mal, wenn er die Auswahl bzgl. der Umrechnungsrichtung ändert, muss sich auch diesen Einheitenanzeige ändern.

Die beiden Radiobuttons haben wir über ihr **name**-Attribut zusammengefasst: Sie gehören damit zu einer Gruppe, es kann also jeweils nur eines von beiden markiert sein. Beachten Sie bitte, dass die Gruppierung der Radiobuttons über das **name**-Attribut und nicht über das **id**-Attribut realisiert wird. Der Unterschied besteht darin, dass die ID tatsächlich eine *eindeutige* Identifizierung darstellt. Dann aber darf es keine zwei Elemente mit der gleichen ID geben. Unsere beiden Radio-

buttons besitzen gar keine ID, denn wir können Sie auch über den Namen aus unserem JavaScript-Programm heraus ansprechen. Der Button, der die Umrechnung startet, besitzt sogar weder eine ID noch einen Namen. Beides ist nicht notwendig, da der Button zwar eine Aktion auslöst und damit unser JavaScript-Programm anstößt, wir aus dem Programm heraus aber nicht auf ihn zugreifen müssen.

Rein designerische Funktion haben die **p**- und **br**-Elemente (**br** erzeugt einen einfachen Zeilenumbruch, *(line) break*), sie helfen uns, das Formular optisch etwas ansprechender zu gestalten. Wie Sie sehen, können in einem **form**-Element also nicht nur **input**-Elemente platziert werden, sondern Sie können die ganze HTML-Elemente-Vielfalt (einschließlich Tabellen, Bildern u.ä.) nutzen, um ein attraktives Formular zu bauen.

Einen ganz wichtigen Bestandteil des Formulars haben wir bislang aber noch nicht betrachtet: Die **onchange**- und **onclick**-Attribute der Radiobuttons **richtung** und des Buttons **Umrechnen**. Der Wert jedes dieser Attribute ist jeweils eine JavaScript-Funktion (ein sogenannter *Event Handler*), der Name des Attributs hängt mit einem Ereignis zusammen, und zwar mit demjenigen Ereignis, bei dessen Auftreten die jeweilige JavaScript-Funktion aufgerufen werden soll. Klickt der Benutzer zum Beispiel auf unseren Button, wird das **click**-Ereignis ausgelöst. Beim **click**-Ereignis prüft der Browser automatisch, ob das Attribut **onclick** gesetzt und führt, wenn das der Fall ist, die dort angegebene JavaScript-Funktion aus. Auf diese Weise verknüpfen wir unsere Oberfläche mit dem JavaScript-Programm.

Dieses sieht in unserem Beispiel folgendermaßen aus:

```
function umrechnen() {
    var temp = Number(
        document.getElementById('temp').value);
    var richtung =
        document.getElementsByName('richtung');

    if (richtung[0].checked == true) {
        document.write(
            `<p>${temp} Kelvin in Grad Celsius sind: ${temp -
            273.15} Grad Celsius.<p>`);
    }
    else {
        document.write(
            `<p>${temp} Grad Celsius in Kelvin sind: ${temp +
            273.15} Kelvin.<p>`);
    }
}

function aendern(einheit) {
    var einheitLabel = document.getElementById(
        'einheitLabel');
    einheitLabel.innerHTML = einheit;
}
```

Das ganze Programm besteht nur aus zwei Funktionen, und zwar den beiden Event Handlern, die mit den Radiobuttons und dem Hauptbutton unserer Anwendung verknüpft sind. Die Funktion **andern()** ist bei den **onchange**-Attributen unserer Radiobuttons als Event Handler hinterlegt und wird deshalb stets aufgerufen, wenn das **change**-Ereignis auftritt. Das geschieht immer dann, wenn der Benutzer einen der Radiobuttons anklickt (wir hätten daher übrigens ebenso gut unseren Event Handler an das **click**-Ereignis hängen können). Tritt das Ereignis auf, wird die Funktion mit einem Argument aufgerufen, und zwar der Einheit, die auf unserem **span**-Element **einheitLabel** als Einheit für die Benutzereingabe dargestellt werden soll. Im HTML-Code sehen Sie sehr schön, dass dem Event Handler beim Aufruf gleich die gewünschte Einheit als Parameter mitgegeben wird: **onchange="andern(' Kelvin')"**.

Innerhalb des Event Handlers **andern()** selektieren wir zunächst mit **document.getElementById()** das **span**-Element und tauschen dann den HTML-Code in seinem Inneren aus; in unserem Beispiel ist das ohnehin nur reiner Text ohne weitere HTML-Codierung.

Nun aber zu unserem anderen Event Handler, **umrechnen()**, der immer dann aufgerufen wird, wenn der Benutzer auf den Button **Umrechnen** klickt. Er ist denkbar einfach aufgebaut. Zunächst ermitteln wir die Temperatur, indem wir das **value**-Attribut des Eingabefeldes **temp** abfragen, das wir über seine ID selektieren. Beachten Sie dabei, dass wir eine Umwandlung in eine **number**-Variable vornehmen müssen, denn wir wollen mit dem Temperatur-Wert ja rechnen. Das Formular selbst speichert den eingegebenen Wert grundsätzlich als **string**; *grundsätzlich* deshalb, weil es durchaus Möglichkeiten gibt, Formular-Eingabefelder so zu konfigurieren, dass sie von vorneherein nur numerische Eingaben zulassen, was wir aber der Einfachheit halber hier nicht getan haben. Als nächstes selektieren wir die Radiobuttons anhand ihres Namens. Dazu verwenden wir die Funktion **getElementsByName()**. Achten Sie auf das Plural-s bei **Elements**! Da der Name – anders als die ID – nicht zwangsläufig eindeutig ist, kann es durchaus vorkommen, dass man bei der Selektion über den Namen mehrere Elemente zu fassen bekommt. Und genau so ist es in unserem Beispiel ja auch. Rückgabewert des Aufrufs von **getElementsByName()** ist ein *Array* von Elementen, in unserem Beispiel den beiden Radiobuttons. Im nächsten Schritt prüfen wir mit Hilfe des **checked**-Attributs der Radiobuttons, ob unser erster Radiobutton (Index **0**) markiert ist. Machen Sie sich an dieser Stelle noch nicht so viele Gedanken um die **if-else**-Konstruktion, mit dieser Art der Programmverzweigung beschäftigen wir uns ausführlich in Abschn. 34.1. Ist unser erster Radiobutton markiert, bedeutet das, der Benutzer unseres Programms wünscht eine Umrechnung von Kelvin in Grad Celsius. Die geben wir dann in der Webseite mit **document.write()** aus und zwar mittels einer Template Literals (wenn Ihnen Template Literals nicht mehr geläufig sind, blättern Sie nochmal einige Seiten zurück zu ► Abschn. 32.2).

Sie sehen also, dass unser JavaScript-Programm ausschließlich aus Event Handlern besteht, die gewissermaßen im Verborgenen schlummern, bis sie vom Browser ausgelöst werden, weil ein Ereignis eingetreten ist, mit dem sie verknüpft sind. Mit Ereignissen beschäftigen wir uns noch etwas eingehender in ► Abschn. 34.3, wenn es um die Ablaufsteuerung von Programmen geht. An dieser Stelle genügt es, den grundlegenden Mechanismus zu verstehen, mit dem wir unseren JavaScript-Code mit den Bedienelementen der Oberfläche „verdrahten“ können.

#### ?

#### 32.4 [5 min]

Ändern Sie das Kelvin-Celsius-Umrechnungsbeispiel so, dass die Ausgabe nicht mit **document.write()** erfolgt, sondern auf ein HTML-Element vom Typ **span**, dass dazu in die Webseiten-Oberfläche eingebaut werden muss.

#### ?

#### 32.5 [30 min]

Entwickeln Sie eine einfache Anwendung, in der man mit einem Schieberegler die Schriftgröße eines auf der Webseite der Anwendung angezeigten Beispieldesetzes ändern kann. Die Schriftgröße lässt sich in HTML/CSS über das über die CSS-Style-Option **fontsize** einstellen und wird in Pixeln (**px**) angegeben; eine gültige Schriftgrößeneinstellung wäre damit beispielsweise **'18px'**.

Wenn Sie denken, dass Sie noch etwas „Anschauungsmaterial“ benötigen, bevor Sie sich dieser Aufgabe zuwenden, beschäftigen Sie sich zunächst mit den Beispielen in den folgenden beiden Abschnitten.

## 32.6 Beispiel: Einfacher Taschenrechner

---

In diesem und dem nächsten Abschnitt werden wir zwei einfache Beispielanwendungen entwickeln. Zunächst wenden wir uns einem einfachen Taschenrechner zu.

Der Taschenrechner soll die vier Grundrechenarten beherrschen und es erlauben, das Ergebnis der Berechnung in die Zwischenablage zu kopieren. Die Eingabe der Zahlen und Operatoren soll wahlweise über Buttons oder durch direkte Eingabe über die Tastatur erfolgen.

Unsere Anwendung besteht aus drei Dateien:

- der HTML-Datei **taschenrechner.html**, die die Web-Oberfläche aufbaut,
- der Cacading-Style-Sheet-(CSS-)Datei **taschenrechner.css**, die uns hilft, das Design der Buttons und des Displays festzulegen, sowie
- dem JavaScript-Programm **taschenrechner.js**, das die Funktionalität für die Oberfläche bereitstellt.

### 32.6.1 Die Web-Oberfläche

Beginnen wir mit der HTML-Datei **taschenrechner.html**:

```
1  <!DOCTYPE html>
2  <html>
3
4  <head>
5      <title>Taschenrechner</title>
6      <link rel="stylesheet" type="text/css"
7          href="taschenrechner.css">
8      <noscript>Bitte JavaScript aktivieren!</noscript>
9  </head>
10
11 <body bgcolor="#282923">
12 <script src="taschenrechner.js"></script>
13
14 <form>
15     <input id="display" type="text" value="0"
16         class="inputOutput">
17     <p></p>
18     <input type="button" value="C"
19         class="normalButton functionButton"
20         onclick="loeschen()">
21     <input type="button" value="Kopieren"
22         style="width:104px"
23         class="normalButton functionButton"
24         onclick="kopieren()">
25     <input type="button" value="/"
26         class="normalButton functionButton"
27         onclick="taste('/')">
28     <p></p>
29     <input type="button" value="7" class="normalButton"
30         onclick="taste('7')">
31     <input type="button" value="8" class="normalButton"
32         onclick="taste('8')">
33     <input type="button" value="9" class="normalButton"
34         onclick="taste('9')">
35     <input type="button" value="*"
36         class="normalButton functionButton"
37         onclick="taste('*')">
38     <p></p>
39     <input type="button" value="4" class="normalButton"
40         onclick="taste('4')">
41     <input type="button" value="5" class="normalButton"
42         onclick="taste('5')">
43     <input type="button" value="6" class="normalButton"
44         onclick="taste('6')">
45     <input type="button" value="-"
46         class="normalButton functionButton"
47         onclick="taste('-')">
```

```
48      <p></p>
49      <input type="button" value="1" class="normalButton"
50          onclick="taste('1')">
51      <input type="button" value="2" class="normalButton"
52          onclick="taste('2')">
53      <input type="button" value="3" class="normalButton"
54          onclick="taste('3')">
55      <input type="button" value="+" 
56          class="normalButton functionButton"
57          onclick="taste('+')">
58      <p></p>
59      <input type="button" value="0" class="normalButton"
60          style="width:104px" onclick="taste('0')">
61      <input type="button" value="." class="normalButton"
62          onclick="taste('.')">
63      <input type="button" value="-"
64          class="normalButton functionButton"
65          onclick="rechnen() ">
66  </form>
67  </body>
68
69  </html>
```

### ■■ Zeilen 1–9.

Der übliche HTML-Header, in dem wir zunächst die CSS-Datei **taschenrechner.css** einbinden und mit Hilfe des **noscript**-Elements eine Vorkehrung für den Fall treffen, dass der Benutzer JavaScript in seinem Browser deaktiviert hat. Die Einbindung der CSS-Datei erfolgt mit dem **link**-Element.

### ■■ Zeile 11.

Wir setzen den Hintergrund der Webseite durch das **bgcolor**-Attribut auf einen dunklen Farbton, damit unser Taschenrechner auch stylisch aussieht. Das Auge rechnet schließlich mit!

### ■■ Zeile 12.

Wir binden das Skript **taschenrechner.js** ein, dass die eigentliche Funktionalität der Seite beinhaltet. Es wird zwar technisch gesehen an dieser frühen Stelle beim Laden der Webseite ausgeführt, allerdings besteht es – wie wir unten noch sehen werden – lediglich aus Funktionen, die dann ereignisgesteuert durch die einzelnen Buttons aufgerufen werden. Solange diese Funktionen nicht explizit aufgerufen werden, passiert beim Ausführen des Skripts anwendungsseitig überhaupt nichts. Ebenso gut hätten wir das Skript auch im **body**-Segment unserer Webseite einbinden können. Die Funktionalität der Anwendung würde das nicht beeinträchtigen.

### ■■ Zeilen 14 und 66.

Der Rest des **body**-Segments unserer Seite ist ein HTML-Formular, das alle Steuer- elemente des Taschenrechners enthält.

### ■■ Zeilen 15–16.

Hier definieren wir das Display unseres Taschenrechners. Wir geben ihm die **id "display"**, sodass wir es später aus unserem JavaScript-Programm heraus ansprechen können. Sein Typ ist **text**, es handelt sich also um ein Eingabefeld, immerhin soll der Benutzer ja auch Zahlen und Operatoren über die Tastatur eingeben können; in unserem Taschenrechner kann er also direkt in das Display hineintippen. Der initiale Wert soll **0** sein, solange der Benutzer noch nichts anderes eingegeben hat. Außerdem geben wir unserem Display noch mit Hilfe des Attributs **class** eine Klasseninformation mit. Für diese von uns definierte Klasse **inputOutput** befinden sich in der CSS-Datei spezielle Design-Anweisungen. Wir müssen also das Design nicht an dieser Stelle über Attribute (insbesondere das CSS-Attribut **style**) direkt im HTML-Code festlegen, sondern wir lagern diese Einstellungen in die separate CSS-Datei aus und machen unseren Code so übersichtlicher und leichter wartbar; auf diese Weise könnten wir also, wenn wir das Design des Taschenrechner-Displays ändern wollten, über das **link**-Element im Header der Seite einfach eine andere CSS-Datei einbinden, die ebenfalls Design-Anweisungen für die Klasse **inputOutput** enthält und schon würde sich die Darstellung des Displays ändern, ohne, dass wir die eigentliche Webseite (das HTML-Dokument) hätten anpassen müssen.

### ■■ Zeilen 17–65.

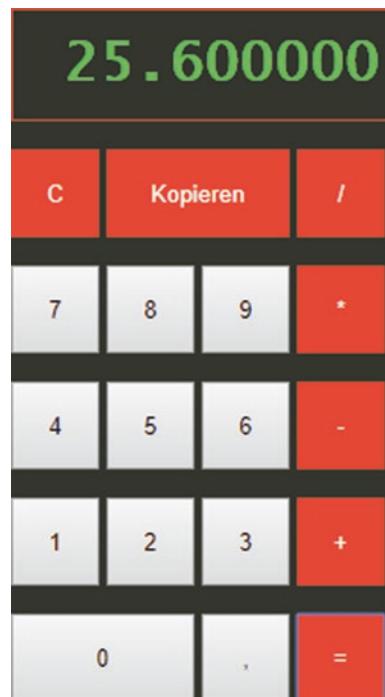
Hier folgen nun die eigentlichen Buttons. Durch das **onClick**-Attribut weisen wir den Buttons jeweils eine JavaScript-Funktion zu, die immer dann ausgelöst wird, wenn der Benutzer auf den Button klickt. Diese Event-Handler-Funktion ist entweder die Funktion **taste(zeichen)**, der als Argument das Zeichen (Ziffer oder Operator), das auf dem Tasten-Button steht, übergeben wird, oder aber die speziellen Funktionen **rechnen()** zum Anstoßen der Berechnung, wenn der Benutzer auf das Gleichheitszeichen klickt, **loeschen()** zum Leeren des Displays oder **kopieren()** zum Kopieren des aktuellen Displayinhalts in die Zwischenablage. Als Attribut **class** haben alle Buttons zunächst **normalButton**, Ausnahmen sind die speziellen Funktionstasten, das heißt, die Operatoren und die Kopieren-, Löschen-, sowie die Gleichheitszeichen-Taste. Diese werden später auf der Webseite in Orange dargestellt. Die Funktionstasten gehören nicht nur der **normalButton**-Klasse, sondern auch der **functionButton**-Klasse an. Die **functionButton**-Klasse sorgt dafür, dass diese Buttons eine Orange-Färbung erhalten, während die Buttons der **normalButton**-Klasse die Standardfärbung (üblicherweise einen Grauton) erhalten. Das schauen wir uns im später im Zusammenhang mit den CSS-Anweisungen noch genauer an.

Wie Ihnen vielleicht aufgefallen ist, haben die Buttons selbst keine **id** erhalten. Das ist strenggenommen etwas unsauber, aber für uns nicht nötig, da wir die Buttons aus unserem JavaScript-Code heraus nicht ansprechen müssen. Es ist genau umgekehrt: Die Buttons sprechen unseren Code an, indem Sie die entsprechende Event-Handler-Funktion aufrufen, wenn sie vom Benutzer angeklickt werden.

Die gesamte Oberfläche sehen Sie in □ Abb. 32.7.

### 32.6 · Beispiel: Einfacher Taschenrechner

■ Abb. 32.7 Die Oberfläche der Taschenrechner-Anwendung



#### 32.6.2 Die CSS-Designanweisungen

Um den in der HTML-Datei beschriebenen Grundaufbau der Oberfläche von der detaillierten designerischen Gestaltung der einzelnen Elemente zu trennen, lagern wir letztere in unserem Beispiel in eine separate CSS-Datei **taschenrechner.css** aus.

Die CSS-Datei definiert Design-Anweisungen für drei Klassen von Objekten, **normalButton** (grundsätzliche alle Buttons), **functionButton** (die Funktionstasten) und **inputOutput** (das Display unseres Taschenrechners). Die Design-Anweisungen werden pro Klasse in einem von geschweiften Klammern umschlossenen CSS-Block dargestellt. Vor dem Block steht der sogenannte *Selektor*, der festlegt, auf welche HTML-Elemente der Webseite die Designanweisungen angewendet werden sollen. Der vorangestellte Punkt bedeutet dabei jeweils „Alle Objekte, deren **class**-Attribut die angegebene Klasse *enthält*“. In ► Abschn. 32.6.1 hatten wir gesehen, dass die Funktionsbuttons zwei Klassen angehören, **normalButton**, also der Klasse für alle Buttons, und der speziellen Klasse **functionButton**. Der CSS-Selektor **.functionButton** führt also dazu, dass die entsprechenden Designanweisungen auf diese Button angewendet werden.

```

1 .normalButton {
2     width:50px;
3     height:50px;
4 }
5
6 .functionButton {
7     background-color: #ED5036;
8     color: #FFFFFF;
9     border: 1px solid #ED5036;
10 }
11
12 .inputOutput {
13     width:208px;
14     height:60px;
15     background-color: #282923;
16     color: #66FF33;
17     border: 1px solid #ED5036;
18     padding-right: 5px;
19     font-family: "Lucida Console";
20     font-size:32px;
21     font-weight: bold;
22     text-align: right;
23 }

```

### ■■ Zeilen 2–3.

Für die Klasse **normalButton**, und damit zunächst für alle Buttons, definieren wir Höhe und Breite in Pixeln (Löschen Sie diese Designanweisungen einmal aus der CSS-Datei und laden Sie die Seite im Browser neu. Was passiert?).

32

### ■■ Zeilen 7–9.

Für die Klasse **functionButton** definieren wir zusätzlich spezielle Hintergrund und Vordergrundfarben und die Gestaltung der Button-Umrundung (hier eine ein Pixel breite, durchgezogene Linie in der gleichen Farbe, die auch der Button-Hintergrund besitzt). Diejenigen Buttons, die nur der Klasse **normalButton** angehören (also vor allem die Zifferntasten), haben natürlich auch eine farbliche Gestaltung, die wir aber nicht explizit festgelegt haben; deshalb werden Standardwerte verwendet, die normalerweise dazu führen, dass diese Buttons grau dargestellt werden. Für die Klasse **functionButton** überschreiben wir diese Standardwerte mit unseren speziellen Farbangaben. Öffnen Sie, nachdem Sie die Seite im Browser geladen haben, einmal die Entwicklerwerkzeuge und klicken Sie auf *Elements* (in anderen Browsern als Google *Chrome* mag das entsprechende Tab etwas anders heißen). Dort finden Sie eine Funktionstaste (in Google *Chrome* ganz oben links), mit der Sie in einen speziellen Element-Inspektionsmodus wechseln können. In diesem Modus können Sie ein Element auf der Webseite durch Anklicken auswählen und erhalten in den Entwicklerwerkzeugen weitere Informationen zu diesem Element angezeigt. Im Element-Inspektor (Abbildung □ Abb. 32.8) sehen Sie links das ausgewählte

### 32.6 · Beispiel: Einfacher Taschenrechner

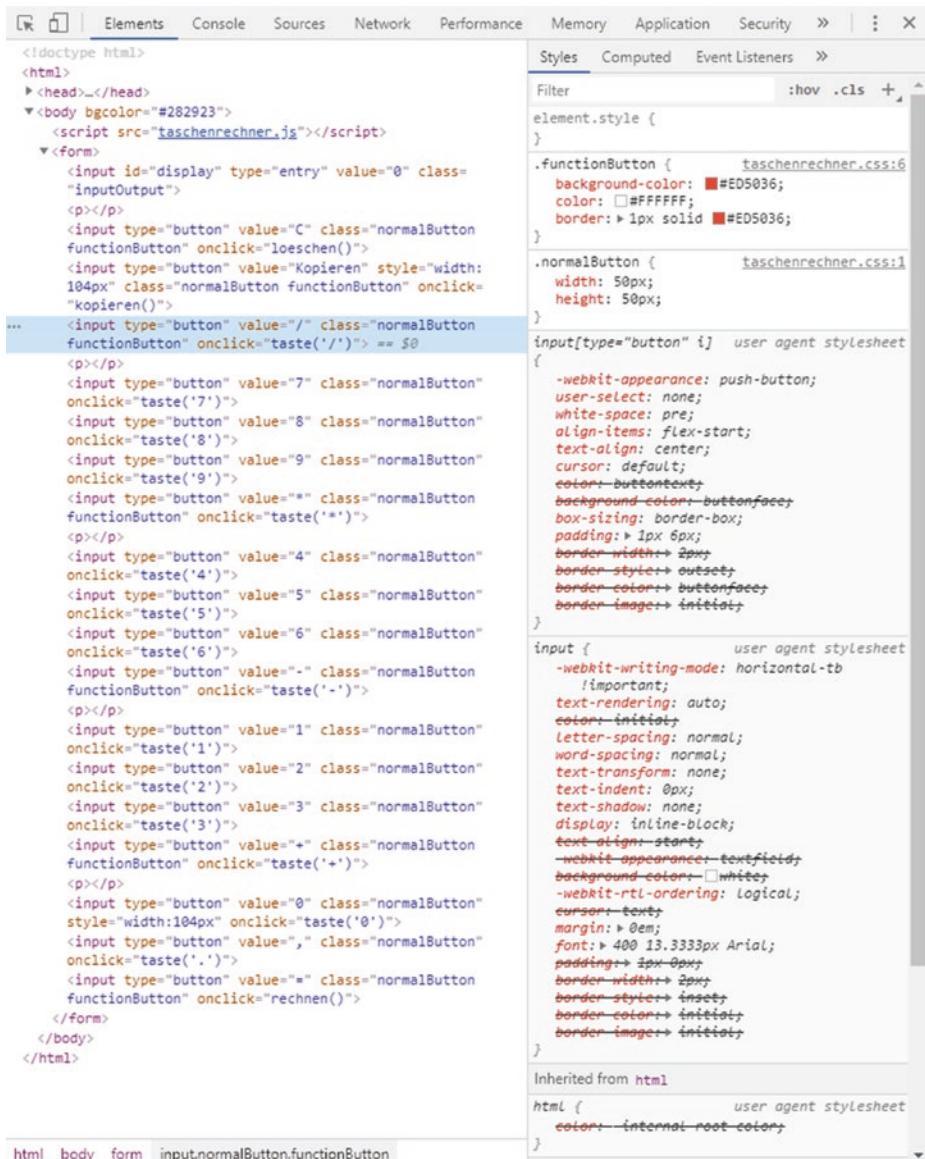


Abb. 32.8 Element-Inspektor in Google Chrome

Element im HTML-Quelltext der Seite und rechts die CSS-Designvorgaben für das Element. Die Designvorgaben lesen sich dabei von unten nach oben. In unserem Beispiel (ausgewählt wurde der Funktionsbutton mit dem Divisionsoperator) sehen Sie eine ganze Reihe CSS-Design-Eigenschaften, die mit Standardwerten vorbelegt sind, und zwar zunächst für die Klasse `input` (CSS-Block ganz unten), dann, unmittelbar darüber, ein Block mit zusätzlichen Eigenschaften speziell für `input`-Elemente vom Typ `button`. Darüber folgenden dann die beiden CSS-Blöcke für

die Buttons der von uns definierten Klassen **normalButton** und **functionButton**. Manche Eigenschaften sind durchgestrichen. Das bedeutet, dass diese Eigenschaften von einem spezielleren CSS-Block überschrieben werden. Beispielsweise sind die Eigenschaften **color** und **background** im CSS-Block für die input-Elemente vom Typ **button** durchgestrichen, weil sie weiter oben im speziellen CSS-Block für die Buttons der Klasse **functionButton** (und der ausgewählte Button ist ja ein solcher) abweichend definiert werden. Auf diese Weise erkennen Sie schnell, welche (Werte für die) CSS-Design-Eigenschaften Ihr Element besitzt und woher diese stammen.

Den Buttons für das Kopieren des Displayinhalts und die Ziffer **0** weisen wir im HTML-Code der Weboberfläche mit Hilfe Ihres **style**-Attributs direkt eine Breite zu. Das **style**-Attribut enthält CSS-Code, der sich nur auf das betreffende Element bezieht. Dieser geht allen anderen Anweisungen in der CSS-Stylesheet-Datei vor, wie Sie sich leicht überzeugen können, wenn Sie die CSS-Anweisungshierarchie eines der Elemente im Element-Inspektor unter die Lupe nehmen.

#### ■ ■ Zeilen 13–22.

Hier wird das Design für das Taschenrechner-Display festgelegt; unter anderem der Hintergrund, die Schriftart, -farbe und -größe sowie die Text-Einrückung vom Rand des Elements (*padding*).

### 32.6.3 Der JavaScript-Code

Der JavaScript-Code unserer Anwendung besteht aus den vier Event-Handler-Funktionen, die wir aus der HTML-Oberfläche heraus beim Anklicken der unterschiedlichen Buttons auslösen:

```
32
1 function taste(zeichen) {
2     var display = document.getElementById('display');
3     display.value = display.value + zeichen;
4 }
5
6 function loeschen() {
7     var display = document.getElementById('display');
8     display.value = '0';
9 }
10
11 function kopieren() {
12     var display = document.getElementById('display');
13     display.select();
14     document.execCommand('copy');
15 }
16
17 function rechnen() {
18     var display = document.getElementById('display');
19     display.value = Number(eval(display.value)).toFixed(6);
20 }
```

### ■■ Zeilen 1–9.

Die Funktionen **taste(zeichen)** und **loeschen()** sind Event-Handler-Funktionen, die aufgerufen werden, wenn der Benutzer einen der entsprechenden Buttons anklickt. Wie Sie sich erinnern, rufen wir die Funktion **taste(zeichen)** aus dem HTML-Code der Webseite immer mit demjenigen Zeichen als Argument auf (egal, ob Ziffer oder Operator), mit dem die gedrückte Taste belegt ist. Durch diesen „Trick“ benötigen wir nur eine Funktion für alle Buttons, statt für jeden Button einen speziellen Event Handler.

Beide Funktionen ändern die Anzeige auf dem Display. Dazu „greifen“ wir zunächst stets das Display-Element unseres Webseiten-Formulars mit Hilfe der Methode **getElementById()** des **document**-Objekts. Danach ändern wir die Eigenschaft **value** des Display-Elements, also den auf dem Eingabeelement angezeigten Text; bei **taste()** fügen wir dem aktuellen Wert einfach die Beschriftung des gedrückten Buttons hinzu (Zeile 3).

### ■■ Zeilen 11–15.

Zum Kopieren des aktuellen Display-Inhalts in die Zwischenablage markieren wir zunächst den vorhandenen Text mit Hilfe der Methode **select()** des **input**-Elements **display** und rufen dann den Kopierbefehl des Browsers auf.

### ■■ Zeilen 17–20.

Wenn der Benutzer die eingegebene Berechnung ausführen will und den Button mit dem Gleichheitszeichen klickt, wird der Wert des Displays aktualisiert. Wir nutzen hier die Funktion **eval(ausdruck)**, die den als String übergebenen **ausdruck** auswertet, das heißt in unserem Fall: ausrechnet. Das Ergebnis wandeln wir dann zunächst mit **Number()** in eine Zahl um, die wir im Anschluss sogleich mit Hilfe ihrer **toFixed()**-Methode in eine String-Darstellung mit sechs Nachkommastellen formatieren.

## 32.7 Beispiel: Color Picker

In diesem Beispiel entwickeln wir eine kleine Anwendung, die es erlaubt, Farben nach dem Rot-Grün-Blau-(RGB-)Schema auf benutzerfreundliche Weise zu gestalten und in die HTML-typische hexadezimale Codierung zu konvertieren. Solche Anwendungen, auch viel ausgefeilte, finden Sie massenhaft im Internet.

### 32.7.1 Die Web-Oberfläche

Die Oberfläche unserer Anwendung ist sehr schlicht. Sie besteht lediglich aus drei Schiebereglern, mit denen man die Farbanteile von Rot, Grün und Blau ändern kann, sowie einem Feld, in dem die resultierende Farbe als hexadezimaler Code der Form **#RRGGBB** dargestellt wird. Die aktuell über die Schieberegler ausgewählte

Farbe wird als Hintergrundfarbe der Webseite verwendet, sodass sich der Benutzer einen guten Eindruck von der Farbe, die er erzeugt hat, machen kann.

Schauen wir uns die Oberfläche im Einzelnen an:

```
1  <!DOCTYPE html>
2  <html>
3
4  <head>
5      <title>Color Picker</title>
6      <noscript>Bitte JavaScript aktivieren!</noscript>
7  </head>
8
9  <body id="bodyElem">
10
11  <div style="background:#FFFFFF; margin: 0 auto;
12      padding:10px; width:400px">
13
14      <form>
15          <input id="hexColor" type="text"
16              value="#000000" readonly/>
17          <p>Rot:</p>
18          <input id="colorRedRange" type="range"
19              value="255" min="0" max="255"
20              oninput="farbeAnpassen()"/>
21          <input id="colorRedOutput"
22              type="text" value="255"
23              readonly/>
24          <p></p>
25          <p>Grün:</p>
26          <input id="colorGreenRange"
27              type="range" value="255"
28              min="0" max="255"
29              oninput="farbeAnpassen()"/>
30          <input id="colorGreenOutput"
31              type="text"
32              value="255" readonly/>
33          <p></p>
34          <p>Blau:</p>
35          <input
36              id="colorBlueRange"
37              type="range"
38              value="255" min="0"
39              max="255"
40              oninput="farbeAnpassen()"/>
41          <input
42              id="colorBlueOutput"
43              type="text"
```

```
44           value="255"
45           readonly/>
46     </form>
47
48   </div>
49
50   <script src="colorpicker.js"></script>
51
52   </body>
53
54 </html>
```

### ■■ Zeile 9.

Das **body**-Element der Seite versehen wir dieses Mal mit einer **id**, denn wir wollen ja den Hintergrund der Seite entsprechend in dem ausgewählten Farbton einfärben. Dazu müssen wir das **bgcolor**-Attribut des **body**-Elements aus unserem JavaScript-Code heraus anpassen.

### ■■ Zeilen 11–12.

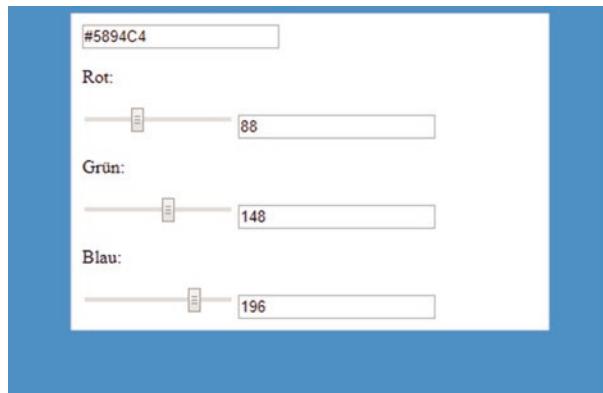
Ein **div**-Element markiert in HTML einfach einen zusammenhängenden Bereich auf der Seite, letztlich also eine (zunächst einmal unsichtbare) Box. In eine solche Box packen wir alle unsere Steuerelemente. Das **div**-Element können wir dann nämlich über sein CSS-Attribut **style** weiß einfärben (#FFFFFF entspricht Farbanteilen für Rot, Grün und Blau von jeweils 255) und auf der Seite zentrieren (das geschieht mit dem Wert **0 auto** für die **margin**-Eigenschaft). Außerdem statten wir die Box mit den Steuerelementen, die durch unser **div**-Element beschrieben wird, mittels der **padding**-Eigenschaft mit einem Rand von 10 Pixeln Breite aus, sodass die Steuerelemente zum Rand der Box etwas Freiraum haben, und fixieren die Breite der Box auf 400 Pixel.

### ■■ Zeilen 14–46.

Das Formular innerhalb der **div**-Box besteht zum einen aus einem Texteingabefeld, in dem wir den in hexadezimale Schreibweise konvertierten Farbcodes darstellen (Zeile 14). Standardmäßig soll die Farbe Weiß sein, solange der Benutzer nicht über die Schieberegler eine andere Farbe ausgewählt hat. Das Eingabefeld soll außerdem nur lesend verwendet werden können, deshalb fügen wir das Attribut **readonly** hinzu, das ein **boolean**-Attribut ist und deswegen für einen wahren Wert keine explizite Wertzuweisung benötigt (wir hätten aber natürlich auch **readonly="true"** schreiben können).

Danach fügen wir drei Schieberegler ein (**colorFarbeRange**), einen für jeden Farbanteil unseres RGB-Werts. Die Schieberegler sind vom Typ **range** und sind regelbar zwischen **0 (min)** und **255 (max)**. Jedes Mal, wenn der Benutzer sie bewegt (Ereignis **onInput**), wird der Event Handler **farbeAnpassen()** ausgelöst, der, wie wir weiter unten sehen werden, den hexadezimalen Farbcodes im **hexColor**-Inputfeld anpasst, die Hintergrundfarbe der Seite neu setzt und schließlich in einem (Readonly-)Eingabefeld (**colorFarbeOutput**) neben dem Schieberegler den neuen Farb-

■ Abb. 32.9 Die Oberfläche der Color-Picker-Anwendung



anteil anzeigt. Durch die leeren Paragraph-Elemente `<p></p>` sorgen wir für Zeilenbrüche und damit bessere Lesbarkeit.

Die gesamte Oberfläche sehen Sie in ■ Abb. 32.9.

### 32.7.2 Der JavaScript-Code

Die JavaScript-Datei `colorpicker.js`, die wir in Zeile 30 in unsere Webseite einbinden, besteht lediglich aus dem Event Handler `farbeAnpassen()`, jener Funktion also, die jedes Mal dann aufgerufen wird, wenn der Benutzer einen der Schieberegler bewegt.

```

32
1  function farbeAnpassen() {
2
3      var bodyElem = document.getElementById("bodyElem");
4      var hexColor = document.getElementById("hexColor");
5
6      var colorRedRange = document.getElementById(
7          "colorRedRange");
8      var colorGreenRange = document.getElementById(
9          "colorGreenRange");
10     var colorBlueRange = document.getElementById(
11         "colorBlueRange");
12
13     var colorRedOutput = document.getElementById(
14         "colorRedOutput");
15     var colorGreenOutput = document.getElementById(
16         "colorGreenOutput");
17     var colorBlueOutput = document.getElementById(
18         "colorBlueOutput");
19
20     var hexRed = Number(colorRedRange.value).toString(16);
21     var hexGreen = Number(colorGreenRange.value)
22         .toString(16);
23     var hexBlue = Number(colorBlueRange.value).toString(16);
24

```

### 32.7 · Beispiel: Color Picker

```
25      if (hexRed.length == 1) hexRed = "0" + hexRed;
26      if (hexGreen.length == 1) hexGreen = "0" + hexGreen;
27      if (hexBlue.length == 1) hexBlue = "0" + hexBlue;
28
29      var hex = "#" + hexRed + hexGreen + hexBlue;
30      hexColor.value = hex.toUpperCase();
31
32      bodyElem.bgColor = hex;
33
34      colorRedOutput.value = colorRedRange.value;
35      colorGreenOutput.value = colorGreenRange.value;
36      colorBlueOutput.value = colorBlueRange.value;
37  }
```

Der Code im Einzelnen:

#### ■■ Zeilen 3–18.

Hier erzeugen wir zunächst Variablen für die unterschiedlichen Elemente der Oberfläche, die wir ansprechen müssen.

#### ■■ Zeilen 20–23.

Als nächstes nehmen wir die aktuellen Werte, auf die die Schieberegler eingestellt sind (mit Hilfe ihres **value**-Attributs), und konvertieren den Wert in einen hexadezimalen String. Wie Sie sich an ► Abschn. 31.3.2 erinnern, hat die **toString()**-Methode ein Argument, dass die Basis des Zahlensystems angibt, in das die Zahl für die Darstellung in Form eines Strings konvertiert werden soll, in unserem Falle also 16, da wir ja eine hexadezimale Darstellung erreichen wollen.

#### ■■ Zeilen 25–27.

Mit den Variablen **hexRed**, **hexGreen** und **hexBlue** haben wir eigentlich schon alles zusammen, was wir benötigen, um den hexadezimalen Farbwert im Format **#RRGGBB** darzustellen. Eine kleine Komplikation gibt es aber noch: Der Wert der Farbanteilsvariablen könnte einstellig ausfallen (wenn nämlich der Dezimalwert des jeweiligen Farbanteils kleiner als 16 ist). In diesem Fall müssen wir eine führende **0** voranstellen, denn hexadezimale Farbcodes der Form **#RRGGBB** erwarten eben pro Farbanteil *genau zwei* Werte-Stellen. Deshalb prüfen wir hier die Länge der zuvor erzeugten Farbanteil-Strings und ergänzen nötigenfalls eine **0**.

#### ■■ Zeilen 29–30.

Nun können wir den hexadezimalen Wert zusammensetzen und ihn in unserem (Readonly-)Eingabe-Element **hexColor** anzeigen.

#### ■■ Zeile 32.

Den soeben ermittelten hexadezimalen RGB-Wert weisen wir auch dem **bgcolor**-Attribut des **body**-Elements zu. Sobald also der Benutzer einen der Farbanteil-Regler betätigt, ändert sich nicht nur die Anzeige des hexadezimalen Farbwerts, **hexColor**, sondern auch die Hintergrundfarbe der Webseite.

**■■ Zeilen 34–36.**

Schließlich zeigen wir noch die Farbanteil-Werte als dezimale Zahlen in den dafür vorgesehenen (Readonly-)Eingabefeldern an, die neben den jeweiligen Farbanteil-Schiebereglern stehen.

## 32.8 Zusammenfassung

---

In diesem Kapitel haben wir uns damit beschäftigt, wie Daten mit Hilfe von JavaScript ein- und ausgeben werden können. Insbesondere haben wir uns dabei mit der JavaScript-Konsole befasst und gesehen, wie JavaScript-Anwendungen mit einer Webseite interagieren können.

Folgende Punkte sollten Sie aus diesem Kapitel unbedingt mitnehmen:

- Mit der Methode **console.log()** können Objekte in der JavaScript-Konsole ausgegeben werden.
- Template Literals erlauben Ihnen, in Zeichenketten Variablen in Form von Platzhaltern einzubauen, die zum Zeitpunkt der Erzeugung des Literals mit dem aktuellen Wert der Variable ersetzt werden; die Variable wird dabei als **`\${variable}`** geschrieben, das gesamte Literal selbst in Backticks (`) eingeschlossen.
- Alternativen zum Template Literal sind die String-Ersetzung und das Verketten von Strings und Objekten, die sich als Strings ausgeben lassen, mit Hilfe des Plus-Operators.
- Mit **console.warn()** und **console.error()** können Sie Warn- bzw. Fehlermeldungen in der Konsole ausgeben.
- Interaktion mit dem Benutzer kann auch über Dialogboxen erfolgen; insbesondere mit **alert()** (Anzeige einer Meldung), **confirm()** (Dialog mit „Okay“ und „Abbrechen“ als Optionen) und **prompt()** (Texteingabe in einem Popup-Dialog).
- JavaScript eignet sich hervorragend dazu, die HTML-Elemente von Webseiten zu bearbeiten und so die Webseite durch die JavaScript-Anwendung dynamisch zu gestalten.
- Der einfachste Weg dazu besteht darin, mit **document.write()** HTML-Code an die aktuelle Stelle, an der das Skript in der Webseite ausgeführt wird, zu schreiben.
- Die Bestandteile von HTML-Seiten, vor allem die HTML-Elemente selbst, ihre Attribute und der Text, der in ihrem Inneren vorhanden ist, lassen sich hierarchisch als Knoten des sogenannten *Document Object Model* (DOM) darstellen.
- JavaScript erlaubt es, Elemente des DOM zu selektieren und in JavaScript als HTML-Element-Objekte zu bearbeiten, wobei sich Änderungen an diesen Objekten unmittelbar in der Darstellung der Webseite widerspiegeln.
- DOM-Knoten können Sie vor allem über ihr Attribut **id** (mit **document.getElementById()**), über ihren Typ (mit **document.getElementsByTagName()**); Plural beachten, hier wird ein Array zurückgegeben, weil es mehrere Elemente

geben kann, auf die das Kriterium zutrifft!), oder ihre CSS-Klasse (mit `document.getElementsByClassName()`; auch hier ist die Rückgabe ein Array).

- Zurückgegeben werden jeweils Element-Objekte/Arrays von Element-Objekten, die die HTML-Elemente der Webseite repräsentieren und deren Eigenschaften die Attribute jener HTML-Elemente sind.
- Auch lässt sich ausgehend von einem Element-Objekt die DOM-Struktur ausnutzen, um mit Eigenschaften wie `childNodes` und `parentNode` hierarchisch verbundene Objekte zu selektieren.
- Die Eigenschaften `innerHTML` und `innerText` eines Element-Objekt stellen den in einem HTML-Element enthaltenen HTML-Code (das heißt, den Code der im DOM hierarchisch untergeordneten Elemente) bzw. den enthaltenen Text dar.
- HTML-Elemente können auf der Webseite mit `document.createElement(typ)` erzeugt und mit der HTML-Element-Objekt-Methode `appendChild(neues_element)` als Kind-Element an dasjenige Element angehängt werden, dessen `appendChild()`-Methode aufgerufen wird; mit `insertBefore(neues_element, vor_kind)` wird ein HTML-Element `neues_element` als Kind-Element vor dem Kind-Element `vor_kind` desjenigen Elements, dessen `insertBefore()`-Methode aufgerufen wird, eingefügt; `remove(element)` entfernt das HTML-Element-Objekt `element` von der Webseite.
- Die in der Praxis wichtigste Form der Interaktion mit dem Benutzer auf Webseiten ist die Verwendung von *Formularen*; sie werden in HTML mit dem `form`-Element erzeugt, ihre Bestandteile, das heißt, die Steuerelemente wie Texteingaben oder Buttons, stehen also zwischen `<form>` und `</form>`.
- Die Elemente der Formulare sind überwiegend vom HTML-Typ `input` und werden über ihr `typ`-Attribut weiter differenziert; so ist beispielsweise "text" der Typ für ein Text-Eingabefeld, "button" der für einen Button und "slider" der für einen Schieberegler.
- Benutzeraktionen mit diesen Steuerelementen (zum Beispiel der Klick auf einen Button) können über Ereignisse mit Event-Handler-Funktionen im JavaScript-Code verknüpft werden, die automatisch immer dann aufgerufen werden, wenn das Ereignis, das mit der Aktion einhergeht, ausgelöst wird; die Aufrufe der Event Handler werden den HTML-Elementen als Attribut-Werte mitgegeben, wobei der sich der Attribut-Name aus `on` und dem Ereignis zusammensetzt, also zum Beispiel `onclick` für das `click`-Event.

## 32.9 Lösungen zu den Aufgaben

### ■ Aufgabe 32.1

Die Geschwisterelemente von `elem` (einschließlich `elem` selbst) sind nichts anderes als die Kind-Elemente des Eltern-Elements von `elem`. Also:

```
elem.parentElement.children
```

Das gibt eine **HTMLCollection** zurück, eine Sammlung von Objekten, die mit den HTML-Elementen korrespondieren. Mit diesem Objekt vom Typ **HTMLCollection** können Sie praktisch wie mit einem Array arbeiten.

### ■ Aufgabe 32.2

Ihr **p**-Element könnte in im HTML-Quelltext der Webseite so aussehen:

```
<p id="absatz1">Hier steht eine Notiz.</p>
```

In Ihrem JavaScript-Programm können Sie nun die Hintergrundfarbe ändern:

```
var pElem = document.getElementById('absatz1');
pElem.style.backgroundColor = '#FFFFCC';
```

Die Eigenschaft lässt sich bequem ändern, weil unser Element-Objekt **pElem** das **style**-Attribut in einer gleichnamigen Objekt-Eigenschaft spiegelt. Beachten Sie dabei bitte, dass die CSS-Eigenschaft **background-color** zur Eigenschaft **backgroundColor** der **style**-Eigenschaft unseres Element-Objekts in JavaScript wird (das heißt, der Bindestrich entfällt, und weitere Anfangsbuchstaben nach dem Anfangsbuchstaben des ersten Bestandteils des Eigenschaftsnamen werden zu Großbuchstaben). Außerdem müssen Sie natürlich die dezimalen RGB-Farbanteile in hexadezimale Zahlen umrechnen, um einen gültigen Farbcode der Form '**#RRGGBB**' zu erhalten. Dabei wird 255 (Rot- und Grün-Anteile) zu **FF** und 204 (Blau-Anteil) zu **CC**.

### ■ Aufgabe 32.3

Der Code könnte so aussehen:

**32**

```
var bodyElem = document.getElementsByTagName('body')[0];

var pElem = document.createElement('p');
pElem.style.fontWeight = 'bold';
pElem.innerText = 'Hier ist Schluss';

bodyElem.appendChild(pElem);
```

Zunächst wird das **body**-Element der Webseite über sein Element-Tag gegriffen (Achtung: **getElementsByTagName()** liefert ein Array von HTML-Knoten zurück, da es ja – zumindest bei anderen Element als body – durchaus mehrere Ex-

emplare eines Element-Typs in der Webseite geben kann). Dann erzeugen wir ein neues **p**-Element, formatieren es über seine CSS-Eigenschaften, geben ihm den '**Hier ist Schluss**'-Text mit und fügen es dem **body**-Element der Webseite als neues Kind-Element hinzu, hängen es also an die bestehenden Kinder hinten an.

Übrigens: Wenn Sie dieses Skript in einer Webseite einbinden, wie wir es üblicherweise tun, also über das **script**-Attribut des **body** der Webseite, dann erscheint '**Hier ist Schluss**' am *Anfang* der Webseite! Des Rätsels Lösung ist eine einfache: Das Skript wird am Anfang des **body** geladen und ausgeführt, zu einem Zeitpunkt also, da im Rumpf der Webseite noch keinerlei sonstige Elemente vorhanden sind. Das Element, das wir durch den obigen Code hinzufügen, ist damit das erste und wird folgerichtig am Anfang der Seite dargestellt.

#### ■ Aufgabe 32.4

In den HTML-Code muss zunächst das (noch „leere“) **span**-Element eingebaut werden:

```
<span id="output"></span>
```

Den Code der Funktion **umrechnen()** müssen Sie dann nur noch geringfügig ändern, indem Sie zunächst das **span**-Element selektieren und die Ausgabe darauf dann per Zuweisung an die **innerHTML**-Eigenschaft vornehmen:

```
function umrechnen() {
    var temp = Number(
        document.getElementById('temp').value);
    var richtung = document.getElementsByName('richtung');
    var outputSpan = document.getElementById('output');

    if (richtung[0].checked == true) {
        outputSpan.innerHTML =
            `<p>${temp} Kelvin in Grad Celsius sind: ${temp -
            273.15} Grad Celsius.<p>`;
    }
    else {
        outputSpan.innerHTML =
            `<p>${temp} Grad Celsius in Kelvin sind: ${temp +
            273.15} Kelvin.<p>`;
    }
}

function aendern(einheit) {
    var einheitLabel = document.getElementById('einheitLabel');
    einheitLabel.innerHTML = einheit;
}
```

### ■ Aufgabe 32.5

Die Webseite könnte folgendermaßen aussehen:

```
<!DOCTYPE html>
<html>

<head>
    <title>Schriftgrößen</title>
    <noscript>Bitte JavaScript aktivieren!</noscript>
</head>

<body>
<script src="fontgroessenregler.js"></script>

<form>
    <p>Font-Größe:
        <input id="regler" type="range" min="1"
               max="150" value="20"
               onchange="fontgroesse()">
    </p>
</form>

<span id="beispieltext" style="font-size: 20px;">
    Ein Beispieltext in Größe 20</span>
</body>

</html>
```

Das zugehörige JavaScript-Programm in **fontgroessenregler.js** würde dann so aussehen:

```
32
function fontgroesse() {
    var groesse = Number(
        document.getElementById('regler').value);
    var text = document.getElementById('beispieltext');

    text.style.fontSize = groesse + 'px';
    text.innerHTML =
        "Ein Beispieltext in Größe " + String(groesse);
}
```



# Wie arbeite ich mit Programm-funktionen, um Daten zu bearbeiten und Aktionen auszulösen?

## Inhaltsverzeichnis

- 33.1 Arbeiten mit Funktionen – 536**
  - 33.1.1 Definition von Funktionen – 536
  - 33.1.2 Rückgabewerte – 541
  - 33.1.3 Argumente und Parameter von Funktionen – 542
  - 33.1.4 Gültigkeitsbereich von Variablen in Funktionen – 546
- 33.2 Arbeiten mit Modulen/Bibliotheken – 549**
  - 33.2.1 Eigene Module entwickeln und verwenden – 549
  - 33.2.2 Externe Module/Bibliotheken finden und einbinden – 551
- 33.3 Frameworks – 552**
- 33.4 Zusammenfassung – 553**
- 33.5 Lösungen zu den Aufgaben – 554**

## Übersicht

Als nächstes werden wir uns mit Funktionen befassen, die in JavaScript – wie in vielen anderen Programmiersprachen auch – das wichtigste Arbeitstier des Programmierers sind. Schließlich arbeiten Sie nicht nur ständig mit vordefinierten Funktionen, die Ihnen JavaScript von Haus aus anbietet, oder die Sie aus Erweiterungsbibliotheken beziehen, sondern Sie schreiben regelmäßig auch eigene Funktionen; insbesondere natürlich die Event Handler, die in den ereignisgesteuerten JavaScript-Applikationen eine zentrale Rolle einnehmen. Kein Wunder also, dass die intensive Beschäftigung mit Funktionen ein Kernbestandteil unserer Tour durch JavaScript ist.

In diesem Kapitel werden Sie lernen:

- wie Sie Funktionen definieren und aufrufen
- wie Funktionen Parameter verarbeiten und Ergebnisse zurückliefern
- dass auch Funktionen Objekte sind, und welche Konsequenzen das hat
- wie die Gültigkeitsbereiche von Variablen in JavaScript – insbesondere in Hinblick auf Funktionen – geschnitten sind und wie sich das auf die Sichtbarkeit von Variablen auswirkt
- wie Sie die zur Verfügung stehenden Funktionen über den Standard-Sprachumfang hinaus mit externen Bibliotheken erweitern, und wie Sie geeignete Erweiterungsbibliotheken finden
- was Frameworks sind und wie sie sich von Bibliotheken unterscheiden.

## 33.1 Arbeiten mit Funktionen

### 33.1.1 Definition von Funktionen

#### ■ Funktionen definieren

Funktionen werden in JavaScript mit dem Schlüsselwort **function** definiert. Etwaige Argumente stehen hinter dem Funktionsbezeichner in runden Klammern, die auch dann notwendig sind, wenn die Funktion gar keine Argumente übergeben bekommt. Der Programmcode, der ausgeführt wird, wenn die Funktion aufgerufen wird, folgt als Code-Block in geschweiften Klammern. Eine einfache Funktion, die lediglich „Hallo Welt“ in die Konsole ausgibt, würde damit so aussehen:

```
function hallo() {
    console.log('Hallo Welt! ');
}
```

Sie könnte nun aus dem Programm (oder der Konsole) aufgerufen werden:

```
hallo();
```

Wenn Sie beim Aufruf der Funktion in der Konsole die runden Klammern vergessen, was rasch passieren kann, wenn die Funktion keine Argumente übernimmt, wird Ihnen der Quelltext der Funktion angezeigt.

### ■ Funktionen als Objekte

Funktionen sind in JavaScript selbst Objekte, und zwar vom Typ **function**, wie sich leicht überprüfen lässt:

```
> typeof(hallo)
"function"
```

Weil sie Objekte sind, können sie auch anderen Variablenobjekten zugewiesen werden:

```
> gruss = hallo
> gruss()
Hallo Welt!
```

Wenn wir hier **gruss = hallo()** geschrieben hätten, hätten wir einer Variablen **gruss** den Funktionswert von **hallo** zugewiesen, denn **hallo()** ist nichts anderes als ein Aufruf der gleichnamigen Funktion (wie wir weiter unten sehen werden, wäre dieser Funktionswert **undefined**, weil die Funktion keinen Wert explizit zurückliefert).

Als Objekte besitzen sie darüber hinaus eine Reihe von Methoden und Eigenschaften; die Funktion **toString()** etwa liefert den Quelltext der Funktion als Zeichenkette zurück:

```
> gruss.toString()
"function hallo() {
  console.log('Hallo Welt!');
}"
```

Die runden Klammern dürfen hier nicht verwendet werden, da wir die Funktion ja nicht aufrufen, sondern lediglich auf ihre Objekt-Methoden und -eigenschaften zugreifen wollen.

Dass Funktionen Objekte sind, wird aber auch an anderen Stellen deutlich, an denen wir mit ihnen arbeiten können, wie mit jedem anderen Objekt auch. In ► Abschn. 31.5.2 haben wir gesehen, wie Objekte mit Hilfe des Schlüsselwortes **var** erzeugt werden können. Genau das können wir auch mit Funktionsobjekten machen:

```
var hallo = function() {
  console.log('Hallo Welt! '');
```

Wir initialisieren also die Variable **hallo** mit einem Funktionsausdruck, der mit dem Schlüsselwort **function** eingeleitet wird. Im Anschluss ist **hallo** ein aufrufbares Objekt und kann mit **hallo()** ausgeführt werden.

Weil **hallo** ja nun ein richtiges Objekt ist, können wir ihm – was tatsächlich etwas kurios anmutet – auch Eigenschaften hinzufügen:

```
var datum = {
    tag: 0,
    monat: 0,
    jahr: 0,
    anzeigen: function () {
        console.log(
            this.tag + '. ' + this.monat + '. ' + this.jahr)
    }
}
```

Das Objekt bleibt also vom Typ **function**, besitzt jetzt aber eine zusätzliche Eigenschaft, **anzahl**. Als wir in ► Abschn. 31.5.5 Objekte mit Hilfe von Konstruktorfunktionen erzeugt haben, haben wir etwas sehr Ähnliches getan (blättern Sie nochmal zurück zur Konstruktor-Funktion des **Produkt**-Objekt-Typs). Die Konstruktorfunktion hat letztlich Eigenschaften des aktuellen Objekts geschrieben und dazu das Schlüsselwort **this** verwendet.

Weil Funktionen einfach Objekte sind, können Sie auch als Eigenschaften in anderen Objekten eingesetzt werden; dadurch bekommen diese Objekte aufrufbare *Methoden*. Angenommen, wir wollten ein Objekt entwickeln, dass ein Datum, zerlegt in seine einzelnen Bestandteile, aufnimmt und eine Methode **anzeigen()** besitzt, die das Datum in einem ansprechenden Format ausgibt. Ein solches Objekt könnten wir folgendermaßen definieren:

```
33
var datum = {
    tag: 0,
    monat: 0,
    jahr: 0,
    anzeigen: function() {
        console.log(this.tag + '. ' + this.monat + '. ' + this.
        jahr)
    }
}
```

Beachten Sie das Schlüsselwort **this**. Es ist uns bereits in ► Abschn. 31.5.5 begegnet und stellt einen Bezug zu dem aktuellen Kontext her, in dem (wie hier) eine Eigenschaft oder eine Methode aufgerufen werden. Mit **this.tag** greifen wir also auf die **tag**-Eigenschaft im aktuellen Kontext zu, und das ist der Kontext der Objektdefinition. Lassen Sie das **this**-Schlüsselwort aus, versteht JavaScript nicht, was **tag** sein soll, denn innerhalb der Funktion **anzeigen()** existiert keine Variable diesen Namens.

Nach dieser Deklaration können wir mit dem Objekt arbeiten, indem wir die Datumskomponenten eingeben und dann die Funktion **anzeigen()** aufrufen:

```
datum.tag = 14;  
datum.monat = 12;  
datum.jahr = 2025;  
datum.anzeigen();
```

Dieser Methodenaufruf liefert uns die Ausgabe **14.12.2025.**

Wie Sie sehen, können wir also auf sehr einfache Weise ein Objekt mit aufrufbaren Methoden bestücken, da die Methoden letztlich nur Eigenschaften der Objekte sind, und zwar Eigenschaften vom Typ **function**. Diese unterscheiden sich von anderen Eigenschaften des Objekts lediglich dadurch, dass sie aufrufbar sind.

## ■ Funktionen in Funktionen

Eine Besonderheit von JavaScript besteht darin, dass Funktionen auch innerhalb von Funktionen definiert werden können.

Ein einfaches (wenngleich zugegebenermaßen inhaltlich nicht sehr sinniges) Beispiel:

```
function HalloWelt() {  
    function Hallo() {  
        console.log('Hallo');  
    }  
    function Welt() {  
        console.log('Welt!');  
    }  
    Hallo();  
    Welt();  
}
```

Hier definieren wir innerhalb der Funktion **HalloWelt()** zwei weitere Funktionen **Hallo()** und **Welt()**, die jeweils eine Ausgabe in der Konsole veranlassen. Danach werden beide Funktionen aufgerufen. In der Konsole entstehen auf diese Weise zwei neue Zeilen **Hallo** und **Welt!**.

Alternativ hätten wir die („Unter“-)Funktionen auch durch Objektzuweisungen erzeugen können:

```
function HalloWelt() {  
    Hallo = function() {  
        console.log('Hallo');  
    }  
    Welt = function() {  
        console.log('Welt!');  
    }  
    Hallo();  
    Welt();  
}
```

Das Definieren von Funktionen innerhalb von Funktionen mag auf den ersten Blick wie eine syntaktische Spielerei erscheinen, besitzt aber in dieser zweiten Variante einen praktischen Nutzen, wenn es um die Arbeit mit Code-Modulen geht.

Übrigens: Was wäre passiert, wenn wir in den Definitionen der („Unter“-)Funktionen **this.Hallo = function...** und **this.Welt = function...** geschrieben hätten? Dann hätten wir eine *Konstruktorfunktion* für ein **HalloWelt**-Objekt entwickelt, die dem aktuellen Objekt zwei Eigenschaften mitgibt, nämlich die beiden Funktionen. Danach hätten wir dann Objekte dieses Typs erzeugen und auf ihre beiden (Funktions-)Eigenschaften (also Methoden) zugreifen können:

```
hi = new HalloWelt();
hi.Hallo();
```

### ■ Anonyme Funktionen

Funktionen haben normalerweise einen Namen, unter dem sie aufgerufen werden können. Allerdings gibt es auch die Möglichkeit, Funktionen zu definieren, die keinen eigenen Namen haben, sogenannte *anonyme* Funktionen.

Ein einfaches Beispiel ist folgendes:

```
(function() {
    console.log('Hallo Welt')
})()
```

Sie sehen zunächst runde Klammern, in denen eine Funktionsdefinition steht, allerdings ohne einen Funktionsbezeichner. Der Klammerausdruck liefert eine Funktion (ein Funktionsobjekt) zurück, die wir *sofort wieder aufrufen*. Das ist an dem Klammerpaar am Ende zu erkennen. Es sind die üblichen Klammern, die auch beim Aufruf einer „normalen“ Funktion mit Funktionsbezeichner verwendet werden. Auf diese Weise benötigt die Funktion keinerlei Namen und kann trotzdem aufgerufen werden, aber in unserem Beispiel eben nur im direkten Zusammenhang mit ihrer Definition, da uns sonst der „Griff“ fehlt, an dem wir sie anfassen können.

Für „Syntax-Feinschmecker“: Es gibt eine Möglichkeit, einen solchen „Griff“ herzustellen, die wir bereits zuvor verwendet haben: Wir weisen die anonyme Funktion einem Objekt zu. Betrachten wir dazu noch einmal ein Beispiel von oben:

```
var hallo = function() {
    console.log('Hallo Welt! ');
}
```

Hier erzeugen wir zwar eine Variable mit dem Namen **hallo**, aber die Funktion, die wir ihr zuweisen, besitzt keinen Bezeichner. Geben Sie den Namen der *Variable* in

die Konsole ein, wird ihnen der Quelltext der Funktion angezeigt; darin ist allerdings kein Funktionsbezeichner zu erkennen:

```
> hallo
f () {
    console.log('Hallo Welt!');
}
```

Auch, wenn es etwas verwirrend erscheinen mag: Wir haben ein Objekt namens **hallo** erzeugt, das eine Funktion darstellt, die Funktion jedoch selbst ist anonym, besitzt also keinen Namen.

Definieren wir die Funktion aber durch unmittelbare Anwendung des Schlüsselworts **function**, erhalten wir wiederum ein Funktionsobjekt; dieses Mal hat die Funktion allerdings einen Namen:

```
> var hallo = function hallo() {
    console.log('Hallo Welt!');
}
> hallo
f hallo() {
    console.log('Hallo Welt!');
}
```

Wir müssen also anonymous Funktionen nicht notwendigerweise direkt nach ihrer Definition aufrufen, sondern können sie auch in einem Objekt auffangen.

Ebenso wie die „Funktionen in Funktionen“ sind auch anonymous Funktionen nützlich, wenn es darum geht, ganze Code-Module zu konstruieren. Im „Programmieralltag“ werden natürlich meist Funktionen mit Funktionsbezeichnern verwendet.

### 33.1.2 Rückgabewerte

Funktionen können mit der **return**-Anweisung Objekte zurückgeben. Die folgende Funktion erzeugt eine Zufallszahl zwischen 0 und 10 und gibt sie zurück:

```
function zufall() {
    var zufallsZahl;
    zufallsZahl = Math.round(Math.random()*10,0);
    return zufallsZahl;
}
```

Die **return**-Anweisung kann statt mit dem Schlüsselwort **return** auch wie eine Funktion geschrieben werden, in unserem Fall als **return(zufallszahl)**.

Funktionen, die keine **return**-Anweisung ausführen, geben automatisch **undefined** zurück:

```
> hallo = function hallo() {
    console.log('Hallo Welt!');
}
> res = hallo()
> res
undefined
```

### 33.1.3 Argumente und Parameter von Funktionen

#### ■ Grundsätzliches zu Argumenten und Parametern

Die Funktionen der Mathematik sind Zuordnungsvorschriften, die einem oder mehreren Argumenten einen Wert zuweisen. Wir haben bislang nur Funktionen betrachtet, die gänzlich ohne Argumente ausgekommen sind. In der Regel werden Funktionen aber gerade dadurch nützlich, dass man sie mit Argumenten aufrufen und ihr Verhalten dadurch steuern oder ihnen auf diese Weise Daten zur Verarbeitung übergeben kann.

Terminologisch waren wir bisher weniger trennscharf, als man es bei JavaScript häufig sieht. Hier wird nämlich nicht selten zwischen *Parametern* und *Argumenten* unterschieden. Parameter sind die in der Funktionsdefinition aufgeführten abstrakten Größen, die eine Funktion entgegennimmt, Argumente die konkreten Werte, die ihr bei einem *Aufruf* tatsächlich übergeben werden.

Parameter werden in JavaScript in der Funktionsdefinition ohne Typ angegeben. Das bedeutet, dass sich der Programmierer selbst darum kümmern muss, etwaig notwendige Typüberprüfungen vorzunehmen.

Das bereits zuvor verwendete Beispiel der Temperaturumrechnung von Kelvin in Grad Celsius würde unter Verwendung von Funktionsargumenten (aber ohne den Typ des Parameters zu prüfen) dann so aussehen:

```
function kelvinToCelsius(kelvin) {
    return kelvin - 273.15;
}
```

**kelvin** ist dabei ein *Parameter* der Funktion. Rufen wir die Funktion dann später beispielsweise mit der Kelvin-Temperatur **54** auf, dann ist **54** das *Argument* für den Parameter **kelvin**.

Mehrere Parameter würden in der Funktionsdefinition durch Kommata getrennt werden.

## ■ Änderung von Argumenten innerhalb der Funktion

In JavaScript werden Argumente immer *als Werte* übergeben, wenn es sich bei den Datentypen der Argumente um elementare Datentypen, das heißt *primitives*, wie **number**, **string** oder **boolean** handelt; übergibt man also einer Funktion eine Variable und verändert die Funktion das ihr übergebene Argument, ändert sich an der ursprünglichen Variablen nichts. Die Funktion arbeitet gewissermaßen mit einer Kopie des ihr übergebenen Werts, nicht mit der übergebenen Variablen selbst. Die Übergabe findet also tatsächlich *by value* statt.

Übergibt man der Variablen aber ein komplexeres Objekt und nimmt daran eine Änderung vor, ändert sich das der Funktion übergebene Objekt sehr wohl. Die Übergabe erfolgt dann also *by reference*.

Das folgende Beispiel verdeutlicht den Unterschied:

```
function Produkt(preis, bezeichnung) {  
    this.preis = preis;  
    this.bezeichnung = bezeichnung;  
}  
  
var stuhl = new Produkt(24.99, 'Gartenstuhl');  
var gekauft = false;  
  
function setzePreis(artikel, preis) {  
    artikel.preis = preis;  
}  
  
function kaufen(kaufStatus) {  
    kaufStatus = true;  
}
```

Im Beispiel nutzen wir wieder den Objekt-Typ **Produkt**, dessen Konstruktorfunktion wir zunächst aufrufen, um ein **Produkt**-Objekt namens **stuhl** zu erzeugen und einige Eigenschaften zu initialisieren, darunter den Preis, der auf **24.99** gesetzt wird. Auch erzeugen wir eine **boolean**-Variable **gekauft**, die anzeigt, ob bereits etwas verkauft worden ist und zunächst mit **false** initialisiert wird. Es folgen die Definitionen zweier Funktionen: **setzePreis(artikel, preis)**, die den Preis eines ihr als erstes Argument übergebenen Produkts ändert, und **kaufen(kaufStatus)**, die eine **boolean**-Variable als Verkaufsindikator übernimmt und diese auf **true** setzt (oder auch nicht, wie wir noch sehen werden). Probieren wir beide Funktionen in der Konsole aus:

```
> setzePreis(stuhl, 50.89)  
> kaufen(gekauft)  
> stuhl.preis  
50.89  
> gekauft  
false
```

Wie Sie sehen, ändert sich tatsächlich der Preis des Stuhls, der Status-Indikator **gekauft** behält aber seinen alten Wert. **gekauft** ist ein *primitive*-Wert, er kann als Argument der Funktion (**kaufStatus**) nicht geändert werden; das Objekt **stuhl** dagegen kann sehr wohl im Code der Funktion geändert werden, wenn es der Funktion als Argument **artikel** übergeben wird.

### ■ Aufruf von Funktionen mit Argumenten

Funktionsargumente können beim Funktionsaufruf auch mit ihrem Parameternamen übergeben werden, in unserem obigen Beispiel also etwa **kelvinToCelsius(kelvin=54)**. Da in diesem Fall die Zuordnung der Argumente zu den Parametern klar ist, erlaubt es dieses Vorgehen auch, die Argumente in einer anderen Reihenfolge als der in der Funktionsdefinition angegebenen Parameterfolge zu übergeben.

JavaScript ist sehr flexibel, was die Zahl der Parameter angeht: Werden der Funktion *zu viele* Argumente übergegen, werden die „überschüssigen“ Argumente einfach *ignoriert*. Die Frage, was im Fall von *zu wenigen* Argumenten geschieht, führt uns zur Thematik der Standardwerte und optionalen Parameter.

### ■ Standardwerte und optionale Parameter

Parameter können, wie in den meisten anderen Programmiersprachen auch, mit einem Standardwert versehen werden, der immer dann zum Tragen kommt, wenn kein Argument, also kein konkreter Wert, für diesen Parameter beim Funktionsaufruf angegeben wird.

Anders jedoch als in vielen anderen Programmiersprachen macht nicht *dieser Standardwert* einen Parameter erst zu einem optionalen Parameter. Stattdessen sind in JavaScript *alle* Parameter optional. Wenn Parameter keinen Standardwert besitzen und dennoch nicht beim Funktionsaufruf angegeben werden, wird ihnen automatisch der Wert **undefined** zugewiesen. In unserem Beispiel also wäre ein Aufruf der Temperaturumrechnungsfunktion von oben als **kelvinToCelsius()** durchaus zulässig, würde aber zum Rückgabewert **NaN (not a number)** führen, weil JavaScript in diesem Fall für den Parameter **kelvin** den Wert **undefined** annimmt, und mit diesem nicht arithmetisch gerechnet werden kann.

33

### ■ Funktionen als Argumente von Funktionen

Die Parameter von Funktionen können von jedem beliebigen Objekt-Typ sein. Sie können also auch selbst *Funktionsobjekte* sein. Das sollte Sie nach dem zuvor Besprochenen eigentlich nicht weiter überraschen. Wir wollen diesen Sachverhalt hier aber mit einem Beispiel verdeutlichen, um zu zeigen, dass dies eine sehr nützliche Eigenschaft von JavaScript ist.

Betrachten Sie die folgenden beiden Funktionen, die den Vornamen und den Nachnamen eines Schülers auf jeweils unterschiedliche Weisen verkettet:

```
function nachVor(vorname, nachname) {
    return nachname + ', ' + vorname;
}
function vorNach(vorname, nachname) {
```

```
    return vorname + ' ' + nachname;  
}
```

Wir könnten nun eine Funktion **zeigeNote()** entwickeln, die neben dem Namen des Schülers und seiner Note auch eine Funktion übernimmt, die sich um die Namensdarstellung kümmert:

```
function zeigeNote(vorname, nachname, note, namenAnzeigen) {  
    console.log(  
        namenAnzeigen(vorname, nachname) + ': ' + note);  
}
```

**zeigeNote()** könnten wir nun zum Beispiel folgendermaßen aufrufen:

```
> zeigeNote('Ulrike', 'Mayer', '1.0', nachVor);  
Mayer, Ulrike: 1.0  
> zeigeNote('Ulrike', 'Mayer', '1.0', vorNach);  
Ulrike Mayer: 1.0
```

**zeigeNote()** greift also auf die ihr als Argument übergebene Funktion **namenAnzeigen()** zu. Das macht **zeigeNote()** äußerst flexibel. Denn solange die übergebene Funktion Vor- und Nachnamen in dieser Reihenfolge entgegennimmt, kann sie zur Namensdarstellung verwendet werden, auch dann, wenn sie gar nicht vom Entwickler der Funktion **zeigeNote()** stammt.

#### ■ Variable Anzahl von Argumenten

JavaScript stellt mit der in allen Funktionen vorhandenen Objekt-Eigenschaft **arguments** eine einfache Möglichkeit zur Verfügung, auf die Argumente einer Funktion zuzugreifen, ohne den Namen der Parameter zu verwenden. Das ist insbesondere dann hilfreich, wenn man im Vorhinein noch nicht genau weiß, wie viele Argumente der Benutzer übergeben wird. Im folgenden einfachen Beispiel haben wir eine Funktion, der man eine Reihe von Zeichenketten als Argumente übergeben kann und die diese Zeichenketten als News-Ticker mit **+++** als Separator verbindet:

```
function newsTicker() {  
    var ticker = Array.from(arguments).join(' +++ ');\n    console.log(ticker)  
}
```

Sie lässt sich nun zum Beispiel so aufrufen:

```
> newsTicker('Bayern-Hannover 2:1', 'Dortmund-Leipzig 1: ')  
Bayern-Hannover 2:1 +++ Dortmund-Leipzig 1: 1
```

Statt zwei Spielergebnissen hätten wir auch eine *beliebige andere Zahl* von Paarungen liefern können. Unsere **newsTicker()**-Funktion würde damit zureckkommen, und das, obwohl sie auf den ersten Blick überhaupt keine Argumente erhält! Da JavaScript es aber zulässt, dass die Funktion mehr Argumente übergeben bekommt, als sie Parameter in ihrer Definition besitzt, können wir unseren News-Ticker einfach nach Herzenslust mit so vielen Fußballresultaten füttern, wie wir wollen.

**arguments** verhält sich in mancherlei Hinsicht wie ein Array. Zum Beispiel kann man auf die einzelnen Argumente, die der Funktion übergeben wurden, mit **arguments[0]**, **arguments[1]** etc. zugreifen. Ein echtes Array ist **arguments** aber nicht. Außer der **length**-Eigenschaft, die die Anzahl der übergebenen Argumente liefert, besitzt **arguments** keine der üblichen Array-Eigenschaften und -Methoden. Für unsere Zwecke müssen wir es zunächst in den **Array**-Typ konvertieren, der uns dann die Funktion **join()** zum Aneinanderhängen der einzelnen Elemente zur Verfügung stellt. Dabei wird hier die Funktion **from()** des **Array**-Objekts verwendet, die ein Array-ähnliches Objekt (wie **arguments**) in ein echtes Array umwandelt.

### 33.1 [5 min]

Definieren Sie ein Objekt **Produkt** mit Produktbezeichnung und Preis als Eigenschaften und entwickeln Sie für dieses Objekt eine Methode, die auf das Produkt einen Preisrabatt anwendet, den der Aufrufer der Methode als Parameter angeben kann. Gibt er keinen Preisrabatt an, so soll davon ausgegangen werden, dass der Preis um 20 % reduziert werden soll.

### 33.2 [5 min]

Schreiben Sie eine Funktion, der man ein **Produkt**-Objekt (aus der vorangegangenen Aufgabe) übergeben kann (die Funktion soll also keine Methode des Objekts sein) und die dann den als Parameter angegebenen Preisnachlass anwendet. Auf welche zwei Arten kann die „Rückgabe“ des geänderten Produkt-Objekts erfolgen, und warum?

33

### 33.3 [5 min]

Entwickeln Sie eine Funktion, die eine unbestimmte Anzahl von Argumenten übernimmt und diese als alphabetisch sortiertes Array wieder zurückgibt.

### 33.4 [5 min]

Schreiben Sie eine Funktion, die eine Ausgabe in der Konsole vornimmt und weisen Sie einer anderen Variable das so entstandene Funktionsobjekt zu. Rufen Sie die Funktion dann mit Hilfe dieser anderen Variable auf.

#### 33.1.4 Gültigkeitsbereich von Variablen in Funktionen

Insbesondere im Zusammenhang mit Funktionen spielt immer wieder der *Gültigkeitsbereich* von Variablen eine wichtige Rolle.

Variablen, die *innerhalb* einer Funktion mit **var** deklariert werden, sind *lokale* Variablen, die nur im Code-Block dieser Funktion existieren. Außerhalb der Funktion kann nicht auf sie zugegriffen werden.

Wird aber innerhalb einer Funktion *ohne var* auf eine Variable zugegriffen, so wird diese nur dann als neue lokale Variable erzeugt, wenn keine gleichnamige globale, das heißt außerhalb der Funktion (gewissermaßen „eine Ebene höher“), deklarierte Variable vorhanden ist.

Betrachten Sie dazu das folgende einfache Beispiel:

```
var faktor1 = 3, faktor2 = 5;

function multiplizieren() {
    var faktor2 = 7;
    res = faktor1 * faktor2;
    faktor1 = 11;
    faktor3 = 200;
    return res;
}

console.log(multiplizieren());
console.log(faktor1);
console.log(faktor2);
console.log(faktor3);
```

Dieses kleine Programm macht vier Ausgaben:

```
21
11
5
200
```

Innerhalb der Funktion **multiplizieren()** wird mit **var** eine *lokale* Variable **faktor2** erzeugt, deren Wert (7) vom Wert der gleichnamigen globalen Variable, die außerhalb der Funktion deklariert und initialisiert wurde (5) abweicht.

Nach dem Aufruf der Funktion geben wir den Wert der Variable **faktor2** in die Konsole aus. Dabei greifen wir automatisch auf die *globale* Variable **faktor2** zu, denn die lokale Variable gleichen Namens hat am Ende der Funktion **multiplizieren()** zu existieren aufgehört. Die lokale Variable mit dem Wert 7 ist es aber, die zur Berechnung der Multiplikation innerhalb der Funktion herangezogen wird. Sie schirmt gewissermaßen die globale Variable **faktor2** ab; diese ist dadurch nicht sichtbar. Als in der Anweisung **res = faktor1 \* faktor2** auf die Variable **faktor2** zugegriffen wird, wird also automatisch die lokale Variable verwendet. Nur, wenn keine lokale Variable existiert, wird nach einer globalen Variable gesucht, und im Falle von **faktor1** auch eine gefunden. Dieser wird dann ein neuer Wert zugewiesen. Da dabei nicht das Schlüsselwort **var** verwendet wird, erfolgt die Zuweisung an

die globale Variable. Diese Veränderung des Variablenwerts ist deshalb auch außerhalb der Funktion sichtbar, wie der zweite Output in der Konsole zeigt. Wäre die Zuweisung mit mit **var** eingeleitet worden, hätten wir statt einer Wertezuweisung an eine globale Variable eine *neue lokale* Variable mit dem Bezeichner **faktor1** geschaffen. Die Zuweisung des Wertes **11** wäre dann an diese lokale Variable gegangen, die globale Variable **faktor1** bliebe davon unberührt, sie würde durch die lokale Variable in ihrer Sichtbarkeit abgeschirmt und hätte ihren Wert auch nach dem Ausführen der Funktion **multiplizieren()** behalten.

Genau dies geschieht, wie man am dritten Output sieht, mit der Variable **faktor2**, der innerhalb der Funktion nur scheinbar ein neuer Wert zugewiesen wird; dieser neue Wert geht nämlich in Wirklichkeit an die neue *lokale* Variable **faktor2**, sodass die globale Variable durch die Zuweisung in ihrem Wert nicht verändert wird.

Interessant ist Output Nummer vier. Er greift auf die Variable **faktor3** zu, die das erste Mal in unserer Funktion **multiplizieren()** verwendet wird und zwar im Rahmen einer Zuweisung. Wir hatten ja bereits gesagt, dass Variablen, auf die innerhalb einer Funktion ohne das Schlüsselwort **var** zugegriffen wird, globale Variablen sind. Und genau so ist es auch mit **faktor3**: Durch die Zuweisung **faktor3 = 200** erzeugen wir eine globale Variable, obwohl die Zuweisung *innerhalb* der Funktion geschieht. Und weil **faktor3** eine globale Variable ist, können wir auf den darin abgelegten Wert auch außerhalb der Funktion ohne Probleme zugreifen.

Die Parameter, die Funktionen übergeben werden, sind übrigens stets lokale Variablen. Existiert eine globale Variable gleichen Namens so wird diese durch die Funktion praktisch maskiert, sie ist unsichtbar. Greift man mit ihrem Bezeichner auf die Variable zu, arbeitet man mit der gleichnamigen lokalen Variablen, also dem Parameter der Funktion.

### ?

### 33.5 [10 min]

Welchen Wert haben die Variablen **x**, **y**, **y1**, **y2** und **z** nach Ausführung des folgenden Programms, und warum?

33

```

x = 5;
z = 3;

function allesAnders(x, y) {
    y1 = x;
    x = null;
    var y2 = y;
    z = 1;
}

allesAnders(6,2);

```

## 33.2 Arbeiten mit Modulen/Bibliotheken

### 33.2.1 Eigene Module entwickeln und verwenden

Module verwenden bedeutet, Code einzubinden, der ausgelagert ist. Das Auslagern von Code macht insbesondere dann Sinn, wenn man den Code in unterschiedlichen Programmen einsetzen möchte. Wenn Sie zum Beispiel eine praktische Funktion entwickelt haben, und diese nicht nur in dem Programm einsetzen wollen, für die Sie sie ursprünglich entworfen haben, sondern auch in anderen Programmen, ist es das einfachste, diese Funktion in ein eigenes Modul auszulagern und dieses Modul dann in alle Programme einzubinden, die auf die Funktion zugreifen sollen.

Er einfachste Weg, eine andere JavaScript-Datei einzubinden, besteht darin, sie mit Hilfe des **script**-Elements in die Webseite einzubinden. Schauen wir uns genau das an einem Beispiel an.

In ► Abschn. 33.1.2 hatten wir eine Funktion **zufall()** entwickelt, die eine Zufallszahl zwischen 0 und 9 liefert. Nehmen wir an, wir wollten diese Funktion, weil sie praktisch ist und wir sie in unterschiedlichen Skripten verwenden möchten, in ein eigenes Modul auslagern.

Dazu erzeugen wir zunächst eine neue JavaScript-Datei **meinmodul.js**, in der wir die Funktion unterbringen. Darüber hinaus definieren wir noch eine Variable namens **ganzeZahl** in unserem Modul. Damit sie unsere Datei **meinmodul.js** dann so aus:

```
function zufall() {  
    var zufallsZahl;  
    zufallsZahl = Math.round(Math.random()*10, 0);  
    return zufallsZahl;  
}  
  
festeZahl = 4;
```

Zugegriffen wird auf die Funktion **zufall()** und die Variable **festeZahl** aus einem anderen JavaScript-Programm namens **modulverwender.js**:

```
document.write('Eine Zufallszahl: ', zufall());  
document.write('<p></p>');  
document.write('Eine feste Zahl: ', festeZahl);
```

Dieses Skript wiederum ist in eine einfache Webseite eingebunden:

```
<!DOCTYPE html>
<html>

    <head>
        <title>Skript mit eigenem Modul</title>
        <noscript>Bitte JavaScript aktivieren!</noscript>
    </head>

    <body>
        <script src="meinmodul.js"></script>
        <script src="modulverwender.js"></script>
    </body>

</html>
```

Damit die Funktion und die Variable, die wir in **meinmodul.js** definiert haben, auch in unserem eigentlichen Programm, das in der Datei **modulverwender.js** verfügbar ist, müssen wir das Modul ebenfalls in die Webseite einbinden, und zwar vor **modulverwender.js**. Da die Skripte der Reihe nach abgearbeitet werden, würde, wenn wir das Modul erst später einbinden würden, sein Inhalt zu dem Zeitpunkt, wo wir aus **modulverwender.js** heraus auf ihn zugreifen, noch nicht bekannt sein. Achten Sie also auf die Reihenfolge!

Wenn wir die Webseite nun öffnen, ergibt sich ein Output wie der folgende:

```
Eine Zufallszahl: 2
Eine feste Zahl: 4
```

Genau so könnten Sie nun das Modul **meinmodul.js** auch in andere Projekte einbinden und dort auf die Funktion **zufall()** zugreifen, ohne, dass Sie den Code der Funktion in ihrem neuen Projekt in der Haupt-Codedatei nochmal wiederholen müssten.

33

Die Möglichkeiten von JavaScript, mit Modulen zu arbeiten, haben sich über die Zeit immer stärker erweitert. Waren anfangs nur einfache Konstrukte möglich, bei denen lediglich mehrere JavaScript-Quelldateien in die Webseite eingebunden werden (wie wir es gerade getan haben), wird mit neueren Sprachstandards echte Modularisierung möglich, die unter anderem eine genauere Steuerung erlaubt, welche Objekte von Modulen exportiert werden und dann von anderen Modulen (und damit auch von den eigentlichen JavaScript-Anwendungen selbst) zur Verwendung wieder importiert werden können. Beim Import stehen auch verschiedene Möglichkeiten zur Verfügung, um mit Namenskonflikten umzugehen, die dann entstehen können, wenn die Module Objekte (zum Beispiel Funktionen) bereitstellen, deren Bezeichner identisch mit den Bezeichnern von im importierenden Code bereits vorhandenen Objekten sind.

Diese Überlegungen führen aber hier über den Umfang einer Einführung in die Entwicklung mit JavaScript deutlich hinaus. Für uns genügt an dieser Stelle zu wissen, dass wir unseren Code dadurch modularisieren können, dass wir ihn in einzelne Skript-Dateien aufteilen und diese dann in unsere HTML-Seite einbinden.

### 33.2.2 Externe Module/Bibliotheken finden und einbinden

---

Anders als für manche anderen Programmiersprachen (wie etwa Python mit dem *Python Package Index*, vgl. ► Abschn. 23.3.3) existiert für JavaScript keine zentrale (quasi-)offizielle Plattform, auf der Sie nützlichen Code finden, den Sie bei der Entwicklung Ihrer eigenen Anwendungen verwenden können. Dennoch gibt es natürlich einige „Hotspots“ der sehr aktiven JavaScript-Community, allen voran *GitHub*, auf das wir bereits in ► Abschn. 13.2 einen Blick geworfen hatten und das eine Plattform zum Austausch und zur Zusammenarbeit unter Entwicklern ist, die auf dem von Linux-Erfinder Linus Torvalds entwickelten Versionierungswerkzeug *git* basiert. Hier finden Sie eine Unmenge von *Repositories*, also Code-Archiven, mit unzähligen nützlichen Funktionen und Klassen.

Informieren Sie sich über die Lizenz-Situation anhand der **LICENSE**-Datei im Repository, bevor Sie fremden Code verwenden! Die meisten Entwickler, die ihre Werke auf *GitHub* zur Verfügung stellen nutzen eine der bekannten Standard-Lizenzen wie *GNU General Public License* oder *Creative Commons* oder *MIT*, die jeweils meist in diversen Varianten und Versionen daherkommen. Praktischerweise bietet Ihnen *GitHub* stets eine Zusammenfassung dieser Standard-Lizenzen, die zeigt, was Sie mit dem Code tun dürfen, und was nicht. Sie müssen also regelmäßig keineswegs lange englische Rechtstexte lesen, um zu verstehen, wie Sie von der Vorarbeit der anderen Entwickler Gebrauch machen dürfen. Diese Arbeit hat *GitHub* für Sie bereits erledigt.

In einem *GitHub*-Repository liegt in aller Regel eine Vielzahl von Dateien. Wundern Sie sich nicht, wenn Ihnen die vermeintlichen Beschreibungen dieser Dateien in der mittleren Spalte einer Repository-Code-Ansicht seltsam vorkommen. Diese Spalte beinhaltet keineswegs eine Beschreibung der Dateien, sondern sogenannte Commit-Kommentare, die die letzte Änderung an der Datei erläutern. Die in der Praxis wichtigsten Dateien liegen regelmäßig im `\dist`-Verzeichnis des Repositories, es sind die *distributable* Dateien, also jene Code-Dateien, die zur Verteilung und zum produktiven Einsatz gedacht sind. Wenn Sie den Code lesen wollen, empfiehlt es sich, einen Blick in den `\src`-Verzeichnis zu werfen. Der Code in `\dist` ist nämlich in der Regel um überflüssige Zeichen bereinigt (zum Beispiel Leerzeichen und Kommentare), um die Dateigröße möglichst gering zu halten und die Performance der Webseite, die diesen Code verwendet, zu verbessern. Manchmal wurde der Code auch *obfuscated*, also auf eine gewisse Weise verschlüsselt (blättern Sie nochmal einige Seiten zurück zu ► Abschn. 29.1.2, wo wir *Obfuscation* besprochen hatten). Sie können aber natürlich auch ohne weiteres mit dem Code im `\src`-Verzeichnis arbeiten.

Der einfachste Weg, den Code für Sie nutzbar zu machen, über den „Close or download“-Button eine ZIP-Datei herunterzuladen und diese dann auf Ihrem

Computer entpacken. Sie enthält das gesamte Repository. Den Code zur Verfügung zu haben, ist zwar notwendig, wichtig ist aber auch zu verstehen, wie Sie ihn benutzen. Auskunft dazu gibt regelmäßig die **README**-Datei des jeweiligen Projekts und ggf. weitere Dateien im `\doc`-Folder.

Neben *GitHub* gibt es natürlich zahlreiche weitere Quellen, aus denen Sie JavaScript-Module beziehen können, etwa ► [javascripting.com](https://javascripting.com). Auch eine gezielte Internet-Suche, in der Sie nach dem Schema „How can I...“ bringen Sie nicht selten zu Antworten, die auf Module verweisen, die Sie von irgendwo her herunterladen können.

Eine der populärsten JavaScript-Bibliotheken ist übrigens *jQuery*, das insbesondere die Arbeit mit dem Objektmodell des Browser (also zum Beispiel das Selektieren und Verändern von HTML-Elementen) vereinfacht.

### 33.3 Frameworks

---

Bei der professionellen Anwendungsentwicklung mit JavaScript spielen heute *Frameworks* eine große Rolle. Frameworks unterscheiden sich von normalen Programmzbibliotheken/Modulen dadurch, dass sie einen Rahmen für die Anwendung bilden, der den vom Entwickler geschriebenen Code immer dann aufruft, wenn das notwendig ist. Bei der Verwendung von Bibliotheken hingegen ist es der Code des Entwicklers, der die Bibliothek aufruft, wenn sie gebraucht wird. Man spricht deshalb im Zusammenhang mit Frameworks von einer *inversion of control*. Frameworks spielen in der Praxis eine große Rolle, weil sie es dem Programmierer erlauben, sich auf die Kernfunktionalität seiner Anwendung zu konzentrieren und andere, eher standardisierte Aufgaben wie etwa die Abwicklung von Logins und Session-Management oder die Abfrage von Daten aus Datenbanken und ihre Darstellung in Template-artigen Seiten an das Framework abzugeben.

Einige bekannte JavaScript-Frameworks sind *Angular*, *React* und *Vue*. Eine Betrachtung ihrer Struktur und Verwendung würde den Rahmen dieses Buchs bei weitem sprengen. Voraussetzung für den Einsatz von Frameworks ist aber die Kenntnis der JavaScript-Grundlagen und genau diese sind es, die wir in diesem Teil des Buches behandeln. Nach dem Studium dieses Teils haben Sie also eine solide Grundlage, auf deren Basis Sie sich auch an die Arbeit mit Frameworks heranwagen können, sollten Sie das wünschen. Tatsächlich genügt aber für die meisten Zwecke im „Privatanwender-Bereich“ (Hobby-Programmierer ist kein wirklich schönes Wort!) die Verwendung von Standard-JavaScript ohne Einsatz von Frameworks, die zwar zweifelsohne mächtig sind, aber auch einen gewissen „Overhead“ an Arbeit mitbringen und eine Struktur erzwingen, die für kleinere Projekte nicht vonnöten ist.

### 33.4 Zusammenfassung

---

In diesem Kapitel haben wir uns mit Funktionen beschäftigt, und gesehen, wie Funktionen definiert und verwendet werden. Darüber hinaus haben wir uns mit der Erweiterung des Funktionsumfangs durch externe Bibliotheken befasst und sind der Frage nachgegangen, was Frameworks sind und wie sie sich von „herkömmlichen“ Bibliotheken unterscheiden.

Folgende Punkte sollten Sie aus diesem Kapitel unbedingt mitnehmen:

- Funktionen werden meist mit dem Schlüsselwort **function** definiert.
- Sie können Argumente übernehmen (deren konkrete Werte beim Aufruf als Parameter bezeichnet werden), die mit einem Standardwert versehen werden können.
- JavaScript ist sehr flexibel, was die Übergabe von Parametern beim Funktionsaufruf angeht; so kann eine Funktion sowohl mit mehr als auch mit weniger Parametern aufgerufen werden, als sie Argumente besitzt; dementsprechend ist praktische jedes Argument einer Funktion ein optionales.
- Argumente können sowohl als Positionsargument, das heißt, anhand ihrer Position in der Reihenfolge der Argumente-Liste, als auch als Schlüsselwortargument, das heißt, unter Angabe ihres Bezeichners, übergeben werden.
- Funktionen können mit der Anweisung **return** (oder der Funktion **return()**) Werte zurückgeben; Funktionen, die nicht explizit einen Wert zurückgeben, liefern **undefined**.
- Funktionen sind in JavaScript Objekte vom Typ **function**.
- Sie können daher auch Variablen zugewiesen werden. Insbesondere können auf diese Weise für Objekte Methoden definiert werden; die Methode eines JavaScript-Objekts ist letztlich nichts anderes als ein Attribut des Objekts, nämlich ein (aufrufbares) Attribut vom Typ **function**.
- Weil Funktionen Objekte sind, können Sie auch anderen Funktionen als Argument übergeben werden.
- Variablen, die innerhalb einer Funktion mit Schlüsselwort **var** definiert werden, sind lokale Variablen, die aufhören zu existieren, sobald die Ausführung der Funktion beendet ist; sie sind dementsprechend aus dem Hauptprogramm heraus nicht sichtbar. Auch die Argumente von Funktionen werden als lokale Variable betrachtet. Eventuell gleichnamige globale Variablen werden von diesen lokalen Variablen gegen Zugriff „abgeschirmt“.
- Wird dagegen innerhalb einer Funktion eine Variable erzeugt, ohne dabei das Schlüsselwort **var** zu verwenden, so entsteht eine globale Variable, die auch außerhalb des Funktionskörpers sichtbar ist. Existiert eine globale Variable dieses Namens bereits, so wird auf diese Variable zugegriffen.
- Über den Standardfunktionsumfang von JavaScript hinaus, können Sie Erweiterungsbibliotheken installieren; für diese existiert zwar keine offizielle Bezugsquelle, wie für viele einige Programmiersprachen, jedoch gibt es mit Seiten wie ► [javascripting.com](http://javascripting.com) durchaus Plattformen, auf denen Entwickler eine große Menge an Bibliotheken für die unterschiedlichsten Zwecke bereitstellen.

- Frameworks spielen in der praktischen (vor allem der professionellen) JavaScript-Entwicklung eine große Rolle, weil sie bei der Entwicklung komplexer Anwendungen zu einer signifikanten Arbeitserleichterung führen und es dem Entwickler erlauben, sich auf das Wesentliche zu konzentrieren. Sie bieten einen Rahmen, in den der Entwickler seinen Code einbetten kann. Der Rahmen organisiert den Ablauf der Anwendung, der Code des Entwicklers wird vom Rahmen dort aufgerufen, wo es nötig ist. Weil hier – anders als „normalerweise“ – die Steuerung der Anwendung nicht beim Entwickler, sondern beim Framework liegt, spricht man auch von einer „inversion of control“.

### 33.5 Lösungen zu den Aufgaben

---

#### ■ Aufgabe 33.1

```
produkt = {
    bezeichnung: 'Gartenstuhl',
    preis: 24.99
};

produkt.rabatt = function (rabattProzent = 20) {
    this.preis = this.preis * (1 - rabattProzent / 100);
```

Zunächst erzeugen wir das Objekt mit seine Attributen **bezeichnung** und **preis**. Dem Objekt fügen wir dann die Funktion **rabatt()** hinzu; genauer gesagt, weisen wir der Eigenschaft **rabatt** ein Funktionsobjekt zu, dass den angegebenen Code besitzen soll. Das Funktionsobjekt ist fortan unter dem Bezeichner **rabatt** aufrufbar, nämlich eben als Methode **rabatt()**.

Ebenso hätten wir unser Objekt **produkt** natürlich mit einer Konstruktorfunktion erzeugen können, wie wir es in ► Abschn. 33.1.3 gesehen haben. Die Definition der Eigenschaft **rabatt** hätten wir dann in den Konstruktor mit aufnehmen können und hätten dann lediglich das Funktionsobjekt an **this.rabatt** anstatt **produkt.rabatt** zuweisen müssen.

33

#### ■ Aufgabe 33.2

```
function rabbattieren(prod, rabattProzent = 20) {
    prod.preis = prod.preis * (1-rabattProzent/100);
    // return prod;
}
```

Diese Funktion ist keine Methode des Objekts **produkt**. Sie übernimmt lediglich ein **produkt**-Objekt und passt seinen Preis an. Das ist möglich, weil in JavaScript Objekte als Funktionsparameter *by reference* übergeben werden, das heißt, wir haben mit dem Parameter **prod** einen direkten Zugriff auf das übergebene Objekt. Anders dagegen wäre der Fall gelagert, wenn das Argument ein *primitive* wäre, also

etwa eine Zahl oder eine Zeichenkette. Dann würde die Übergabe des Parameterwerts *by value* erfolgen; Änderungen, die wir an diesen Parametern vornehmen würden, hätten dementsprechend keinerlei Auswirkung auf die übergebene Variable.

Statt mit der Parameterübergabe *by reference* zu arbeiten, könnten wir natürlich auch das geänderte **produkt**-Objekt mittels **return**-Anweisung an den Aufrufer zurückgeben (im Lösungsvorschlag oben auskommentiert). Diese Möglichkeit bestünde übrigens natürlich auch dann, wenn der Parameter ein *primitive*-Wert wäre und *by value* übergeben würde.

### ■ Aufgabe 33.3

Die Funktion könnte zum Beispiel so aussehen:

```
function arrayErzeugenUndSortieren() {  
    return Array.from(arguments).sort();  
}
```

Hierbei nutzen wir das Objekt **arguments**, dass für jede Funktion die übergebenen Parameterwerte enthält. Obwohl es selbst kein echtes Array ist, lässt es sich mit der Funktion **Array.from()** in ein Array umwandeln und dann sortieren.

Danach könnten wir die Funktion zum Beispiel so aufrufen:

```
> arrayErzeugenUndSortieren('Hallo', ' ', ' ', 'ich', 'werde',  
  'zu', 'einem', 'sortierten', 'Array')  
[" ", "Array", "Hallo", "einem", "ich", "sortierten", "werde",  
 "zu"]
```

### ■ Aufgabe 33.4

```
var meineFunktion = function () {  
    console.log(  
        'Diese Funktion wird tatsächlich aufgerufen. ');  
}  
var meineFunktion2 = meineFunktion;  
  
meineFunktion2();
```

Hier weisen wir zunächst der Variablen **meineFunktion** einen Funktionsausdruck zu; die Variable **meineFunktion** ist damit ein Objekt vom Typ **function**. Wie jedes andere Objekt auch, können wir dieses Objekt nun einer anderen Variable zuweisen, in unserem Beispiel der Variable **meineFunktion2**. Mittels dieser Variablen lässt sich die Funktion (denn nichts anderes repräsentiert ja das **function**-Objekt) aufrufen. Wie bei jedem Funktionsaufruf müssen dabei natürlich die runden

Klammern angegeben werden, selbst dann, wenn die Funktion keine Parameter entgegennimmt.

#### ■ Aufgabe 33.5

Unser Programm führt zu folgenden Werten der Variablen:

- **x = 5:** x wird als globale Variable mit dem Wert **5** initialisiert. Innerhalb der Funktion wird dann der gleichnamige Parameter auf **null** gesetzt. Die Parameter der Funktion sind aber *lokale Variablen*. Greifen wir also innerhalb der Funktion auf x zu, arbeiten wir nicht mit der globalen Variable x, sondern mit der lokalen Variable gleichen Namens, nämlich dem Parameter x, mit dem die Funktion aufgerufen wurde (und der am Ende des Funktionskörpers aufhört, zu existieren). Deshalb bleibt der Wert der globalen Variable unverändert.
- **y** existiert nicht: Auch die Variable y ist innerhalb der Funktion eine *lokale Variable*. Sie hört nach vollständiger Ausführung der Funktion auf, zu existieren. Da keine globale Variable gleichen Namens existiert, können wir nach Abschluss der Ausführung unserer Funktion nicht mehr auf eine Variable mit dem Bezeichner y zugreifen.
- **y1 = 6:** Die Variable **y1** wird zwar innerhalb der Funktion erzeugt, aber ohne das Schlüsselwort **var**. Deshalb entsteht bei der Zuweisung **y1 = x** eine neue *globale Variable*, die auch nach Beendigung der Funktion noch verfügbar ist.
- **y2** existiert nicht: Ähnlich wie **y1** wird auch **y2** innerhalb der Funktion durch Zuweisung erzeugt, allerdings unter Verwendung des Schlüsselworts **var**. Dadurch entsteht bei der Zuweisung keine globale Variable, wie das bei **y1** der Fall gewesen ist, sondern eine *lokale Variable*.
- **z = 1:** z ist eine globale Variable, die beim Aufruf der Funktion den Wert **3** besitzt. Innerhalb der Funktion wird z dann auf den Wert **1** gesetzt. Da die Zuweisung ohne das Schlüsselwort **var** erfolgt (das eine neue lokale Variable mit diesem Bezeichner erzeugt hätte), manipulieren wir hier also die *globale Variable*.



# Wie steuere ich den Programmablauf und lasse das Programm auf Benutzeraktionen und andere Ereignisse reagieren?

## Inhaltsverzeichnis

- 34.1 if-else-Konstrukte – 558
- 34.2 switch-case-Konstrukte – 561
- 34.3 Ereignisse – 562
- 34.4 Lösungen zu den Aufgaben – 566
- 34.5 Zusammenfassung – 568

## Übersicht

Wir haben bereits mit zwei verschiedenen Arten, im Programmablauf zu verzweigen, gearbeitet; linear mit **if-else**-Konstrukten und ereignis-gesteuert mit Event Handlern. Beide werden wir uns jetzt noch etwas genauer ansehen.

In diesem Kapitel werden Sie lernen:

- wie Sie Bedingungen mit Hilfe von **if-else**-Konstrukte verwenden, im Programm zu verzweigen
- wie Sie dazu Bedingungen formulieren und miteinander verknüpfen
- wie Sie den Bedingungsoperators **?** anstatt eines **if-else**-Konstrukt benutzen, und wann das Sinn macht
- wie Sie mit **switch-case**-Konstrukten viele ähnlich strukturierte Bedingungen effizient überprüfen und entsprechend verzweigen können
- wie Sie Event Handler einsetzen, um Ereignisse zu bearbeiten
- welches die wichtigsten Arten von Ereignissen sind und wie Sie Informationen zu einem aufgetretenen Ereignis auswerten.

### 34.1 if-else-Konstrukte

#### ■ Formulierung mit Hilfe der Schlüsselwörter if und else

Im Beispiel der Kelvin-Celsius-Umrechnung aus ► Abschn. 32.5.2 prüfte die Funktion **umrechnen()**, die ereignisgesteuert ausgelöst wurde, wenn der Benutzer auf den entsprechenden Button klickte, ob von Celsius nach Kelvin umgerechnet werden sollte, oder umgekehrt – das nämlich konnte der Benutzer mit Hilfe zweier Radiobuttons festlegen. Der Code der Funktion sah so aus:

```
function umrechnen() {
    var temp = Number(
        document.getElementById('temp').value);
    var richtung = document.getElementsByName('richtung');

    if (richtung[0].checked == true) {
        document.write(`<p>${temp} Kelvin in Grad Celsius sind:
                      ${temp - 273.15} Grad Celsius.</p>`);
    }
    else {
        document.write(`<p>${temp} Grad Celsius in Kelvin sind:
                      ${temp + 273.15} Kelvin.</p>`);
    }
}
```

34

Hier sieht man sehr schön die Verzweigung innerhalb des Codes der Funktion: *Wenn (if) richtung[0].checked == true* (also Radiobutton Nummer 1 markiert ist) *dann* erfolgt eine Umrechnung von Kelvin in Celsius, *andernfalls (else)* wird von Celsius nach Kelvin umgerechnet.

Der allgemeine Aufbau des **if-else-Konstrukts** lässt sich hier leicht erkennen:

```
if(bedingung) {  
    // Anweisungen  
}  
else {  
    // Anweisungen  
}
```

**bedingung** ist dabei ein Ausdruck, der zu **true** oder **false** ausgewertet werden kann. Häufig sind die Ausdrücke *Vergleiche* wie im Beispiel. Dabei ist zu beachten, dass in JavaScript (wie in vielen anderen Programmiersprachen auch – Python ist hier ebenfalls keine Ausnahme) der Gleichheitsoperator als **==** und der Ungleichheitsoperator als **!=** (also „Nicht-Gleich“) geschrieben wird.

Auch in JavaScript kann ein Vergleich mit **true** oder **false** entfallen. Statt also wie im Beispiel oben **if(richtung[0].checked == true)** zu schreiben, hätte bereits **if(richtung[0].checked)** genügt, weil **richtung[0].checked** ein Ausdruck ist, der sich zu **true** oder **false** auswerten lässt; der Vergleich mit **true** wird also standardmäßig durchgeführt, auch dann, wenn wir ihn nicht explizit hinschreiben. Letztlich sind ja auch andere Vergleiche Ausdrücke, die zu **true** oder **false** ausgewertet können. Sie könnten – auch wenn es etwas umständlich wäre – nämlich statt **if(x>5)** auch **if((x>5) == true)** schreiben; **(x>5)** ist hier ein Ausdruck, der wahr oder falsch sein kann, je nachdem welchen Wert **x** annimmt.

Übrigens werden die speziellen Werte **null**, **undefined** und **NaN** in Bedingungen stets als **false** bewertet. Manche Funktionen mögen einen dieser Werte zurückgeben; besteht Ihr zu prüfender Ausdruck aus dem Aufruf einer solchen Funktion, sollten sie also überlegen, ob Ihr Programm im Fall einer solchen Rückgabe wirklich so verzweigt, wie Sie es wollen. Zeichenketten werden immer als **true** betrachtet (selbst dann, wenn die Zeichenkette '**false**' oder '**0**' lautet (hier findet also keine implizite Konvertierung statt)), Zahlen immer als **true**, es sei denn, es handelt sich um die **0**; sie wird als **false** bewertet.

Beachten Sie bitte, dass die Bedingung – anders als etwa in Python – immer in *runde Klammern* eingeschlossen sein muss. Innerhalb des zu prüfenden Ausdrucks können natürlich zusätzlich Klammern verwendet werden. Das ist insbesondere dann ratsam, wenn Sie komplizierte Ausdrücke mit vielen Operatoren verwenden und Sie nicht ganz genau wissen, in welcher Reihenfolge die Operatoren abgearbeitet werden; um dann eine bestimmte Reihenfolge sicherzustellen, schadet es nicht, ausreichend Klammern zu verwenden. Besser eine Klammer zu viel, als eine zu wenig!

Das gilt auch dann, wenn Sie mit zusammengesetzten Bedingungen arbeiten, also solchen, die aus mehreren Teilbedingungen zusammengesetzt sind. Die Teilbedingungen werden dann mit den logischen Operatoren **&&** (logisches *UND*) und **||** (logisches *ODER*) mit einander verknüpft. Wollten Sie also beispielsweise prüfen, ob die Variable **alter** zwischen 18 und 68 liegt, würde die passende Bedingung lauten: **if(alter >= 18 && alter <=68)**. Wenn Ihre Bedingung zusätzlich das Geschlecht der betreffenden Person mit berücksichtigen und immer dann erfüllt sein

soll, wenn die Person eine Frau ist oder das Alter der Person zwischen 18 und 68 Jahren liegt, würden Sie die Bedingung folgendermaßen formulieren: `if((alter >= 18 && alter <=68) || geschlecht == 'f')`. Beachten Sie hier die Klammersetzung, mit der wir sicherstellen, dass `(alter >= 18 && alter <=68)` als Teilbedingung zuerst ausgewertet wird. Die Klammern um den ersten Bedingungsausdruck wären zwar hier gar nicht notwendig gewesen, weil die logischen *UND*- und *ODER*-Operatoren einfach von links nach rechts in der Reihenfolge verarbeitet werden, wie sie im Code erscheinen, und deshalb nie die Gefahr bestanden hätte, dass `alter <=68 || geschlecht == 'f'` als Teilbedingung ausgewertet wird; dennoch macht die Schreibweise mit den Klammern deutlich, was hier zusammengehört und erhöht so die Lesbarkeit des Programmcodes.

Als dritter logischer Operator fehlt noch das logische *NICHT*, das in JavaScript als `!` geschrieben wird und den Wahrheitsgehalt einer Aussage herumdreht. Das logische *NICHT* ist im Gegensatz zum logischen *UND* und zum logischen *ODER* ein *unärer* Operator, er benötigt also nur einen Operanden (nämlich den Ausdruck, dessen Wahrheitswert herumgedreht wird); das logische *UND* und das logische *ODER* dagegen verknüpfen als *binäre* Operatoren zwei Operanden (hier: logische Ausdrücke) mit einander. Mit Hilfe des `!`-Operators könnten wir also die Bedingung `if((alter >= 18 && alter <=68) || geschlecht == 'f')` auch schreiben als `if((alter >= 18 && alter <=68) || !(geschlecht == 'm'))`. Dabei bedeutet `!(geschlecht == 'm')`, dass die Aussage `geschlecht == 'm'` nicht zutreffen soll. Im Umkehrschluss muss dann `geschlecht` den Wert `'f'` haben (zumindest, wenn wir wie üblich von zwei Geschlechtern ausgehen). Auf die Klammer um den zu verneinenden Ausdruck kann hier übrigens nicht verzichtet werden, denn sonst bezöge sich der `!`-Operator nur auf `geschlecht`; `geschlecht` aber besitzt als Zeichenkette immer den Wahrheitswert `true`, der durch `!` zu `false` verdreht würde. Damit reduziert sich der Ausdruck `!geschlecht == 'm'` zu `false == 'm'`, eine logische Aussage, die selbst falsch ist, weil `"m"` als Zeichenkette den Wahrheitswert `true` besitzt.

Nach der `if`-Bedingung und nach dem Schlüsselwort `else`, das die alternative Verzweigung einleitet, welche immer dann ausgeführt wird, wenn die `if`-Bedingung zu `false` evaluiert, folgt ein Code-Block in geschweiften Klammern. Enthält der Block nur eine einzelne Anweisung, können die geschweiften Klammern auch entfallen (das hätten Sie also auch in unserem Eingangsbeispiel tun können).

34

Ebenfalls entfallen kann der gesamte `else`-Zweig. Die Minimalform des `if-else`-Konstrukts beinhaltet also nur einen `if`-Zweig. Trifft die Bedingung, an deren Ausführung er gekoppelt ist, nicht zu, geschieht einfach gar nichts. Das Programm läuft normal weiter, beginnend mit der ersten Anweisung nach dem Code-Block des `if`-Zweigs.

#### ■ Formulierung mit Hilfe des Bedingungsoperators ?

Eine besondere Art, zu verzweigen, stellt der Bedingungsoperator `?` dar. Er ist ein *ternärer* Operator, also ein Operator der anders als beispielsweise der *unäre* Operator `!` (das logische *NICHT*) oder die *binären* Operatoren `&&` (das logische *UND*) und `||` (das logische *ODER*) sogar *drei* Operanden verarbeitet, daher die Bezeichnung *ternär*. Er prüft eine Bedingung und gibt als Ergebnis von zwei Ausdrücken ent-

weder den ersten oder den zweiten zurück, je nachdem, ob die Bedingung erfüllt war oder nicht.

Betrachten Sie dazu das folgende Beispiel, das wir der Einfachheit halber in der Konsole ausführen:

```
> name = 'Anderson'  
> geschlecht = 'm'  
> console.log("Hallo", (geschlecht == 'f') ? 'Frau' : 'Herr',  
name)
```

Hier wird in Abhängigkeit davon, ob die Person weiblich oder männlich ist, die Anrede entsprechend angepasst. Geprüft wird dabei die Bedingung (**geschlecht == 'f'**); ist diese erfüllt, gibt der Operator den Ausdruck hinter dem Fragezeichen zurück, anderenfalls den hinter dem Doppelpunkt. Die allgemeine Form der Anwendung des Bedingungsoperators ist also **bedingung ? rueckgabeWenn : rueckgabeSonst**. Vorteilhaft ist diese knappe Formulierung, weil sie eine kompaktere und leichter in andere Anweisungen integrierbare Art der Verzweigung bietet.

### ?

#### 34.1 [3 min]

Wie lässt sich einfach zeigen, dass Zeichenketten in Bedingungen immer als **false** ausgewertet werden?

### ?

#### 34.2 [5 min]

Formulieren Sie das Beispiel der Anredeformulierung so um, dass es nicht mehr den Bedingungsoperator **?:** verwendet, sondern ein herkömmliches **if-else**-Konstrukt.

### ?

#### 34.3 [5 min]

Formulieren Sie die Funktion **umrechnen()** zur Umrechnung von Temperaturen zwischen Kelvin und Grad Celsius so um, dass Kelvin-Temperaturen unter 0 Kelvin und Celsius-Temperaturen unter dem absoluten Nullpunkt von 273,15 Grad Celsius (= 0 Kelvin) mit einer Fehlermeldung quittiert werden.

## 34.2 switch-case-Konstrukte

Manchmal möchte man viele gleichartige Bedingungen auf einmal prüfen; dann ist ein verschachteltes **if-else**-Konstrukt zwar möglich, wird aber schnell sehr unübersichtlich. Deshalb existiert in JavaScript, wie in vielen anderen Sprachen auch, ein **switch**-Konstrukt.

Nehmen wir beispielsweise an, Sie wollten für einen gegebenen Monat die Zahl der Tage ermitteln. Mit ineinander verschachtelten **if-else**-Konstrukten würden Lesbarkeit und Wartbarkeit des Programmcodes deutlich leiden. Einfacher ist es dagegen mit **switch**:

```

monat = 'Dezember';

switch (monat) {
    case 'Januar', 'März', 'Mai', 'Juli', 'August',
        'Oktober', 'Dezember':
        tage = 31;
        break;
    case 'April', 'Juni', 'September', 'November':
        tage = 30;
        break;
    case 'Februar':
        tage = 28;
        break;
    default:
        tage = -1;
        break;
}

```

Die **switch**-Anweisung erhält in Klammern den zu prüfenden Ausdruck übergeben. Die einzelnen Fälle werden dann jeweils mit **case** eingeleitet. Dabei können auch mehrere Fälle auf einmal geprüft werden, wie wir es hier im Beispiel machen. Hinter den Doppelpunkt für geprüften Fall die Anweisungen, die ausgeführt werden sollen, wenn der Fall eintritt; in unserem Beispiel wird lediglich die Variable **tage** auf die entsprechende Tagesanzahl gesetzt. Der Anweisungsblock endet stets mit dem Schlüsselwort **break**.

Beim Ausführen ermittelt der Interpreter, welchen Wert der zu prüfende Ausdruck besitzt und springt direkt in den entsprechenden Anweisungsblock. Nach dem Abarbeiten des Anweisungsblocks wird die Programmausführung hinter dem **switch**-Konstrukt fortgesetzt. Trifft keiner der Fälle zu, wird der **default**-Block ausgeführt, der aber optional ist und daher auch weggelassen werden kann. Wird auf **default** verzichtet und keiner der Fälle trifft zu, so wird die Ausführung direkt hinter dem **switch**-Konstrukt fortgesetzt.

### 34.3 Ereignisse

## 34

Neben den **if-else**- und **switch**-Konstrukten sind *Ereignisse* die wichtigste Art, den Programmablauf zu steuern; tatsächlich sind sie in JavaScript sogar die wichtigste Form der Ablaufsteuerung überhaupt.

#### ■ Event Handler direkt in den HTML-Code einbauen

Mit Ereignissen haben wir bereits gearbeitet, als wir in ► Kap. 32 Ereignisbehandlungsrichtinen (Event Handler) an Formularelemente wie Buttons angehängt haben, um etwa auf einen Klick des Benutzers reagieren zu können. In diesem Abschnitt wollen wir uns die Arbeit mit Ereignissen noch etwas genauer ansehen.

Der folgende HTML-Code ist ein Auszug aus unserer Color-Picker-Anwendung aus Abschnitt ► Abschn. 32.7:

### 34.3· Ereignisse

```
<input id="colorRedRange" type="range" value="255" min="0" max="255"  
oninput="farbeAnpassen()">
```

Hier hatten wir bei einem **range**-Input-Element, also einen Schieberegler, die Eigenschaft **oninput** auf den Event Handler **farbeAnpassen()** gesetzt. Immer, wenn der Benutzer den Schieberegler bewegt, wird diese von uns entwickelte Funktion aufgerufen und kann auf die Benutzereingabe reagieren.

Das Ereignis, das hier mit einem Event Handler abgedeckt wird, heißt **input**; die entsprechende Eigenschaft des HTML-Elements trägt per Konvention den Namen **oninput**, dem Ereignisnamen wird also stets ein **on** vorangestellt, wie wir es an anderer Stelle zum Beispiel auch bei **onclick** gesehen haben.

Als Wert wird der Eigenschaft unser Event Handler zugewiesen, genauer gesagt, der *Aufruf* des Event Handlers – leicht zu erkennen an den runden Klammern. Statt dieses Aufrufs hätten wir hier auch direkt mehr JavaScript-Code eingeben können, zum Beispiel **oninput="alert('Veränderung!'); console.log('Veränderung!')"**. Dieses Vorgehen ist aber überhaupt nur bei sehr kurzen Code-Segmenten zu empfehlen und auch dann eigentlich nicht, denn es macht die Wartung des Codes natürlich schwieriger. Normalerweise wird man an dieser Stelle also den Aufruf eines Event Handlers sehen, wie in unserem Beispiel.

- **Event Handler über die HTML-Element-Eigenschaften im JavaScript-Code zuweisen**

Betrachten wir als nächstes das folgende einfache Beispiel. Zunächst das HTML-Dokument:

```
<!DOCTYPE html>  
<html>  
  <body>  
    <form>  
      <input id="eingabe" type="text">  
    </form>  
    <script src="eventtest.js"></script>  
  </body>  
</html>
```

Hier wird also lediglich ein Texteingabefeld mit der ID **eingabe** erzeugt. Ihnen ist vielleicht aufgefallen, dass wir das Skript dieses Mal ganz am Ende in das HTML-Dokument eingebunden haben. Der Grund liegt darin, dass wir, wie Sie gleich sehen werden, im Skript direkt auf Elemente der Webseite zugreifen. Hätten wir das Skript am Anfang eingebunden, würden wir bei diesen Zugriffsversuchen ins Leere greifen, weil die Elemente der Seiten zu diesem frühen Zeitpunkt ja noch gar nicht existieren. Das Skript würde dementsprechend auf einen Fehler laufen (probieren Sie es aus und beobachten Sie die Fehlermeldungen in der JavaScript-Konsole).

Die Programmlogik steckt in der JavaScript-Datei **eventtest.js**:

```
function mausklick(e) {
    console.log('Es wurde geklickt. ');
    console.log('X: ', e.x, '\nY: ', e.y);
}

var inpFeld = document.getElementById('eingabe');
inpFeld.onclick = mausklick;
```

In diesem JavaScript-Code wird zunächst unten das Eingabefeld der Webseite selektiert. Dann wird der Eigenschaft **onclick** dieses Elements die Funktion **mausklick** zugewiesen, die weiter oben definiert ist. Genauer gesagt, wird der Eigenschaft das Funktionsobjekt **mausklick** zugewiesen. Beachten Sie bitte, dass hier kein Aufruf der Funktion erfolgt, deshalb auch keine runden Klammern hinter dem Funktionsnamen.

Die Eigenschaft **onclick** unseres **inpFeld**-Objekts ist ein Beispiel dafür, dass die JavaScript-Objekte, die HTML-Elemente repräsentieren, Event-Handler-Eigenschaften besitzen, deren Namen nach der gleichen Logik gebildet werden, wie bei den HTML-Elementen selbst, also nach dem Schema **onereignis**.

Das Ereignis, das wir hier verarbeiten, ist das **click**-Ereignis. Die Verarbeitung übernimmt unser Event Handler **mausklick()**. Wie Sie sehen, übernimmt **mausklick()** dabei einen Parameter, ein *Ereignis-Objekt*, dass das eingetretene Ereignis näher beschreibt. Je nachdem, welches Ereignis verarbeitet wird, ist auch das Ereignis-Objekt anders zusammengesetzt. Im Falle des **click**-Ereignisses hat das Objekt unter anderem die Eigenschaft **x** und **y**, die angeben, wo genau auf der Seite der Klick stattgefunden hat. Das machen wir uns zunutze und geben diese Information in der Konsole aus.

Wenn Sie wissen wollen, welche Eigenschaften das Ereignis-Objekt für ein bestimmtes Ereignis besitzt, lassen Sie es sich einfach mit **console.log(e)** in der Konsole anzeigen.

Als wir die Event Handler direkt im HTML-Code mit den HTML-Elementen „verdrahtet“ hatten, wurden die Event Handler jeweils ohne ein solches Ereignis-Objekt aufgerufen, zum Beispiel bei **oninput="farbeAnpassen()"**. Ein Ereignis-Objekt hätten wir hier auch gar nicht übergeben können. Das war auch insoweit kein Problem, als dass wir in diesen Beispielen gar keinen Bedarf hatten, auf Eigenschaften des Ereignisses zuzugreifen. Hätten wir das aber machen wollen, so hätten wir innerhalb unseres Event Handlers auf das Standard-Objekt **event** zugreifen können, dass uns die Informationen dennoch zur Verfügung gestellt hätte. Strenggenommen bräuchten wir also unseren Parameter **e** gar nicht und könnten stattdessen immer mit dem Standard-Objekt **event** arbeiten.

Dass wir Funktionen, die eigentlich ein Ereignis-Objekt übergeben bekommen sollten, auch ohne ein solches aufrufen können und die Funktion sogar ganz ohne diesen Parameter definieren können, verdanken wir JavaScripts flexilem Umgang mit Funktionsparametern, mit dem wir uns bereits in ► Abschn. 33.1.3 beschäftigt haben.

Das **click**-Event ist nicht das einzige Ereignis rund um das Thema Mausklicks. Mit **mousedown** und **mouseup** stehen zwei Ereignisse zur Verfügung, die immer dann ausgelöst werden, wenn eine Maustaste gedrückt bzw. losgelassen wird. Die Ereignis-Objekte dieser beiden Ereignisse besitzen die Eigenschaft **buttons**, die mit **1** für linke Maustaste und **2** für rechte Maustaste anzeigt, welche Maustaste verwendet wurde. Das **click**-Event wird nach **mousedown** und **mouseup** ausgelöst. Bei einem Doppelklick zeigt die **detail**-Eigenschaft des Ereignis-Objekts des zweiten **click**-Events mit dem Wert **2** an, dass es sich um einen Doppelklick gehandelt hat. Unabhängig davon wird in diesem Fall auch noch das Event **dblclick** ausgelöst. Wenn Sie gar nicht an den Klicks, sondern eher an den *Mausbewegungen* interessiert sind, sollten Sie sich das Ereignis **mousemove** genauer anschauen. Da es bei jeder noch so kleinen Mausbewegung ausgelöst wird, ist es allerdings besser, keinen umfangreichen Code an dieses Ereignis zu hängen.

#### ■ Event Handler mit **addEventListener()** hinzufügen

Die dritte Möglichkeit, einen Event Handler an ein Ereignis zu hängen besteht darin, die Methode **addEventListener()** des JavaScript-Objekts, das das betreffende HTML-Element repräsentiert, aufzurufen. Genau das machen wir im folgenden Beispiel, in dem wir Eingaben in unserem Eingabefeld mit Hilfe des **keypress**-Events verarbeiten. Der JavaScript-Code unserer Datei **eventtest.js** sieht damit so aus:

```
function tastendruck(e) {
    if(e.key != 'a') inpFeld.value = inpFeld.value + e.key;
    e.preventDefault();
}

var inpFeld = document.getElementById('eingabe');
inpFeld.addEventListener('keypress', tastendruck);
```

Der Event Handler wird in diesem Beispiel mit der Methode **addEventListener()** installiert (Event Listener ist ein Synonyme für Event Handler). Der Aufruf der Methode erfolgt mit dem Namen des Ereignisses sowie dem Event-Handler-Objekt als Argumente.

Unser Event Handler **tastendruck()** bewirkt, dass das eingegebene Zeichen nur dann im Inputfeld dargestellt wird, wenn es kein **a** ist. Die **a** werden also gewissermaßen herausgefiltert (probieren Sie es aus!). Dabei machen wir uns die Eigenschaft **key** des Ereignis-Objekts zunutze, die das eingegebene Zeichen beinhaltet. Nun liegt es aber nun mal in der Natur eines Eingabefeldes, dass die eingegebenen Zeichen auch dargestellt werden. Dieses Standardverhalten von Eingabefeldern wird vom Browser bereitgestellt. Wenn wir aber die eingegebenen Zeichen filtern wollen, müssen wir dieses Standardverhalten irgendwie unterbinden. Genau das geschieht durch Aufruf der Methode **preventDefault()** unseres Ereignis-Objekts. Sie verhindert, dass der Browser das Standardverhalten, das normalerweise mit einem solchen Ereignis verbunden ist, ausführt. Auf diese Weise könnten Sie also beispielsweise auch das Standardverhalten, dass bei einem Klick mit der rechten Maustaste ein Kontextmenü geöffnet wird, unterdrücken.

Analog zu **click** und den beiden „Detail-Ereignissen“ **mousedown** und **mouseup** beim Mausklick existieren auch für den Tastendruck spezielle Ereignisse mit den Namen **keydown** und **keyup**. Anders als **keypress**, das nur bei der Eingabe darstellbarer Zeichen ausgelöst wird, werden **keydown** und **keyup** immer dann getriggert, wenn überhaupt *irgendeine* Taste gedrückt wird. Lassen Sie sich das Ereignis-Objekt von **keydown** oder **keyup** einmal mit **console.log()** anzeigen. Sie werden sehen, dass mit **ctrlKey**, **shiftKey** und **altKey** besondere **boolean**-Eigenschaften verfügbar sind, die anzeigen, ob eine der Sondertasten gedrückt wurde. Wurde eine Sondertaste in Kombination mit einem anderen Zeichen gedrückt, als beispielsweise <CTRL>+<S>, dann enthält **key** das Zeichen; wurde hingegen *nur* die Sondertaste gedrückt enthält **key** einen String wie **Control**, **Shift** oder **Alt**.

Verschaffen Sie sich bei den Ereignissen durch Ausgabe der Ereignis-Objekte in der Konsole einen Überblick über die angebotenen Informationen und ein Gefühl dafür, wann (und wie oft) die Ereignisse ausgelöst werden.

Mit **addEventListener()** könnten Sie übrigens auch mehrere Event Handler für das gleiche Ereignis an das gleichen Objekt hängen. Mit der Methode **removeEventListener()**, die die gleichen Parameter besitzt wie **addEventListener()** können Sie dann einen Event Handler wieder „abklemmen“.

#### ■ Weitere Ereignisse

JavaScript kennt neben den hier angesprochenen noch zahlreiche weitere Ereignisse. Dabei können nicht nur HTML-Elemente Träger von Ereignissen sein. Auch das Dokument (Standardobjekt **document**) und das Browser-Fenster (Standardobjekt **window**) verfügen über Ereignisse; so wird beispielsweise beim Verändern der Größe des Browser-Fensters das **window**-Ereignis **resize** ausgelöst, beim Verlassen der Webseite (Navigation zu einer anderen URL) das **window**-Ereignis **onbeforeunload**. Wenn Sie wissen wollen, welche Ereignisse ein Objekt unterstützt, geben Sie in die JavaScript-Konsole den Objekt-Bezeichner gefolgt von **.on** ein, und Sie gelangen in der Popup-Liste der Objekt-Eigenschaften direkt zu den verfügbaren Events (deren Eigenschaftsnamen ja allesamt mit **on** beginnen).

## 34.4 Lösungen zu den Aufgaben

### 34

#### ■ Aufgabe 34.1

Eine einfache Möglichkeit, zu zeigen, dass Zeichenketten immer als **true** ausgewertet werden, besteht darin, eine Bedingung wie die folgende in der JavaScript-Konsole auszuführen:

```
> if('Eine Zeichenkette') console.log('Ist true')
Ist true
```

Beachten Sie bitte dabei, dass eine Anweisung der Form **if('Eine Zeichenkette' == true)** nicht zu einer Ausgabe geführt hätte. Das liegt daran, dass die Zeichenkette selbst natürlich nicht den Wert **true** hat (sie ist eben eine Zeichenkette und hat den

#### 34.4· Lösungen zu den Aufgaben

in der Zeichenkette abgelegten Wert). Sie wird aber, wenn sie tatsächlich als **true** oder **false** bewertet werden muss, weil an der entsprechenden Stelle eben ein *logischer Ausdruck* erwartet wird, stets als **true** betrachtet.

#### ■ Aufgabe 34.2

Eine Lösung könnte so aussehen:

```
function umrechnen() {
    var temp = Number(
        document.getElementById('temp').value);
    var richtung = document.getElementsByName('richtung');

    if (richtung[0].checked == true) {
        if (temp >= 0) {
            document.write(`<p>${temp} Kelvin in Grad Celsius sind:
                           ${temp - 273.15} Grad Celsius.<p>`);
        }
        else {
            alert(
                'Die angegebene Temperatur in Kelvin muss ' +
                'größer oder gleich Null sein');
        }
    }
    else {
        if (temp >= -273.15) {
            document.write(
                `<p>${temp} Grad Celsius in Kelvin sind: ${temp +
                273.15} Kelvin.<p>`);
        }
        else {
            alert(
                'Die angegebene Temperatur in Celsius muss ' +
                'größer oder gleich -273.15 sein');
        }
    }
}
```

#### ■ Aufgabe 34.3

Eine Lösung könnte so aussehen:

```
if(geschlecht == 'm') anrede = 'Herr'
else anrede = 'Frau';

console.log('Hallo', anrede, name);
```

Weil in den Code-Blöcken des **if**- und des **else**-Zweiges jeweils nur eine Anweisung folgt, kann hier auf die geschweiften Klammern verzichtet werden. Gleches gilt für die vorangegangene Aufgaben, wo allerdings das Weglassen der geschweiften Klammern zu einem etwas unübersichtlicheren Code führt.

## 34.5 Zusammenfassung

---

In diesem Kapitel haben wir uns damit beschäftigt, wie im Programmablauf in Abhängigkeit von Bedingungen und Ereignissen verzweigt werden kann.

Folgende Punkte sollten Sie aus diesem Kapitel unbedingt mitnehmen:

- **if-else**-Konstrukte haben die allgemeine Form **if(bedingung) { codeWenn } else { codeSonst }**, wobei der **else**-Zweig optional ist.
- Die Bedingung ist dabei ein Ausdruck, der sich als **true** oder **false** auswerten lässt; die speziellen Werte **null**, **undefined** und **NaN** werden stets als **false** betrachtet, Zeichenketten immer als **true**; gleiches gilt für Zahlen mit Ausnahme von **0**.
- Eine Bedingung kann aus mehreren Teilbedingungen zusammengesetzt sein, die mit den logischen Operatoren **&&** (logisches UND), **||** (logisches ODER) und **!** (logisches NICHT) miteinander verknüpft werden.
- Neben den üblichen numerischen **Vergleichsoperatoren** **>**, **>=**, **<** und **<=** kommen für den Test auf Gleichheit die Operatoren **==** sowie **!=** (ungleich) zum Einsatz.
- Der Bedingungsoperator? erlaubt effiziente Formulierungen von Verzweigungen in der Form **bedingung ? rueckgabeWenn : rueckgabeSonst** und ist hilfreich, wenn über einen Wert anhand einer Bedingung entschieden werden soll.
- **switch-case**-Konstrukte der Form **switch(ausdruck) { case wert1: anweisungWert1; ...; break; ... case wertN: anweisungWertN; ... ; break; default: anweisungDefault; ... ; break; }** eignen sich gut, wenn ein Ausdruck (zum Beispiel der Wert einer Variablen) auf viele gleichartige Bedingungen getestet werden soll; eine tief verschachtelte und daher unübersichtliche **if-else**-Struktur kann auf diese Weise vermieden werden. Der **default**-Zweig ist optional.
- Event Handler sind Funktionen, die automatisch immer dann aufgerufen werden, wenn ein Ereignis eingetreten ist.
- Event Handler können mit einer Eigenschaftszuweisung der Form **onereignis= "eventHandler()"** direkt im HTML-Code der Webseite für ein HTML-Element festgelegt oder aber im JavaScript-Code dynamisch mit der Methode **addEventListener(ereignisName, ereignisHandlerObjekt)** eines HTML-Element-Objekts installiert werden.
- Die wichtigsten Ereignisse sind Maus- und Tastatur-Ereignisse, von denen es jeweils verschiedene Ausprägungen gibt, die bei bestimmten Konstellationen von Aktionen (wie etwa ein Doppelklick = zwei Klick) oder Teilaktionen (wie etwa dem Loslassen einer Maustaste oder Herunterdrücken einer Tastaturtaste) ausgelöst werden.

### 34.5 Zusammenfassung

- Den Event Handlern wird automatisch ein Ereignis-Objekt übergeben, dessen Eigenschaften das Ereignis genauer beschreiben (zum Beispiel, welche Taste genau auf der Tastatur gedrückt oder an welcher Position auf dem Bildschirm ein Mausklick stattgefunden hat); auch wenn Ihr Event Handler überhaupt kein Argument vorsieht, können Sie in seinem Code doch stets auf das Standardargument `event` zugreifen.



# Wie wiederhole ich Programmanweisungen effizient?

## Inhaltsverzeichnis

- 35.1 Abgezählte Schleifen (for und for..of) – 572**
  - 35.1.1 for-Schleifen mit numerischen Laufvariablen – 572
  - 35.1.2 for-Schleife mit Objekt-Laumvariable (for...of) – 579
- 35.2 Bedingte Schleifen (while und do...while) – 580**
- 35.3 Zusammenfassung – 582**
- 35.4 Lösungen zu den Aufgaben – 582**

**Ergänzende Information** Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann [https://doi.org/10.1007/978-3-658-29850-0\\_35](https://doi.org/10.1007/978-3-658-29850-0_35).

## Übersicht

Wie die meisten anderen Programmiersprachen bietet auch JavaScript Schleifen-Konstrukte, die es erlauben, gleichartige Anweisungen zu wiederholen. In der Praxis werden solche Schleifen gerne und häufig eingesetzt, erlauben sie doch eine übersichtliche und elegante Formulierung von Wiederholungen, deren Durchführungszahl zumindest zu dem Zeitpunkt, da der Code geschrieben wird, noch nicht bekannt ist. Im folgenden werden wir uns die wesentlichen Schleifentypen in JavaScript genauer anschauen und an einem größeren Beispiel in Aktion sehen.

In diesem Kapitel werden Sie lernen:

- wie Sie eine abgezählte **for**-Schleife mit numerischer Laufvariable einsetzen, um Anweisungen oder Blöcke von Anweisungen für eine festgelegte Zahl von Wiederholungen auszuführen
- wie Sie auf einfache Weise eine Menge von Objekten (etwa den Inhalt eines Arrays) mit einer **for-of**-Schleife durchlaufen
- wie Sie kopfgesteuerte (**while**) und fußgesteuerte (**do..while**) bedingte Schleifen entwickeln, deren Durchlaufen von einer frei definierbaren Laufbedingung abhängt.

## 35.1 Abgezählte Schleifen (for und for..of)

### 35.1.1 for-Schleifen mit numerischen Laufvariablen

#### ■ Aufbau und Funktionsweise der for-Schleife

Die normale **for**-Schleife zur Wiederholung eines Code-Blocks kommt in JavaScript in folgender Form daher:

```
for(initialisierung; pruefung; inkrementierung) {
    // Code-Block, der wiederholt wird
}
```

Angenommen, wir haben ein Array mit Namen, das wir mit einer **for**-Schleife durchlaufen wollen:

```
> freunde = ['Peter', 'Sophie', 'Hellen', 'Mike', 'Fatih']
```

Wie bei abgezählten Schleifen üblich, benötigen wir eine (**number**-)Laufvariable, nennen wir sie **i**, die im Rahmen der **for**-Schleife zunächst mit ihrem einem initialisiert wird. Da die Indizierung von Arrays in JavaScript, wie wir mittlerweile wissen, bei **0** beginnt, empfiehlt es sich, die Laufvariable **i** zu Beginn auf diesen Wert einzustellen: **i = 0**. Als nächstes müssen wir im Schleifenkopf eine Prüfbedingung festlegen, die erfüllt sein muss, damit die Schleife weiterläuft. In unserem Beispiel, in dem wir das Array von **0** bis zum letzten Element durchlaufen wollen, muss diese Bedingung lauten: **i <= freunde.length-1** (beachten Sie bitte, dass das letzte Ele-

### 35.1 · Abgezählte Schleifen (for und for..of)

ment den Index **length-1** besitzt, weil wir die Zählung ja bereits bei 0 beginnen!). Schließlich müssen wir der Schleife noch mitteilen, in welchen Schritten die Laufvariable erhöht, also inkrementiert, werden soll, in unserem einfachen Beispiel natürlich um 1. Damit würde eine Schleife, die die Elemente des Arrays durchläuft und in der Konsole ausgibt, so aussehen:

```
for(i = 0; i <= freunde.length-1; i = i+1) {  
    console.log('Freund Nr. ', i+1, ': ', freunde[i])  
}
```

Die Schleife beginnt nun mit einem Wert von **0** für die Laufvariable und prüft zunächst, ob die Laufbedingung erfüllt ist. Weil 0 kleiner ist als die um eins reduzierte Länge des Arrays (nämlich vier), beginnt die Schleife, den Code im Schleifenrumpf auszuführen. Diesen Code durchläuft sie im ersten Durchgang mit **0** als Wert der Laufvariable. Der tatsächlich ausgeführte Code lautet damit:

```
console.log('Freund Nr. ', 1, ': ', freunde[0])
```

Nachdem das Ende des Schleifenrumpfs erreicht ist, springt die Ausführung wieder in den Schleifenkopf und erhöht die Laufvariable entsprechend der Inkrementierungsanweisung, in unserem Beispiel also um eins. Danach wird wiederum die Laufbedingung geprüft, die in unserem Fall auch für den Wert **1** der Laufvariable erfüllt ist. Damit wird ein weiteres Mal der Code im Schleifenrumpf ausgeführt. Diese Runden dreht die Schleife so lange bis die Laufbedingung nach einer neuerlichen Inkrementierung der Laufvariable nicht mehr erfüllt wäre. Dass ist dann der Fall, wenn **i** den Wert **5** annehmen würde. Da dann die Laufvariable größer ist als die um eins verringerte Länge des Arrays, wird der Code-Block im Schleifenrumpf kein weiteres Mal ausgeführt. Stattdessen wird die Ausführung des Programms *hinter* der **for**-Schleife fortgesetzt. Die Laufvariable behält ihren alten Wert und wird kein weiteres Mal inkrementiert.

Statt der Inkrementierungsanweisung **i = i+1** wird gerne unter Verwendung des (unären, weil nur mit einem Operanden arbeitenden) Inkrement-Operators **++** auch einfach **i++** geschrieben. Die Werte der Laufvariablen können natürlich auch dekrementiert werden (so hätten wir das Array auch von hinten nach vorne durchlaufen können). Dann muss natürlich auch der Start-Wert, auf den die Laufvariable initialisiert wird, angepasst werden, da die Schleife sonst überhaupt nicht durchlaufen würde. Auch für die Verringerung der Laufvariable um **1** gibt es einen praktischen Operator, den Dekrement-Operator **--**.

Übrigens: Alle Schleifen können in JavaScript mit der **break**-Anweisung verlassen und mit der **continue**-Anweisung in den nächsten Durchlauf geschickt werden. Das gilt nicht nur für die **for**-Schleife, sondern auch die **for-of**, **while** und **do-while**-Schleifen, die wir uns in den folgenden Abschnitten ansehen werden.

#### ■ Ein praktisches Beispiel

Das folgende Beispiel zeigt eine **for**-Schleife, genauer gesagt sogar zwei ineinander verschachtelte Schleifen, im praktischen Einsatz. Ziel ist es dieses Mal, eine ganz

## Tabellendokument

				0
	3			3
5			9	14
	7			7
5	10	0	9	0

Berechnen

Abb. 35.1 Tabellenblatt mit 5 Zeilen und 4 Spalten

einfache Tabellenkalkulation zu entwickeln, mit der man Zahlen in den Zellen eines Tabellenblatts eingeben und sich dann die Zeilen- und Spaltensummen berechnen lassen kann. Der Benutzer gibt dabei im ersten Schritt die Größe des Tabellenblatts ein.

Das Tabellenblatt, das dann gemäß der Benutzerangaben erzeugt werden soll, sehen Sie in Abb. 35.1.

Für die Eingabe der Blattgröße bieten wir dem Benutzer die folgende einfache Web-Oberfläche:

```
<!DOCTYPE html>
<html>

    <head>
        <title>Tabellenkalkulation</title>
        <noscript>Bitte JavaScript aktivieren!</noscript>
    </head>

    <body>
        <script src="tabellenkalkulation.js"></script>

        <h1>Tabellengröße festlegen</h1>
        <form>
            Zeilen:<br>
            <input id="zeilen" type="text" value="0"><p></p>
            Spalten:<br>
            <input id="spalten" type="text" value="0"><p></p>
            <input type="button" value="Tabelle erzeugen"
                  onclick="tabelle()">
        </form>
    </body>

</html>
```

## 35.1 · Abgezählte Schleifen (for und for..of)

Beim Klick auf den Button „Tabelle erzeugen“ wird die JavaScript-Funktion **tabelle()** aus der Datei **tabellenkalkulation.js** aufgerufen. Diese Funktion sieht folgendermaßen aus:

```
1  function tabelle() {
2      var num_zeilen = Number(
3          document.getElementById("zeilen").value);
4      var num_spalten = Number(
5          document.getElementById("spalten").value);
6      var i, f;
7
8      document.write("<H1>Tabellendokument</H1>");
9      document.write("<form><table>");
10
11     // Zellen schreiben
12     for (i = 1; i <= num_zeilen; i++) {
13         document.write("<tr>");
14         for (f = 1; f <= num_spalten; f++) {
15             document.write("<td><input id='R", i, "C", f,
16                         "type='text' value=''></td>");
17         }
18         // Zelle für Summenspalte hinzufügen
19         document.write("<td><input id='SUM_R", i,
20                         "' type='text' value='' readonly='true' \
21                         style='background-color: #d1d1d1;'></td>");
22         document.write("</tr>");
23     }
24
25     // Summenzeile hinzufügen
26     document.write("<tr>");
27     for (f = 1; f <= num_spalten; f++) {
28         document.write("<td><input id='SUM_C", f,
29                         "' type='text' value='' readonly='true' \
30                         style='background-color: #d1d1d1;'></td>");
31     }
32     document.write("</tr>");
33
34     document.write("</table>");
35
36     document.write(
37         "<input type='hidden' id='nzeilen' value='",
38         num_zeilen, "'>");
39     document.write(
40         "<input type='hidden' id='nspalten' value='",
41         num_spalten, "'>");
42
43     document.write("<p></p>")
44     document.write(
45         "<input type='button' value='Berechnen' \
46             onclick='berechnen()'>");
47     document.write("</form>");
48 }
```

Dieser JavaScript-Code erzeugt die Tabelle für unsere Tabellenkalkulation. Einfache Tabellen haben in HTML folgende Form:<table>

```
<tr><td>Zeile 1, Spalte 1</td><td>Zeile 1, Spalte 2</td></tr>
<tr><td>Zeile 2, Spalte 1</td><td>Zeile 2, Spalte 2</td></tr>
</table>
```

Die einzelnen Tabellenzeilen werden durch **tr**-Elemente (*table row*), die darin enthaltenen Zellen durch **td**-Elemente (*table data*) repräsentiert.

Die äußere **for**-Schleife mit Laufvariable **i**, deren Schleifenkopf in Zeile 12 zu finden ist, erzeugt die *Zeilen* der Tabelle. Im Rumpf dieser Schleife läuft eine weitere **for**-Schleife mit Schleifenkopf in Zeile 14; diese „innere“ Schleife mit Laufvariable **f** schreibt für die aktuelle Zeile, also Zeile **i** (die Laufvariable der „äußeren“ **for**-Schleife), jeweils eine Zelle *für jede Spalte*. Auf diese Weise wird durch die beiden ineinander verschachtelten **for**-Schleifen das rechteckige Tabellschema komplett „abgefahren“.

In den Zeilen 18–22 (dieser Code steht *nicht* in der inneren Schleife!) wird für die jeweils aktuelle Zeile **i** noch eine weitere Zelle als Bestandteil einer Summenspalte geschrieben. Analog wird außerhalb der beiden Schleifen (Zeilen 27–31) mit Hilfe einer weiteren Schleife eine *Summenzeile* geschrieben.

Beachten Sie bitte, dass wir unseren Werte-Zellen IDs der Form **RxCy** mitgeben wobei **x** für die Zeile und **y** für die Spalte steht, in der die jeweilige Zelle lokalisiert ist. Die Summenzeilen bzw. -Spalten verfügen über IDs der Form **SUM\_Rx** bzw. **SUM\_Cy**. Diese Art der systematischen ID-Zusammensetzung wird es uns gleich erlauben, auf einfache Weise auf die einzelnen Zellen zuzugreifen.

Dabei helfen uns auch die Formularelemente in den Zeilen 37–41: sie sind vom Typ **hidden** und sind letztlich nichts anderes als eine versteckte Ablage für Informationen. Wir speichern hier die Zahl der Zeilen und der Spalten, um später beim Summieren darauf zugreifen zu können.

Das Summieren übernimmt die Funktion **berechnen()**, die der Benutzer über den Button auslösen kann, den wir in Zeilen 44–46 anlegen. Alternativ können Sie die Funktion auch als Event Handler an die **change**- oder **input**-Ereignisse der einzelnen Zellen-Eingabefelder hängen; dazu müssen Sie lediglich in den Zeilen 15/16 die die **onchange**- bzw. **oninput**-Eigenschaft des jeweiligen Input-Elements mit einem Verweis auf die Funktionen **berechnen()** versorgen (probieren Sie es aus!).

Der Code der Funktion **berechnen()** sieht folgendermaßen aus:

## 35.1 · Abgezählte Schleifen (for und for..of)

```
1  function berechnen() {
2      var num_zeilen = Number(
3          document.getElementById("nzeilen").value);
4      var num_spalten = Number(
5          document.getElementById("nspalten").value);
6      var i, f, summe, summen_zelle;
7
8      // Zeilensummen ermitteln
9      for (i = 1; i <= num_zeilen; i++) {
10         summen_zelle = document.getElementById("SUM_R" + i);
11         summe = 0;
12         for (f = 1; f <= num_spalten; f++) {
13             summe = summe + Number(document.getElementById(
14                 "R" + i + "C" + f).value);
15         }
16         summen_zelle.value = summe;
17     }
18
19
20     // Spaltensummen ermitteln
21     for (f = 1; f <= num_spalten; f++) {
22         summen_zelle = document.getElementById("SUM_C" + f);
23         summe = 0;
24         for (i = 1; i <= num_zeilen; i++) {
25             summe = summe + Number(document.getElementById(
26                 "R" + i + "C" + f).value);
27         }
28         summen_zelle.value = summe;
29     }
30 }
```

**berechnen()** fragt als erstes die Zeilen- und die Spaltenzahl aus unseren beiden **hidden**-Formularelementen ab. Danach werden dann die Zeilen-Summen (Zeilen 9–17) und Spalten-Summen (Zeilen 21–29) berechnet. Dabei machen wir uns zu nutze, dass die IDs der Werte-Zellen die Form **RxCy** und die Zellen der Summen-Zeilen und -Spalten IDs der Form **SUM\_Rx** bzw. **SUM\_Cy** besitzen.

Im Fall der Zeilensummen zum Beispiel gehen wir alle Zeilen mit Hilfe einer **for**-Schleife durch (Zeile 9) und selektieren zunächst die jeweilige Zelle der Summenspalte (Zeile 10). Im Anschluss müssen wir nur noch die einzelnen Wertespalten durchgehen (Zeile 12), die enthaltenen Zahlen addieren (Zeilen 12–15) und die Summe in die entsprechende Summenzelle dieser Tabellen-Zeile schreiben (Zeile 16).

### 35.1 [10 min]

Wie sehen zwei **for**-Schleifen aus, die unser Array **freunde** in der Konsole ausgeben, wobei

- eine Schleife nur jeden zweiten Eintrag anzeigt
- die andere Schleife zwar jeden Eintrag anzeigt, dabei aber von hinten nach vorne vorgeht?

### ? ! 35.2 [60 min]

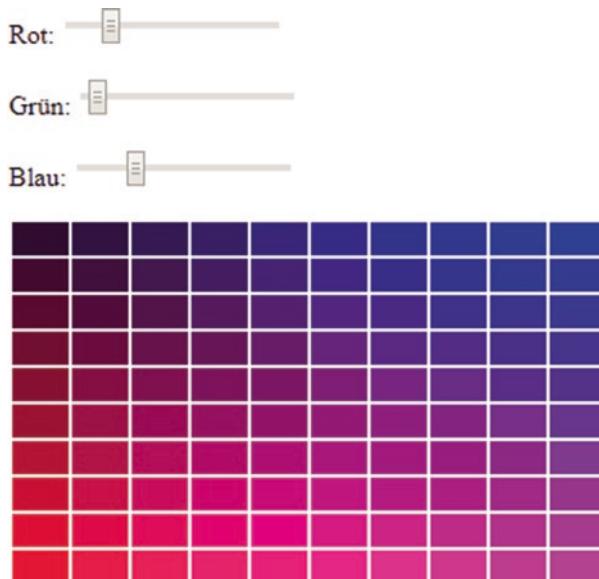
Entwickeln Sie eine Anwendung, bei der der Benutzer zunächst über Schieberegler eine Farbe anhand ihrer Rot-, Grün und Blau-Anteile vorgeben kann. Ausgehend von dieser Farbe soll dann eine Schattierungstabelle aufgespannt werden und zwar so, dass jede der Tabellen-Zellen eine andere Farbe ausweist. Dabei sollen in der Horizontalen die Rot- und in der Vertikalen die Blau-Anteile in insgesamt 10 Schritten bis auf 255 (das Maximum nach dem RGB-Schema) erhöht wird. Der Grün-Anteil bleibt fest bei dem vom Benutzer eingestellten Wert stehen.

Die Schattierungstabelle soll sich automatisch aktualisieren, wenn der Benutzer einen der Schieberegler bewegt.

Ihre fertige Anwendung könnte dann so aussehen wie in □ Abb. 35.2.

Einige Tipps:

- Benutzen Sie die Methode `toString()` des **Number**-Objekts, um eine Dezimalzahl in eine Hexadezimalzahl umzuwandeln, die Sie für die Darstellung eines RGB-Farbwerts HTML-üblichen Format `#RRGGBB` benötigen. Als Argument nimmt `toString()` die Basis des Zahlensystems, in das konvertiert werden soll, bei hexadezimalen Zahlen also **16**.
- Denken Sie daran, dass beim Format `#RRGGBB` immer zwei Stellen pro Farbanteil vorhanden sein müssen! Zahlen kleiner 16 führen aber zu *einstelligen* Hexadezimalzahlen. Diesen muss dann noch eine 0 vorangestellt werden.
- Die Farbanteile, die Sie berechnen, müssen ganzzahlig sein. Runden Sie sie deshalb sicherheitshalber mit der Funktion **Math.floor(zahl)**. Diese Funktion gibt die nächst kleinere Ganzzahl zu der als Argument übergebenen Zahl zurück.



□ Abb. 35.2 Schattierungstabelle in der Übungsanwendung

### 35.1.2 for-Schleife mit Objekt-Lauffvariable (for...of)

Die zweite Form von **for**-Schleifen, die JavaScript kennt, zählt nicht eine numerische Lauffvariable hoch oder runter, sondern läuft durch eine Menge von Objekten; dabei ist der Inhalt der Lauffvariable dann das Objekt, dem der jeweilige Schleifendurchlauf gilt.

Damit lässt sich das Beispiel aus dem vorangegangenen Abschnitt dann so schreiben:

```
> freunde = ['Peter', 'Sophie', 'Hellen', 'Mike', 'Fatih']
> i = 0;
  for(meinFreund of freunde) {
    i++;
    console.log('Freund Nr.', i, ':', meinFreund)
  }
```

Die Variable **i** müssen wir hier selbst hochzählen. Sie ist nicht die Lauffvariable der Schleife (das ist **meinFreund**), sondern dient uns hier nur, eine fortlaufende Nummer für unsere Konsolen-Ausgabe zu erzeugen.

Damit erhalten Sie folgenden Output:

```
Freund Nr. 1 : Peter
Freund Nr. 2 : Sophie
Freund Nr. 3 : Hellen
Freund Nr. 4 : Mike
Freund Nr. 5 : Fatih
```

Die allgemeine Form der **for-of**-Schleife lautet demnach:

```
for(laufvariable of iterierbaresObjekt) {
  // Code-Block, der wiederholt wird
}
```

Das *iterierbare Objekt*, das wir durchlaufen, ist hier ein Array. Auch andere Objekte können iterierbar sein; so könnte man beispielsweise ein Objekt, das eine Adresse repräsentiert, so gestalten, dass seine Eigenschaften (wie etwa der Straßename, die Hausnummer und die Postleitzahl) iterierbar sind und deshalb mit Hilfe einer **for-of**-Schleife durchlaufen werden könnten. Was dazu notwendig ist, geht aber über das JavaScript-Einstiegslevel und damit unsere Zielsetzung hier deutlich hinaus.

Übrigens sind auch Strings, auf die ja auch in Array-Notation zugegriffen werden kann (► Abschn. 31.4), iterierbare Objekte und können mit **for-of**-Schleifen durchlaufen werden:

```
for(zeichen of 'Hallo Welt!') {
  console.log(zeichen);
}
```

Beachten Sie bitte, dass beim Durchlaufen der **for-of**-Schleife die Laufvariable nicht einfach nur eine *Kopie* des Elements unseres iterierbaren Objekts ist, dem der aktuelle Schleifendurchlauf gilt. Es ist praktisch *das Element selbst*. Änderungen, die Sie in ihrer **for-of**-Schleife an der Laufvariablen vornehmen, wirken sich daher auf das durchlaufene Objekt aus!

## 35.2 Bedingte Schleifen (while und do...while)

Mit der **while** sowie der **do-while**-Schleife verfügt JavaScript über zwei bedingte Schleifen. Während **while** kopfgesteuert ist, die Bedingung also zu Beginn der Schleife geprüft wird und die Schleife deshalb, wenn die Bedingung direkt von Anfang an nicht erfüllt ist, gar nicht erst durchlaufen wird, ist **do..while** eine fußgesteuerte Schleife. Sie läuft auf jeden Fall mindestens einmal; am Ende des ersten Durchlaufs (und natürlich auch jedes weiteren Durchlaufs) wird die Bedingung geprüft und so festgestellt, ob die Schleife ein weiteres Mal laufen soll.

Die beiden Schleifen haben allgemein den folgenden Aufbau:

```
while(bedingung) {
    // Code-Block, der wiederholt wird
}
```

und

```
do {
    // Code-Block, der wiederholt wird
}
while(bedingung)
```

Die Aushabe unseres **freunde**-Arrays aus dem vorangegangenen Abschnitt würden wir mit einer **while-schleife** zum Beispiel so erreichen können:

**35**

```
> freunde = ['Peter', 'Sophie', 'Hellen', 'Mike', 'Fatih']
> i = 0;
while(i <= freunde.length - 1) {
    console.log('Freund Nr. ', i+1, ': ', freunde[i]);
    i = i + 1;
}
```

Eine Formulierung mit **do-while** könnte stattdessen so aussehen:

```
i = 0;
do {
```

```
    console.log('Freund Nr.', i+1, ':', freunde[i]);
    i = i + 1;
}
while(i <= freunde.length - 1)
```

**do-while**-Schleifen sollten nur dann eingesetzt werden, wenn man sicher davon ausgehen kann, dass die Laufbedingung der Schleife beim ersten Durchlauf auf jeden Fall erfüllt sein wird; in unserem Fall hier ist das kein Problem, denn wir wissen ja, dass wir kein leeres Array vor uns haben.

In unserem Beispiel haben wir mit **while**- und **do-while**-Schleifen eine Aufgabe gelöst, für die man normalerweise sicherlich eine **for**-Schleife verwenden würde, denn die Zahl der Durchläufe lässt sich ja mit Hilfe der **length**-Eigenschaft des Arrays gut bestimmen. Die bedingten Schleifen kommen normalerweise dann zum Einsatz, wenn die Ausführung tatsächlich an einer allgemeinen Bedingung hängt und die Zahl der Durchläufe *a priori* nicht unbedingt ermittelbar ist. Unser Beispiel zeigt aber sehr schön einen Grundsatz, der uns bereits an früherer Stelle begegnet ist: Jedes Problem, das mit einer abgezählte Schleife lösbar ist, kann auch mit einer bedingten Schleife gelöst werden, weil man als Bedingung im Zweifel auch den Wert einer numerischen Laufvariablen überprüfen kann, die man manuell hochzählt, genau so, wie wir es in unserem Beispiel getan haben. In diesem Sinne ist die *abgezählte* Schleife eine spezielle Form der *bedingten* Schleife, nämlich eben eine, deren Bedingung eine Laufvariable prüft, die zu initialisieren und hochzuzählen die abgezählte Schleife freundlicherweise gleich auch noch mit übernimmt.

### ?

### 35.3 [10 min]

Schreiben Sie zwei weitere Versionen der **while**-Schleife aus diesem Abschnitt, die unser Array **freunde** in der Konsole ausgibt, wobei

- einmal die „Laufvariable“ **i** mit **1** initialisiert wird (statt mit **0**)
- mit dem Kleiner-Operator in der Bedingung gearbeitet wird (statt mit dem Kleiner-Gleich-Operator).

### ?

### 35.4 [30 min]

Entwickeln Sie eine Funktion **zaehleZellen()**, die wahlweise die Zeilen oder die Spalten der Tabelle, die unsere Tabellenkalkulationsanwendung aus ► Abschn. 35.1.1 erzeugt, zählt.

Verwenden Sie eine **while**-Schleife, mit der Sie die IDs der Zellen durchgehen. Machen Sie sich dabei den Umstand zu Nutze, dass die Funktion **document.getElementById()** den Wert **null** zurückgibt, wenn ein Element mit der angegebenen ID nicht gefunden werden konnte.

Die Funktion, die Sie hier entwickeln, könnte von unserer Funktion **berechnen()** verwendet werden, um die Zahl der Zeilen und Spalten zu ermitteln, wenn wir sie nicht als Informationen mit Hilfe von **hidden**-Elementen direkt im Seitenquelltext des HTML-Dokuments „zwischengespeichert“ hätten.

### 35.3 Zusammenfassung

---

In diesem Kapitel haben wir gesehen, wie in JavaScript Code mit Hilfe von abgezählten (**for**, **for..of**) und bedingten (**while**, **do...while**) Schleifen wiederholt wird.

Folgende Punkte sollten Sie aus diesem Kapitel unbedingt mitnehmen:

- Abgezählte **for**-Schleifen haben in JavaScript die Form **for(initialisierung; pruefung; inkrementierung) { Anweisungen }**. Dabei wird eine zu Beginn auf einen Wert initialisierte numerische Laufvariable vor jedem Schleifendurchlauf gemäß einer Inkrementierungsanweisung verändert und geprüft, ob der neue Wert eine Prüfbedingung erfüllt. Ist das der Fall, so wird der folgende Block von Anweisungen durchlaufen, ist dies aber nicht der Fall, wird die Programmabföhrung hinter der Schleife fortgesetzt, und die Laufvariable behält ihren Wert von vor der letzten Inkrementierung.
- Die Inkrementierung kann dabei auch negativ sein, also dargestaltet, dass die Laufvariable mit jedem Schleifendurchlauf kleiner wird.
- Mit einer **for..of**-Scheife der Form **for(laufvariable of iterierbaresObjekt) { Anweisungen }** können Objekte durchlaufen werden, deren Elemente iterierbar sind, also von JavaScript in eine wie auch immer geartete Reihenfolge gebracht werden können, wie zum Beispiel Arrays. Die Laufvariable ist in diesem Fall grundsätzlich kein numerischer Wert, sondern dasjenige Element des iterierbaren Objekts, dem der aktuelle Schleifendurchlauf gilt. Änderungen an der Laufvariable wirken sich auf das iterierbare Objekt, dessen Elemente durchlaufen werden, aus.
- Bedingte Schleifen können entweder kopfgesteuert in der Form **while(bedingung) { Anweisungen }** oder fußgesteuert mit **do { Anweisungen } while(bedingung)** dargestellt werden. Schleifen des letzteren Typs werden mindestens einmal durchlaufen, weil die Bedingung erst am Ende geprüft wird.

### 35.4 Lösungen zu den Aufgaben

---

#### ■ Aufgabe 35.1

Eine Schleife, die nur jeden zweiten Eintrag aus dem Array anzeigt, könnte so aussehen:

```
35
for(i = 0;i <= freunde.length-1; i=i+2) {
    console.log('Freund Nr.', i+1, ':', freunde[i]);
}
```

Eine Schleife, die bei der Anzeige der Array-Einträge von hinten nach vorne vorgeht, könnte so aussehen:

```
for(i = freunde.length-1;i >= 0; i--) {  
    console.log('Freund Nr. ', i+1, ':', freunde[i]);  
}
```

## ■ Aufgabe 35.2

Die Oberfläche unserer Schattierungstabellen-Anwendung könnte im HTML-Code so gestaltet werden:

```
<!DOCTYPE html>  
<html>  
  
<head>  
    <title>Farbtabelle</title>  
    <noscript>Bitte JavaScript aktivieren!</noscript>  
</head>  
  
<body>  
    <script src="farbtabelle.js"></script>  
  
    <form>  
        <p>  
            Rot: <input id="rot" type="range" min=0 max=255  
                  value="0" onchange="farbenNeu()">  
        </p>  
        <p>  
            Grün: <input id="gruen" type="range" min=0 max=255  
                  value="0" onchange="farbenNeu()">  
        </p>  
        <p>  
            Blau: <input id="blau" type="range" min=0 max=255  
                  value="0" onchange="farbenNeu()">  
        </p>  
    </form>  
  
    <table id="farbtabelle">  
    </table>  
    </body>  
</html>
```

Dazu die JavaScript-Datei **farbtabelle.js**:

```

function farbenNeu() {
    var farbeRot = Number(
        document.getElementById('rot').value);
    var farbeGruen = Number(
        document.getElementById('gruen').value);
    var farbeBlau = Number(
        document.getElementById('blau').value);
    var tab = document.getElementById('farbtabelle');
    var i, f, farbe, schrittRot, schrittBlau;

    schrittRot = Math.floor((255 - farbeRot) / 10);
    schrittBlau = Math.floor((255 - farbeBlau) / 10);
    tabelleHTML = "";

    for (i = 1; i <= 10; i++) {
        tabelleHTML = tabelleHTML + '<tr>';
        for (f = 1; f <= 10; f++) {
            farbeRotNeu =
                (farbeRot + i * schrittRot).toString(16);
            farbeGruenNeu = farbeGruen.toString(16);
            farbeBlauNeu =
                (farbeBlau + f * schrittBlau).toString(16);

            if (farbeRotNeu.length == 1) farbeRotNeu =
                '0' + farbeRotNeu;
            if (farbeGruenNeu.length == 1) farbeGruenNeu =
                '0' + farbeGruenNeu;
            if (farbeBlauNeu.length == 1) farbeBlauNeu =
                '0' + farbeBlauNeu;

            farbe = "#" + farbeRotNeu + farbeGruenNeu +
                farbeBlauNeu;
            tabelleHTML = tabelleHTML +
                '<td style="background-color: "' + farbe +
                '"; color: "' + farbe + '">xxxx</td>';
        }
        tabelleHTML = tabelleHTML + '</tr>';
    }
    tab.innerHTML = tabelleHTML;
}

```

35

### ■ Aufgabe 35.3

- Eine **while**-Schleife, die das Array durchläuft und bei 1 startet:

```

i = 1;
while(i <= freunde.length) {
    console.log('Freund Nr.', i, ':', freunde[i-1]);
    i = i + 1;
}

```

- b. Eine **while**-Schleife, die das Array durchläuft und mit dem Kleiner-Operator in der Laufbedingung arbeitet:

```
i = 0;
while(i < freunde.length) {
    console.log('Freund Nr.', i+1, ':', freunde[i]);
    i = i + 1;
}
```

#### ■ Aufgabe 35.4

Die Funktion **zaehleZellen()** könnte folgendermaßen aussehen:

```
function zaehleZellen(spaltezeile = 'spalte') {
    var num = 0;
    var id;

    do {
        num = num + 1;
        if (spaltezeile == 'spalte') {
            id = 'R1C' + num;
        }
        else {
            id = 'R' + num + 'C1';
        }
    }
    while (document.getElementById(id) != null)

    return num - 1;
}
```

Wir arbeiten hier mit einem Argument **spaltezeile**, das den Standardwert "spalte" besitzt und angibt, ob die Spalten- oder die Zeilenanzahl bestimmt werden soll. Die Bestimmung der Anzahl läuft dann darüber, dass in einer **do-while**-Schleife (wir gehen hier davon aus, dass mindestens eine Spalte bzw. Zeile existiert, sodass wir die Laufbedingung der Schleife auch am Ende testen können) Zellen-IDs zusammengesetzt werden und mit **getElementById()** versucht wird, diese Zellen zu selektieren. Gelingt das nicht, weil die Zelle nicht existiert, gibt **getElementById()** den Wert **null** zurück und die Schleife endet. Die in der Schleife hochgezählte Zeilen-/Spaltenzahl **num** ist dann der Rückgabewert der Funktion.

# Wie suche und behebe ich Fehler auf strukturierte Art und Weise?

## Inhaltsverzeichnis

- 36.1 Fehlerbehandlung zur Laufzeit – 588
- 36.2 Fehlersuche und -beseitigung während der Entwicklung – 590
- 36.3 Zusammenfassung – 594

## Übersicht

Zum Abschluss unserer Einführung in JavaScript wenden wir uns dem ungeliebten, aber wichtigen Thema der Fehlersuche und -behandlung zu. JavaScript ist insgesamt etwas robuster als andere Programmiersprachen, wenn es um Laufzeitfehler geht. Dort also, wo Ihr Programm in anderen Sprachen mit einer Fehlermeldung abbrechen würde, läuft es bei JavaScript einfach weiter. Das macht die Behandlung von *Laufzeitfehlern* aber nur scheinbar einfacher; besondere, ungewöhnliche Situationen müssen von Ihnen als Programmierer trotzdem vorhergesehen und bearbeitet werden.

Für die Fehlersuche bereits während der Entwicklung bringen die Entwicklertools der modernen Browser einige sehr nützliche Werkzeuge mit, von denen Sie ausgiebig Gebrauch machen sollten.

In diesem Kapitel werden Sie lernen:

- wie JavaScript mit Laufzeitfehlern umgeht, und was das für Sie als Programmierer bedeutet
- wie Sie Ausnahmen auffangen können
- wie Sie mit den Debugging-Tools der Browser Fehler in Ihren Programmen suchen, insbesondere, wie Sie Haltepunkte setzen, Variablen beobachten und Ihre Programme im Einzelschrittmodus ausführen.

### 36.1 Fehlerbehandlung zur Laufzeit

Laufzeitfehler, bei denen das Programm abbricht, sind in JavaScript seltener als in vielen anderen Programmiersprachen. Die Ursache liegt darin, dass JavaScript manche Situationen, die in den anderen Sprachen eine Ausnahme hervorgerufen hätten, anders behandelt und eine weniger dramatische Auflösung sucht. Wenn Sie beispielsweise mit **Math.sqrt(-1)** die Quadratwurzel aus einer negativen Zahl ziehen, erhalten Sie als Rückgabewert einfach **NaN – not a number**. Wenn Sie eine Zahl durch **0** teilen, wird Ihnen **Infinity** (oder **-Infinity**, wenn der Dividend negativ war) zurückgeliefert. Wenn Sie einen String, der keine Zahl darstellt, in eine Zahl umwandeln wollen (etwa mit **Number("abc")**), so resultiert diese Konvertierung abermals in **NaN**. In allen diesen Fällen hätten viele andere Programmiersprachen den Dienst versagt und eine Ausnahme geworfen. Nicht so JavaScript. JavaScript läuft stur weiter und signalisiert lediglich durch die Ergebnisse der durchgeföhrten Operationen, dass irgendetwas nicht ganz so gelaufen ist, wie eigentlich geplant.

Auf den ersten Blick ist diese Art des Umgangs mit Fehlern für Sie als Programmierer eine gute Sache, ist doch so das Risiko geringer, dass Ihr Programm mit irgendeiner merkwürdigen Ausnahme vollkommen aus dem Tritt gerät. Allerdings bedeutet der Umstand, dass keine Ausnahmen geworfen werden, natürlich nicht, dass Ihr Programm tatsächlich das tut, was es soll; denn ob Ihr JavaScript-Code auch nach der Konvertierung einer vermeintlichen Zahleneingabe durch den Benutzer, die tatsächlich gar keine Zahl war und deshalb einen **NaN**-Wert hervorruft, immer noch zu sinnvollen Ergebnissen kommt, ist erst einmal fraglich. Sie müssen also als Entwickler selbst durch geeignete Überprüfungen dafür Sorge tragen, dass Ihr Programm auch wirklich mit allen denkbaren Konstellationen zu-

rechtkommt – auch und insbesondere dann, wenn diese Konstellationen *nicht* zu Ausnahmen führen.

Ausnahmen können aber natürlich auch in JavaScript auftreten. Wenn Sie etwa für eine numerische Variable die Methode `toLowerCase()` aufrufen, die einen String in Kleinbuchstaben umwandelt, werden Sie ebenso eine Ausnahme erhalten, wie wenn Sie in einem Skript (wie etwa unserem Tabellenkalkulationsbeispiel aus dem vorangegangenen Kapitel) auf ein HTML-Element mit einer ID zuzugreifen versuchen, die tatsächlich aber keinem Element der Webseite zugeordnet ist. Fehler dieser Art haben Ihre Ursache aber oft nicht in Umständen, die erst zur Laufzeit eintreten, sondern sind – zumindest bei ausreichendem Testen – bereits während der Entwicklung erkenn- und behebbar. Im Kampf gegen solche Fehler sind insbesondere die Debugging-Features der Entwickertools im Browser wichtige Unterstützer. Mit ihnen beschäftigen wir uns im nächsten Abschnitt.

Auch, wenn Ausnahmen in JavaScript insgesamt von geringerer Bedeutung sind als in vielen anderen Sprachen, so kennt doch auch JavaScript ein *Versuche...Bei Fehler*-Konstrukt. Es besitzt folgende Syntax und hat damit, wenn Sie sich an unsere Überlegungen aus ► Abschn. 16.2 erinnern, einen ganz „klassischen“ Aufbau:

```
try {
    // Code, der "versucht" wird
}
catch(err) {
    // Code zur Fehlerbearbeitung
}
finally {
    // Code, der auf jeden Fall immer ausgeführt wird
}
```

`catch` übernimmt dabei als Argument ein Fehlerobjekt, aus dem Sie eine Reihe von Informationen herauslesen können, insbesondere den Fehlernamen (in der Eigenschaft `name`) und eine Fehlermeldung (in der Eigenschaft `message`). Wenn Sie besonders interessiert, in welcher Zeile Ihres Codes die Ausnahme geworfen wurde, können Sie die Eigenschaft `line` des Objekts `err` (oder wie immer Sie das Argument von `catch()` auch in Ihrem Skript nennen wollen), abfragen.

Mit Hilfe der Anweisung `throw()` können Sie Ausnahmen übrigens auch selbst erzeugen. So könnten Sie zum Beispiel den Test, ob bei der Division zweier Zahlen der Dividend gleich 0 ist, auch mit Hilfe einer Ausnahme realisieren:

```
var a = 10, b = 0;

try {
    if (b == 0) throw new Error(
        'Division durch 0 ist nicht möglich!');
}
catch (fehler) {
    console.log('Es ist ein Fehler aufgetreten: ',
               fehler.message);
}
```

Wenn Sie Ihre eigene Exception nicht auffangen, erhalten Sie dann natürlich auch einen Programmabbruch mit einer angemessen dramatischen Fehlermeldung in der JavaScript-Konsole des Browsers, wie es sich für eine ordentliche Ausnahme eben gehört!

## 36.2 Fehlersuche und -beseitigung während der Entwicklung

Zur Fehlersuche während der Entwicklung stellen die modernen Browser regelmäßig einige Werkzeuge bereit, die wir bereits in ► Abschn. 29.2.1 kennengelernt haben. Dazu gehören insbesondere Haltepunkte, Variablenbeobachtung und schrittweise Ausführung.

Auch wenn wir uns in diesem Abschnitt regelmäßig auf die Situation in Google *Chrome* beziehen und auch alle Abbildungen die Debugging-Tools in *Chrome* zeigen, so finden Sie doch die gleichen bzw. sehr ähnlichen Werkzeuge in praktisch allen anderen modernen Browsern auch. Sogar die Funktionsweise und Bedienung ist oft äußerst ähnlich.

Um die folgenden Überlegungen etwas plastischer zu machen, betrachten Sie das folgende Beispiel einer Webseite, in der der Benutzer lediglich eine Zahl eingeben und auf einen Button klicken kann:

```
<!DOCTYPE html>
<html>

    <head>
        <title>Skript mit Fehlern</title>
        <noscript>Bitte JavaScript aktivieren!</noscript>
    </head>

    <body>
        <script src="mitfehler.js"></script>

        <form>
            <p>Zahl: <input type="text" id="zahl"></p>
            <p><span id="erg"></span>
            <p></p>
            <input type="button" value="Berechnen"
                   onclick="berechnen()">
        </form>
    </body>

</html>
```

### 36.2 · Fehlersuche und -beseitigung während der Entwicklung

Zur Ausgabe eines Berechnungsergebnisses haben wir ein **span**-Element mit der ID **erg** angelegt. In die Webseite eingebunden wir das Skript **mitfehler.js**, in dem auch die beim Klick auf den Button ausgelöste Funktion **berechnen()** enthalten ist. Es sieht folgendermaßen aus:

```
function berechnen() {
    var zahl = Number(document.getElementById('zahl').value);
    zahl = Math.round(zahl,0);
    zahl = Math.sqrt('zahl');

    ergebnis = document.getElementById('erg');
    ergebnis.innerHTML = 'Wurzel: ' + zahl;
}
```

Wie Sie sehen, machen wir hier nichts anderes, als die vom Benutzer eingegebene Zahl einzulesen, sie auf eine ganze Zahl zu runden und daraus die Quadratwurzel zu ziehen. Das Ergebnis geben wir auf unserem **span**-Element aus. So weit, so gut.

Wenn Sie nun die Webseite im Browser öffnen, eine Zahl eingeben und auf den Button klicken, erhalten Sie die Ausgabe **NAN** auf dem **span**-Element **erg** der Webseite angezeigt. Das Ergebnis der Berechnung ist also scheinbar keine Zahl. Um der Ursache des Problems auf den Grund zu gehen, könnten Sie nun natürlich entweder den Programmcode studieren, und vielleicht ist Ihnen der (natürlich vollkommen absichtlich eingebaute) Fehler bereits aufgefallen; oder Sie lassen die Variable **zahl**, mit der wir im Skript durchgängig arbeiten, einfach nach jeder Operation einmal mit **console.log(zahl)** anzeigen, indem Sie entsprechende Ausgabeanweisungen in den Programmcode integrieren. Natürlich werden beide Wege Sie zum Ziel führen, und das selbst dann, wenn Sie keine Debugging-Werkzeuge im Browser zur Verfügung hätten. Das temporäre Einfügen von Ausgabeanweisungen im Programmcode ist wahrscheinlich die beliebteste Debugging-Methode überhaupt, weil sie sowohl einfach als häufig wirkungsvoll ist. Gerade aber, wenn der Code-Abschnitt, in dem Sie den Fehler vermuten, sehr lang ist, werden Sie unter Umständen sehr viele Debugging-Ausgaben benötigen oder eine Ausgabe schrittweise immer weiter durch den Code-Abschnitt verschieben müssen, bis Sie die interessante Stelle finden, an der Ihr Problem tatsächlich liegt. Das aber ist mühsam. Gerade in solchen Fällen lohnt der Einsatz der Debugging-Tools, die der Browser bereitstellt, und genau die werden wir jetzt benutzen.

Wenn Sie in den Entwickertools auf den Reiter „Sources“ klicken und dann Ihre JavaScript-Datei auswählen, wird Ihnen deren Inhalt in der Mitte des Entwickertools-Bereichs angezeigt.

Das sehen Sie in Abb. 36.1. Im rechten Teil des Entwickertools-Bereichs sehen Sie einige auf- und zuklappbare Unterbereiche, unter anderem den Unterbereich „Breakpoints“. Klicken Sie auf die Zeilenzahl vor der Code-Anzeige, wird an dieser Stelle ein *Breakpoint* gesetzt. Wenn Sie nun den JavaScript-Code ausführen, in unserem Beispiel, indem Sie auf den Button auf der Webseite klicken, wird der JavaScript-Code so lange abgearbeitet, bis der Breakpoint erreicht wird. Dann

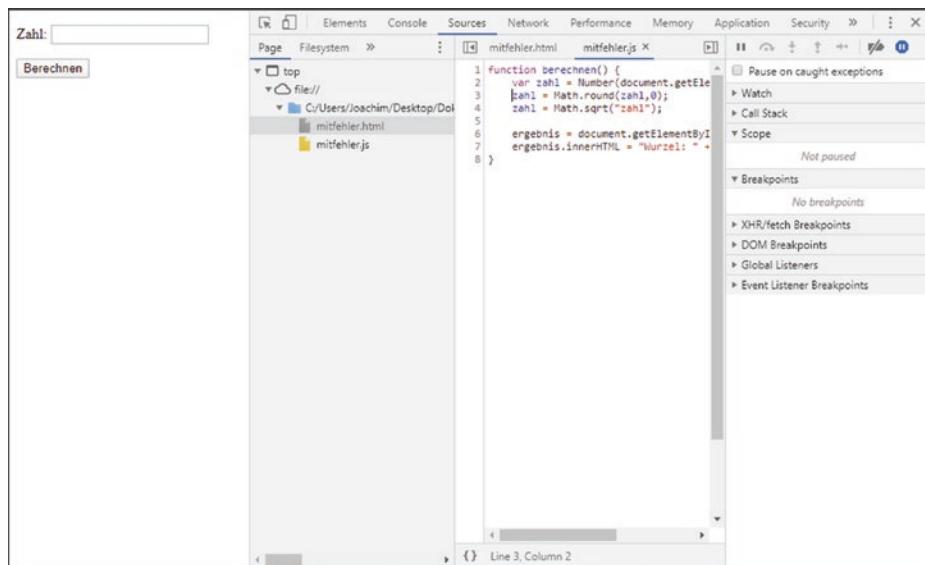


Abb. 36.1 JavaScript-Debugging-Bereich in Google Chrome

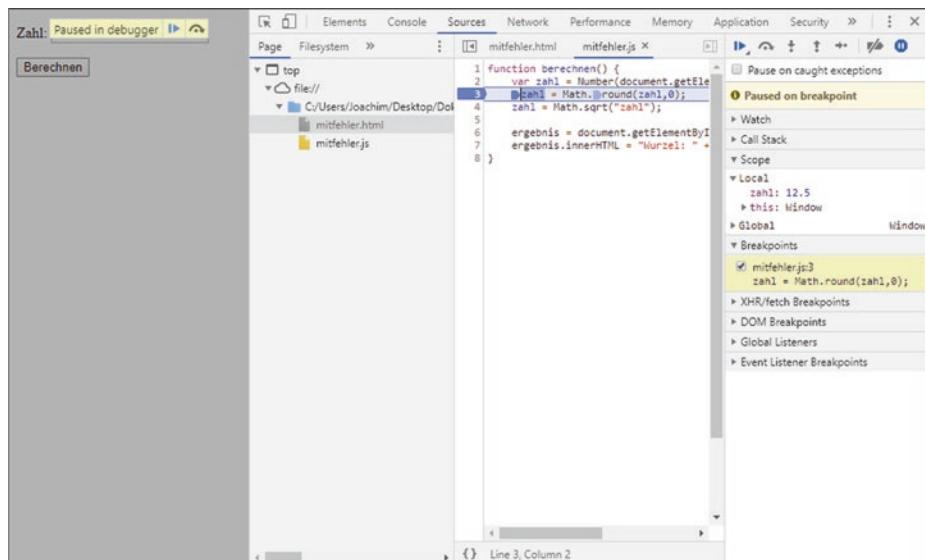
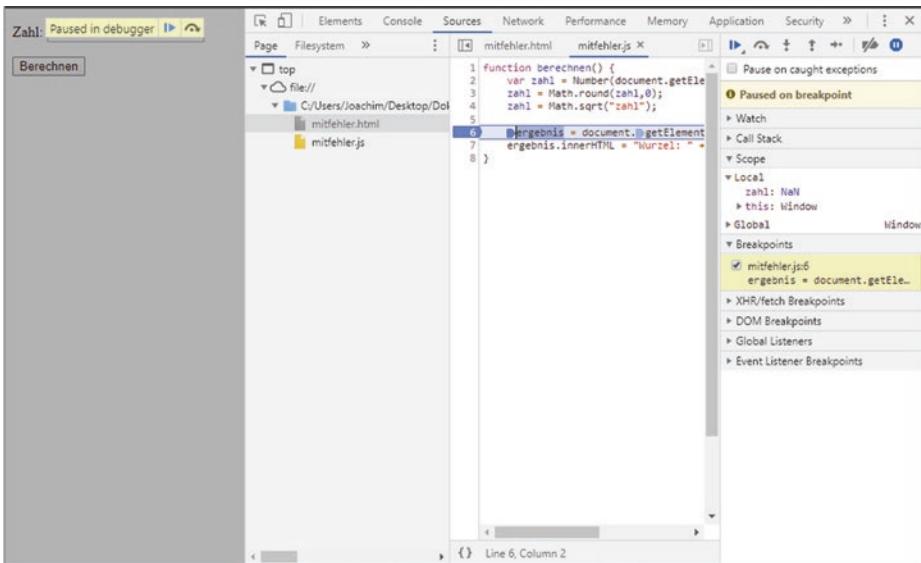


Abb. 36.2 Erreichter Breakpoint in Google Chrome

36 zeigt sich eine Situation wie in Abb. 36.2 dargestellt. Die Ausführung des Programms hat nun angehalten, bevor Zeile 3 ausgeführt wird. Im Unterbereich „Scope“ sehen Sie im aktuellen Gültigkeitsbereich vorhandenen Variablen, insbesondere auch unsere Variable **zahl**. Deren Inhalt würden Sie ebenso sehen, wenn Sie mit dem Mauszeiger auf den Bezeichner der Variable im Code zeigen würden.



■ Abb. 36.3 Erreichen eines Breakpoints, nach dem Fehler (in Zeile 4)

Hier ist also offenbar also noch alles in Ordnung, unsere Variable **zahl** hat den Wert **12.5**, den wir auf der Webseite eingegeben haben. ■ Abb. 36.3 zeigt nun den Zustand des Programms an einem anderen Haltepunkt, nämlich in Zeile 6. Hier sehen wir am „Scope“-Unterbereich rechts, dass **zahl** mittlerweile den Wert **NaN** angenommen hat.

Irgendetwas muss also in den Zeilen 3 und 4 passiert sein. Genaueres Hinsehen offenbart natürlich sofort die Fehlerquelle: In Zeile 4 wird der Funktion **Math.sqrt()** nicht die Variable **zahl**, sondern ein String '**zahl**' als Argument übergeben (haben Sie es vorher gesehen?). Aus einer Zeichenkette kann natürlich keine Quadratwurzel errechnet werden; weil JavaScript aber, wie bereits besprochen, relativ fehlertolerant ist, liefert **Math.sqrt()** einfach **NaN** als Rückgabewert, statt eine Ausnahme zu werfen.

Ein interessantes Debugger-Feature ist die Möglichkeit, sogenannte *Event Listener* hinzuzufügen. Dies geschieht in dem entsprechenden Unterbereich, den Sie in ■ Abb. 36.4 sehen. Hier haben Sie die Möglichkeit, einen Ereignis-Breakpoint zu setzen, also einen Haltepunkt, der immer dann ausgelöst wird, wenn ein bestimmtes Ereignis eintritt, also zum Beispiel ein Mausklick erfolgt. In einem ereignisgesteuerten Programmablauf ist diese Variante des Breakpoints eine nützliche Alternative zu einem Breakpoint, der an einer bestimmten Zeile festmacht, insbesondere dann, wenn Sie viele Event Handler haben, die auf Ereignisse eines bestimmten Typs reagieren.

Breakpoints können also durchaus bei der Fehlerdiagnose gute Dienste leisten. Wenn Sie das Programm von einem Breakpoint aus fortsetzen wollen, klicken Sie in der Symbolleiste über dem rechten Teil des Debugging-Bereich auf die Play-Schaltfläche. Mit den Pfeilen daneben können Sie von einem Haltepunkt aus im *Einzelschrittmodus* weitergehen.

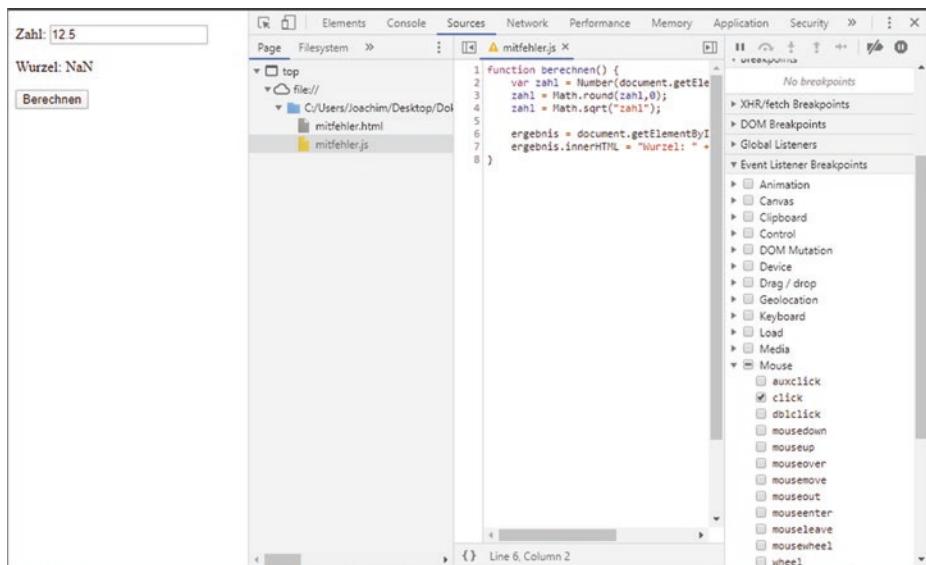


Abb. 36.4 Setzen eines Ereignis-Breakpoints in Google Chrome

Im ersten Unterbereich auf der rechten Seite, dem Unterbereich „Watch“, können Sie *Variablenbeobachtungen* einrichten, also Ausdrücke eingegeben, deren Wert Sie überwachen und sich genauer ansehen wollen, wenn die Programmausführung an einem Breakpoint zum Halt gekommen ist. Die Ausdrücke, die Sie hier eingeben, müssen keineswegs einfach Variablen sein, wie in unserem Beispiel, wo wir lediglich den Wert der Variable `zahl` beobachten, sondern Sie können hier auch komplexere Ausdrücke wie etwa `Math.sqrt(zahl)>2.8` (ein boole'scher Ausdruck) eingeben.

### 36.3 Zusammenfassung

Zum Ende des JavaScript-Teils haben wir uns mit der Fehlerdiagnose und -behandlung beschäftigt.

Folgende Punkte sollten Sie aus diesem Kapitel unbedingt mitnehmen:

- JavaScript ist verhältnismäßig robust gegenüber Laufzeitfehlern, bricht also eher selten mit einer Fehlermeldung ab; das entbindet den Programmierer aber nicht von der Pflicht, sicherzustellen, dass sein Programm auch in außergewöhnlichen Situationen das tut, was es soll.
- Mit dem `try...catch...finally`-Konstrukt steht auch in JavaScript eine Möglichkeit zur Verfügung, Ausnahmen abzufangen; der `catch()`-Anweisung wird, wenn eine Ausnahme auftritt, automatisch ein Fehler-Objekt übergeben, dass Sie auswerten können, um Näheres über die Ausnahme zu erfahren.
- Die Entwicklertools praktisch aller modernen Browser bringen eine ganze Reihe nützliche Debugging-Features mit, um während der Entwicklung Fehler

### 36.3 · Zusammenfassung

zu diagnostizieren; zu diesen Debugging-Features zählen insbesondere Funktionen zur Arbeit mit Haltepunkten (Breakpoints), zur Beobachtung des Inhalts von Variablen und zur Ausführung des Programms im Einzelschrittmodus; daneben besteht mit sogenannten Event Listeners die Möglichkeit, Haltepunkte auch an das Auftreten eines Ereignisses als solchem zu knüpfen.

- Die in der Praxis vermutlich am häufigsten genutzte „Debugging-Werkzeug“ ist die Funktion **console.log()**. Ausgaben, die so erzeugt werden, können in vielen Situationen helfen, Fehlerursachen aufzuspüren; ihnen gegenüber spielen die Debugging-Werkzeuge der Entwicklertools ihre Stärken insbesondere (aber natürlich nicht nur) dann aus, wenn noch wenig Anhaltspunkte vorhanden sind, wo die Fehlerquelle liegen könnte.

# Serviceteil

## Stichwortverzeichnis – 599

# Stichwortverzeichnis

---

## A

- Ablaufsteuerung
  - Bedingung 7, 170, 178
  - Bedingung, komplexe 171, 180
  - Ereignis 126, 169, 186
  - Formen 171
  - Verzweige-Falls-Konstrukt 171, 183
  - Verzweigung 184, 186
  - Wenn-Dann-Sonst-Konstrukt 171, 172, 175, 186
- Ablaufsteuerung (JavaScript)
  - Bedingung 559, 561, 568, 580
  - Ereignis 515, 557, 562
  - Verzweige-Falls-Konstrukt (switch-case) 561
  - Verzweigung 560
  - Wenn-Dann-Konstrukt (if-else) 558
- Ablaufsteuerung (Python)
  - Bedingung 387, 392, 394, 408
  - Bedingungsoperator 560
  - Verzweigung 389, 394
  - Wenn-Dann-Sonst-Konstrukt (if-else) 387
- Algorithmus 6, 8, 26
  - Alltags- 6
  - Grenzen 8
- Analytical Engine 5
- Application Programming Interface (API) 164
- Argument 113, 152, 166
  - optionales 155
  - Positions- 155
  - Standardwert 154
  - Übergabe als Referenz 157, 166
  - Übergabe als Wert 157
- Argument (JavaScript) 542
  - Anzahl 545
  - optionales 465, 477
  - Standardwert 521, 544
  - Übergabe als Referenz 554
  - Übergabe als Wert 543, 555
- Argument (Python) 359, 380
  - Anzahl 362
  - optionales 345, 361, 401
  - Positions- 360
  - Schlüsselwort- 304, 360, 363, 380
  - Standardwert 336, 361, 371
  - Typhinweis 364
- Auskommentieren 251

## B

- Babbage, Charles 5, 22, 30
- Backus, John 32
- Benutzeroberfläche, grafische 73, 127, 134, 193, 210, 304
  - Gestaltung 59, 67
  - Unterschiede zu Konsolenanwendung 139
- Berners-Lee, Tim 442
- Bibliothek
  - geeignete Auswahl 161
  - ins Programm einbinden 162
  - installieren 162
  - Rolle 160
  - zentrale Plattform für 161
- Boole, George 94, 264
- Byron, Augusta Ada 5, 22, 37

## C

- Code-Block 77, 153, 173
- Code-Block (JavaScript) 536
- Code-Block (Python) 246
- Code-Editoren, spezielle
  - Atom 58
  - Notepad++ 58, 439
  - Sublime Text 58, 439
  - Vim 58
  - Visual Studio Code 58, 439
- CPython 228

## D

- Dateien bearbeiten (Python)
  - Anhangemodus 344, 348
  - aus Dateien lesen 344, 345
  - Dateien öffnen 344
  - Dateien schließen 346
  - in Dateien schreiben 344
  - Lesemodus 345, 348
  - Schreibmodus 348
  - Schreib- und Lesemodus 346, 348
- Datenein- und -ausgabe
  - Formen der 124
  - mit Dateien 140
  - mit Datenbanken 145
  - mit grafischen Benutzeroberflächen 125, 127
  - über die Konsole 126, 137, 138

- Datenein- und -ausgabe (JavaScript) 493
  - Datenein- und -ausgabe (Python)
    - Arbeit mit Dateisystem 403
    - mit Dateien 343
    - über die Konsole 302
  - Datentyp
    - Aufzählung/Enumeration 95
    - Boole'sche Variable 94
    - Datum 95
    - einfacher 90, 92
    - Fließkommazahl 92
    - Ganzzahl 91
    - Genauigkeit 91
    - Länge 91
    - Wahrheitswert 94
    - Wertebereich 91, 95
    - Zeichen 92
    - Zeichenkette 92, 96
  - Datentyp (JavaScript)
    - Array 472
    - Genauigkeit 460
    - Länge 475
    - NaN 467, 468
    - null 467
    - undefined 467, 468
    - Wahrheitswert 460
    - Wahrheitswert (boolean) 467
    - Zahl (number) 460
    - Zeichenkette (string) 462
  - Datentyp (Python)
    - Array 275
    - bool 264, 293
    - class 289
    - dictionary 284, 293
    - Fließkommazahl 262, 293
    - float 262, 293
    - Ganzzahl 262, 293
    - geordneter 275, 282
    - Größe 262
    - int 262, 293
    - komplexer 275, 284
    - Länge 280
    - NoneType 265
    - object 267
    - set 287, 293
    - str 262, 293
    - tuple 282, 293
    - ungeordneter 284, 287
    - Wahrheitswert 264, 293, 295
    - Zeichenkette 262, 293
  - Debugging
    - Ausnahme (Exception) 218
    - Einzelschrittmodus 221
    - Fehler zur Entwicklungslaufzeit 216, 217
    - Fehler zur Laufzeit 216, 217
    - Haltepunkt (Breakpoint) 59, 221
    - Methoden 220
    - Syntaxfehler 216
    - Testen 219
    - Variablenbeobachtung (Watch) 221
    - Versuche ... Bei Fehler-Konstrukte 218
    - Werkzeuge 56, 219, 220
  - Debugging (JavaScript)
    - Ausnahme (Exception) 588
    - Fehler zur Laufzeit 588
    - Variablenbeobachtung (Watch) 594
    - Versuche...Bei Fehler-Konstrukte (try... catch) 589
    - Werkzeuge 590
  - Debugging (Python)
    - Ausnahme (Exception) 423, 428
    - Debugging mit PyCharm 424
    - Fehler zur Entwicklungslaufzeit 420
    - Fehler zur Laufzeit 420
    - Haltepunkt (Breakpoint) 425, 428
    - Manuelle Anzeige Variableninhalt 427
    - Schrittweise Ausführung 428
    - Syntaxfehler 250
    - Variablenbeobachtung (Watch) 424, 427
    - Versuche...Bei Fehler-Konstrukte (try... except) 422
    - Werkzeuge 424
    - zur Entwicklungszeit 424
    - zur Laufzeit 420
  - Docstring (Python) 249, 252
  - Dokumentation (Programmcode)
    - Dokumentationsgeneratoren 83
    - Doxygen 83, 254
    - durch Kommentare 47, 80, 251, 444
    - Javadoc 83, 84
    - phpDocumentor 83
    - roxygen 83
  - Dokumentation von Programmcode (Python)
    - autodoc 254
    - Docstring 249, 252
    - doxygen 254
    - Function Annotations 255, 364
    - pydoc 254
    - Rolle und Funktion 245
  - Doppelpunkt-Operator (Python) 247, 276, 284, 289
- E**
- Eich, Brendan 434
  - Entwicklungswerkzeug 56
    - Automatische Syntaxprüfung (Feature) 59
    - Auto vervollständigung (Feature) 59
    - Code-Editor 56, 57, 60, 67

- Code-Editoren, spezielle 58 (*Siehe Auch*  
Code-Editoren, spezielle)
- Community Edition 57, 60
- Compiler 28, 31, 56, 90, 97, 98, 216
- Dark Theme 58
- Features 58
- Grafische Oberflächenentwicklung  
(Feature) 73
- Integrierte Entwicklungsumgebung  
(IDE) 58, 63
- Interpreter 29, 56, 90, 97, 216
- Linker 71
- Entwicklungswerkzeug (JavaScript)
  - Autovervollständigung (Feature) 449
  - Code-Editor 438
  - Integrierte Entwicklungsumgebung  
(IDE) 438
  - Interpreter 438
  - Syntax-Highlighting (Feature) 439
- Entwicklungswerkzeug (Python)
  - Automatische Syntaxprüfung (Feature) 240
  - Autovervollständigung (Feature) 268
  - Dark Theme 231
  - Integrierte Entwicklungsumgebung  
(IDE) 230
  - Interpreter 228, 236, 379
  - Syntax-Highlighting (Feature) 242, 317, 427
- Ereignis (Python) 333, 335, 385, 394
  - Callback-Funktion 334
  - Event Handler 314, 334, 335, 394
  - Maus- 335
  - Tastatur- 335
- Escape-Sequenz 93, 303, 344, 463

## F

- Feld (Variable) 92, 101, 105
  - assoziatives 105, 284
  - geordnetes 101
  - Indizierung 102, 276, 277, 464
  - mehrdimensionales 281
  - Zugriff auf Elemente 101, 295
- for-Schleife (JavaScript) 572
- for-Schleife (Python)
  - Aufbau 400
  - Laufvariable 400
  - Listenkomprehensionsausdruck 406, 411
  - mit nächstem Durchlauf fortsetzen 412
  - verschachtelte 405
  - vorzeitig verlassen 410
- Framework 41, 163
- Framework (JavaScript) 552
- Framework (Python) 224, 305
- Funktion
  - definieren 153

- Kopf 153
- Rückgabewert 152
- Rumpf 153
- Funktion (JavaScript)
  - anonyme 540
  - definieren 536
  - Parameter 542
  - Rückgabewert 541
- Funktion (Python)
  - als Klassenmethode 370
  - def-Anweisung 359
  - Kopf 359
  - Rückgabewert 365
  - Rumpf 359

## G

- Garbage Collection (Python) 261
- Gates, Bill 17
- Geometry Manager (Python, tkinter)
  - Grid 327
  - Pack 327
  - Place 327
  - Rolle und Funktion 327
- GitHub 19, 161, 551
- Gödel, Kurt 37

## H

- Hallo-Welt-Programm
  - (JavaScript) 447
- Hallo-Welt-Programm (Python) 238
- Hamilton, Margaret 10, 22
- Hilfe 65
  - Funktionsreferenz 65
  - Sprachreferenz 65
  - Stack Overflow 41, 65
- Hilfe (JavaScript) 439
- Hilfe (Python)
  - aus dem Internet 231
  - help()-Funktion 232
- Hohmann, Joachim 31
- Hopper, Grace 22, 31
- Houston, Drew 17
- HTML (Hypertext Markup Language  
(HTML)) 498
  - Attribute 443, 504, 507, 512
  - Aufbau 443
  - Cascading Style Sheets (CSS) 444
  - Document Object Model 443, 493,  
500, 503
  - Einbindung von JavaScript 442
    - -Elemente 442, 500, 502, 506, 508
  - Sicherheit 446
  - Tags 498

**I**

- IDLE 229
- if-else Konstrukt (Python) 386
  - mit alternativen Bedingungen 392
  - mit zusammengesetzten Bedingungen 390
  - verschachteltes 389
- Indexierung (JavaScript) 473
- Indexierung (Python) 276, 281
- Integrierte Entwicklungsumgebung (IDE) (JavaScript) 438
- Integrierte Entwicklungsumgebung (IDE) (Python) 230
- Integrierte Entwicklungsumgebung (IDE), spezielle
  - AppCode 63
  - Aptana 63
  - C++ Builder 63
  - Clion 63
  - Eclipse 60
  - IntelliJ IDEA 63, 230
  - JBuilder 63
  - Komodo 63
  - NetBeans 60, 438
  - Padre 63
  - PhpStorm 63
  - PyCharm 63, 228, 230, 236, 242, 377
  - QT Creator 63
  - RAD Studio 59
  - RCommander 63
  - Rodeo 63
  - RStudio 60, 63
  - RubyMine 63
  - SharpDevelop 63
  - Spyder 63
  - Thonny 63
  - Visual Studio 59, 63
  - Visual Studio Tools for Office 63
  - WebStorm 63, 230, 438
  - Xcode 63
  - Zend Studio 63
- Interpreter (JavaScript) 435, 438
- Interpreter (Python) 228
  - in PyCharm 236
  - python (Programm) 228
- Iterierbarkeit (Python) 400

**J**

- Jacquard, Joseph-Marie 6
- Jacquard-Webstuhl 6
- JavaScript
  - Anwendungsgebiete 494
  - Einbindung in HTML-Webseite 442

**Geschichte**

- Standardisierung 434
- JS Bin (JavaScript-Webdienst) 439, 449
- JS Do (JavaScript-Webdienst) 439
- JSON (JavaScript Object Notation) 485

**K**

- Klasse 108
  - Attribut 109, 118
  - Hierarchie 111
  - Instanz 109, 158
  - Konstruktor 115
  - Methode 113
  - Polymorphismus 116, 121
  - Vererbung Methoden und Attributen 110
- Klasse (JavaScript)
  - Attribut 502, 512
  - Grundkonzept 480
  - Konstruktor 483, 484
  - Methode 482
- Klasse (Python)
  - ableiten 290
  - Attribut 267, 290, 292
  - definieren 289
  - Hierarchie 372
  - Instanz 267
  - Konstruktor 272, 274, 290, 295, 372
  - Methode 267
  - name mangling 292
  - Standardmethoden 373
  - Vererbung 290
- Kommentar 47, 76, 80, 251
- Kommentar (JavaScript) 454
- Kommentar (Python)
  - auskommentieren 251
  - FIXME-Kommentar 252
  - Funktion in PyCharm 251
  - mehrzeiliger 251
  - Rolle und Funktion 251
  - TODO-Kommentar 252
- Konsole (JavaScript) 447, 449, 494
- Konsole (Python) 241, 260
- Konstante 101
- Konvertierung (von Variablen) 96, 97
- Künstliche Intelligenz (KI) 9
  - ethische Erwägungen 10
  - Expertensysteme 13
  - gesellschaftliche Diskussion 11
  - im Verkehr 10
  - im Weltraum 10
  - maschinelles Lernen 12
  - neuronales Netz 11
  - schwache 10
  - starke 9

**L**

- Linter 454
- Listenkomprehensionsausdruck (Python) 406
- Liste (Python) 275, 293
  - ändern 277
  - Elemente selektieren 276
  - erzeugen 275
  - Länge ermitteln 280
  - Listen als Elemente von 280
  - löschen 279
  - sortieren 279
- Lochkarte 6
- Logischer Operator 560

**M**

- Makro 17, 182, 200
- Menabrea, Frederico Luigi 5
- Modul (Python)
  - importieren, ausgewählte Klassen 375
  - importieren, gesamtes Modul 376
  - Python Package Index (PyPI) 377
  - Virtuelle Umgebung 237, 379
- Modus, interaktiver 74
- Modus, interaktiver (Python) 229, 241
- Mozilla Foundation 439

**N**

- Nebenwirkung (side effect) 368
- Nerds 20, 42
- NumPy 224, 266

**O**

- Operator, logistischer 181
- Operator (Python)
  - logischer 559
  - Vergleichs- 388
  - Zuweisungs- 388

**P**

- Package (Python) 266, 305, 374, 377
- Plunker (JavaScript-Webdienst) 439, 450
- Programmieren
  - Arbeitsmarkt 18
  - Erlernen 67, 74, 85, 120, 149, 166, 190, 208, 222
  - erstes Programm 46
  - Frauen und Männer 18, 22
  - Geld verdienen 18
  - Hallo-Welt-Programm 72, 305
  - Hype 21
  - Klischees und Vorurteile 20
  - Kunst oder Wissenschaft 47
  - Suchtpotential 21
- Programmierer 5, 16, 21
- Programmierkurs 42
- Programmiersprache
  - Anwendungsgebiet 34
  - Apps (Anwendungsgebiet) 40
  - Assembler- 30
  - Ausführen 71
  - Byte-Code-Compiler für 29
  - Compiler für 22, 28, 31, 56, 90, 97, 98, 216
  - Dialekt 34, 36
  - Fahrplan zum Erlernen 67, 85, 120, 149, 166, 190, 208, 222
  - geeignete Auswahl 37
  - General-Purpose- 35, 224
  - Grammatik 26, 53, 57, 216
  - Hilfe zu 231
  - Hochsprache 22, 30
  - Interpreter für 56, 228, 438
  - Just-in-time-Compiler 30
  - Maschinensprache 22, 28, 30
  - Ökosystem 161
  - Open-Source 41, 65
  - Paradigma 36, 42
  - Popularität 224, 434, 486
  - Quelltext 56, 57, 246, 444, 447, 498
  - Special-Purpose- 35
  - Statistik (Anwendungsgebiet) 34
  - Syntax 26, 57, 216
  - Tipps zum Start mit 45
  - Übersetzen 29, 31
  - Unterschiede zu natürlicher Sprache 26
  - Vielfalt 36
  - Web (Anwendungsgebiet) 434
- Programmiersprache, spezielle
  - ActionScript 435
  - Ada 22, 37
  - Algorithmic Language (ALGOL) 31
  - BASIC 37
  - C# 36, 63
  - C/C++ 35, 36, 43, 60, 63, 111
  - Common Object Business Language (COBOL) 22, 32, 34
  - Delphi 37, 57, 73
  - ECMAScript 434, 439
  - Eiffel 37
  - Formula Translation (FORTRAN) 34
  - Gödel 37
  - Java 30, 35, 84, 182
  - Kotlin 60, 107, 230
  - LiveScript 434
  - Objective C 40

- Programmiersprache, spezielle (*fort.*)
  - Object Pascal 37, 101
  - PHP 34, 40, 63, 140, 158
  - Plankalkül 31
  - Prolog 36
  - R 34, 71, 90, 94, 161
  - Structured Query Language (SQL) 34, 148
  - Swift 34, 40, 63
  - Tcl 305
  - TypeScript 435
  - Visual Basic 35, 37
  - Visual Basic for Applications (VBA) 16, 37, 98
  - Wolfram Language 37
- Pseudo-Code 109
- PyCharm
  - Debugging-Funktionen 424
  - Grundfunktionen 230
  - installieren 230
  - interaktiver Modus 241
  - Interpreter festlegen 236
  - neues Projekt anlegen 231
  - Programm ausführen 236, 238
  - Python-Konsole 241
  - Run-Konsole 238
  - ToDo-Bereich 252
- Python Enhancement Proposal 248, 254
- Python Software Foundation 224, 232

## S

- Schleife
  - abgezählte 197, 198, 207
  - Arten 196–198, 204
  - bedingte 204
  - fußgesteuerte 204
  - kopfgesteuerte 204
  - Laufbedingung 204, 209
  - Laufvariable 198, 199
  - Objekt-Lauffvariable 199
  - Verhältnis abgezählte zu bedingte 207
  - verschachtelte 184, 202
  - vorzeitig verlassen 203
- Schleife (JavaScript)
  - abgezählte 572
  - Arten 572, 580
  - bedingte 580
  - fußgesteuerte 580
  - kopfgesteuerte 580
  - Lauffvariable 579
  - Objekt-Lauffvariable 579
- Schleife (Python)
  - abgezählte 400
  - Arten 400, 407
  - bedingte 407

- kopfgesteuerte 408
- Laufbedingung 407
- Lauffvariable 401, 412
- verschachtelte 405
- vorzeitig verlassen 410
- script (HTML-Tag) 442
- Skript-Modus 72, 229
- StackOverflow 440
- Steuerelement
  - Button 59, 128, 129, 189, 307, 318, 338, 519
  - Checkbox 129, 305, 318, 512
  - List View 131
  - Menü 128, 311, 353
  - Picker 132, 525
  - Radiobutton 129, 318, 514
  - Slider 130
  - Toggle Button 129
  - Tree View 131
- Style Guide 80
  - Style Guide (JavaScript) 453
    - Airbnb Style Guide 453
    - Google JavaScript Style Guide 453
    - JavaScript Standard Style Guide 453
  - Style Guide (Python) 247

## T

- Thiel, Peter 8
- this (Schlüsselwort, JavaScript) 538

TIOBE-Index 41, 43

Tk 305

tkinter 304

- Anwendungsfenster 317, 332
- Beispiel-Applikation 337
- Ereignisse 333
- Hauptschleife 394
- Hilfe und Dokumentation 308
- Standardoption 309
- Widget 307

Typisierung (Variable) 99, 460

## U

- Unterstrich am Bezeichneranfang (Python) 250

User-Experience-Designer 125

User-Interface-Designer 125

## V

van Rossum, Guido 224

Variable

- Bezeichner 78, 89
- deklarieren 98, 101, 120

## Stichwortverzeichnis

- globale 156
- Gültigkeitsbereich 156
- initialisieren 98, 101
- lokale 156
- Variable (JavaScript)
  - Bezeichner 452
  - deklarieren 459
  - globale 547
  - Gültigkeitsbereich 546
  - konvertieren 469
  - konvertieren, explizit 470
  - konvertieren, implizit 469
  - lokale 548
- Variable (Python)
  - als Objekte 267
  - Bezeichner 250, 255
  - Datentyp 260, 261
  - erzeugen 259
  - globale 354, 366
  - Grundtyp 261
  - konvertieren 273
  - konvertieren, explizit 274
  - konvertieren, implizit 273
  - löschen 261
  - lokale 366
  - zuweisen 259
- Verzweigung (Ablaufsteuerung) 568.. *Siehe  
Auch Ablaufsteuerung, Verzweigung*
- Button (Widget) 307
- Canvas (Widget) 326
- Checkbutton (Widget) 318
- Datei-Öffnen/Datei-Speichern-Dialoge 325
- Entry (Widget) 315
- Frame (Widget) 332
- Label (Widget) 317
- Listbox (Widget) 321
- Menu (Widget) 311
- Messagebox(Widget) 324
- Nachrichten-/Entscheidungsdialoge 324
- Notebook (Widget) 326
- Progressbar (Widget) 326
- Radiobutton (Widget) 318
- ScrolledText (Widget) 315
- Spinbox (Widget) 326

Wolfram, Stephen 37

World Wide Web Consortium (W3C) 442, 445

## Z

Zeilenumbruch (JavaScript) 452

Zeilenumbruch (Python)

- in der Ausgabe 264, 303

- im Code 264

Zuse, Konrad 28, 31

## W

while-Schleife (JavaScript) 580

while-Schleife (Python) 407

Widget (Python, tkinter)