

Joachim L. Zuckarelli

Learn coding with Python and JavaScript

A practical introduction for beginners



Springer

Learn coding with Python and JavaScript

Joachim L. Zuckarelli

Learn coding with Python and JavaScript

A practical introduction for beginners



Springer

Joachim L. Zuckarelli
München, Germany

ISBN 978-3-658-42911-9 ISBN 978-3-658-42912-6 (eBook)
<https://doi.org/10.1007/978-3-658-42912-6>

This book is a translation of the original German edition „Programmieren lernen mit Python und JavaScript“ by Zuckarelli, Joachim L., published by Springer Fachmedien Wiesbaden GmbH in 2021. The translation was done with the help of artificial intelligence (machine translation by the service DeepL.com). A subsequent human revision was done primarily in terms of content, so that the book will read stylistically differently from a conventional translation. Springer Nature works continuously to further the development of tools for the production of books and on the related technologies to support the authors.

Translation from the German language edition: “Programmieren lernen mit Python und JavaScript” by Joachim L. Zuckarelli, © Springer Fachmedien Wiesbaden GmbH, ein Teil von Springer Nature 2021. Published by Springer Fachmedien Wiesbaden. All Rights Reserved.

© The Editor(s) (if applicable) and The Author(s), under exclusive license to Springer Fachmedien Wiesbaden GmbH, part of Springer Nature 2024

This work is subject to copyright. All rights are solely and exclusively licensed by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors, and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Fachmedien Wiesbaden GmbH, part of Springer Nature.

The registered company address is: Abraham-Lincoln-Str. 46, 65189 Wiesbaden, Germany

Paper in this product is recyclable.

To my wonderful mother

Introduction

The New World

Our world has always been a world of physical things—of cars, houses, furniture, and toasters. Their production formed the foundation of the prosperity of our industrial societies. In the past two to three decades, this physical world has been joined more and more strongly and with increasing speed and intensity by another, non-material world, which by no means renders the world of physical things meaningless, but which redefines the function and interaction of physical things to a large extent. Moreover, this non-material world brings forth entirely new phenomena that exist largely detached from any physical basis and yet play a decisive role in determining our prosperity and standard of living.

This non-material world is the world of *software*.

The use of software has long shaped our lives in large parts, even if we are not fully aware of it. No train runs, no refrigerator cools the beer, no car rolls even a single foot without software working in secret. The world is becoming more digital and networked—statements like these from politicians, representatives of associations, or managers almost seem like platitudes to us today. And yet they are true. What makes the world more digital and networked is software. Its importance is increasing, and with it the importance of those who can understand and develop software. Today, software is what holds the world together at its core.

Compared to the immense importance that computer programs have for our daily lives, many people—including decision-makers in business and politics—know little about what software is and how it works. Every child learns the basics at school about how car engines work and why airplanes fly; but if a school is expected to teach basic programming, still in some countries a debate flares up about whether this is sensible and organizationally feasible given the packed timetable.

But there are many good reasons to take a closer look at software and its development.

Economic reasons are part of it, even if you leave aside the hero stories of successful software start-up entrepreneurs that the media present to us practically every day. Both developing software, and working with people who do so, are becoming increasingly important skills in the working world, even if managers and employees who are not tech gurus themselves will never work in software development.

Social reasons are important because the increase in automation that goes hand in hand with the growing importance of software has implications for the world of work and our role in it—much greater and more fundamental than many politicians are aware of. The social debate about the benefits and possible risks of artificial intelligence is also easier to conduct for those who have a basic knowledge of

how algorithms and software work. Anyone who wants to make informed political decisions must understand how the world is shaped and changed by software.

Last but not least, there are very practical reasons for dealing with software and the development of software: If you know how to program, you can make your life much easier in many places; you can do things faster, with fewer errors and with less effort than others, and you can even do some things that nobody else can do. And—this should also not be concealed here—a lot of fun is involved, too!

Many of us are good consumers of software. This book sets out to introduce you to the other side, the side of those who develop what we consume every day. You might not end up as a professional programmer, but still understand the basics of developing software and be able to write programs yourself and expand your knowledge if you choose to.

The Approach of This Book

If you talk to professional software developers, you will often hear that they know a range of languages in which computer programs are written. Some programmers list an impressively long list of languages with sometimes very strange names. How is that possible? How can they all speak two, three, four of these programming languages? Are they all geniuses?

Not at all. However, programmers take advantage of a simple fact, namely that programming languages are very similar in many aspects, much more similar than natural languages like English or Spanish are to each other. Many basic concepts are found in virtually all programming languages in one form or another. They may be called differently in each language, but in the end they are always just variants of the same idea implemented in different ways. If you understand these basic concepts, you can learn new programming languages quickly because you know exactly what to look for and only need to understand how the new programming language you're learning implements each basic concept. This makes language learning much easier.

Most books that promise an introduction to programming cover one specific language—and *only* that one language. They start right away with that language and teach all the basic concepts using that one language as an example. The book before you now takes a different approach.

In *Part I*, we deal with the question of what programming is, why you should learn it, and what programming languages are all about.

Part II is dedicated exclusively to the basic concepts of programming. It explains them through 9 questions that you can ask yourself every time you learn a new programming language. The basic concepts covered in these 9 questions are the key to learning to program. Once you understand these basic concepts, you can learn any programming language. Even though dealing with the basic concepts, which are similar in all programming languages, may sound rather theoretical and dry at first, it is not: you will see countless examples from a whole range of programming languages, so that you can see for yourself how similar the languages are after all.

Armed with a good understanding of the basic concepts of programming, you will then be introduced to two of the most popular programming languages around—*Python* and *JavaScript*—in *Parts III* and *IV* of the book. Both parts follow the structure of our 9 questions from the basic concepts part. And it is the basic concepts that we will be looking at implementing in Python and JavaScript in these parts of the book. In doing so, we take a very pragmatic approach. We will not elaborate on every subtlety of our programming languages. You are not meant to become a connoisseur of theoretical language definition. We want you to learn how to program practically with Python and JavaScript. The goal is not for you to write the most efficient and elegant programs, but rather that you write *understandable* programs that *work*. That is the core of programming. Don't let anyone tell you that something has to be a certain way. It's right if it works. If your program is also nice to look at, all the better. But here, the first priority is to develop *functioning* code.

If you already have programming experience, you could also jump right into the Python or JavaScript parts. Both are structured in such a way that they can be read almost like their own books, even if you skipped the basic concepts part.

Of course, you don't necessarily have to learn both programming languages one after the other. Feel free to look at one first, work with it a bit, and then start with the second. Or learn a completely different language! After the basic concepts part, you will be very well prepared for it. It will be easier for you than if you had learned all the basic concepts using only one language as an example.

With numerous exercises, the two parts on Python and JavaScript offer many opportunities to practically apply your knowledge and thereby deepen it. Each exercise shows you the approximate time you should expect to need. Some tasks are easier, others (the ones with the exclamation mark in front) require the development of larger, coherent programs—but you will find a sample solution for all tasks equally.

Have fun with your first steps into the world of programming!

Contents

About Programming

1	What Is Programming?	3
1.1	The Mysterious Power—or: Consciousness Determines Being	4
1.2	Algorithms	6
1.3	Limits of Classical Algorithms: The Playing Field of Artificial Intelligence	8
1.3.1	Only Apparently Intelligent	8
1.3.2	Cat or Not Cat: That Is the Question Here	10
2	Why Learn Programming?	13
2.1	Many Good Reasons	14
2.1.1	Make Your Everyday Life Easier!	14
2.1.2	Speak “IT”!	15
2.1.3	Make Money!	16
2.1.4	Understand What Holds the World Together At Its Core!	17
2.1.5	Train Your Logical, Problem-Solving Thinking!	17
2.2	Clichés and Prejudices	18
2.2.1	Programming is Only for Nerds	18
2.2.2	Software and Its Programming Are Just a Fashion, a Hype	19
2.2.3	Programming is Only for Men	19
3	What Is a Programming Language?	21
3.1	Languages for Humans, Languages for Machines	22
3.2	Translation and Execution of Programming Languages	24
3.3	From Machine Language to High-Level Language	26
4	Why Are There So Many Programming Languages?	29
4.1	Different Fields of Application	30
4.2	Different Paradigms	31
4.3	Further Developments and Dialects	32
5	Which Programming Languages Should You Learn?	35
5.1	Subject of Your Project	36
5.2	Customer Preference and Employer Preference	37
5.3	Pedagogical Aspects	38
6	Some Tips	41
6.1	Learn the Basics First!	42
6.2	Play!	42
6.3	Don’t Lose Heart!	42
6.4	Start Small, and Let Your Program Grow Incrementally!	43

6.5	Let Beauty Be Beauty!	43
6.6	Document!	43

The Basic Concepts of Programming

7	Nine Questions	47
8	What Do I Need for Programming?	51
8.1	Tools	52
8.1.1	Compiler and Interpreter	52
8.1.2	Code Editors.....	53
8.1.3	Integrated Development Environments (IDEs)	54
8.1.4	Simple Online Development Environments.....	59
8.2	Help and Information	60
8.3	Generative Artificial Intelligence Like ChatGPT	62
8.4	Your Roadmap to Learning a New Programming Language	63
9	What Do I Have to Do to Get a Program Running?	65
9.1	All Beginnings Are Easy	66
9.2	Hello, World!.....	68
9.3	Your Roadmap to Learning a New Programming Language	69
10	How Do I Make Sure That I (and Others) Still Understand My Program Later?	71
10.1	Understandable Program Code.....	72
10.2	Design of the Program Code and Naming of Program Elements	73
10.3	Comments	76
10.3.1	Explain Your Own Program Code	76
10.3.2	What Else Comments Are Useful For	77
10.3.3	Documentation Outside the Program Code	78
10.4	Your Roadmap to Learning a New Programming Language	80
11	How Do I Store Data to Work With?	81
11.1	Variables as Placeholders for Data	82
11.2	Data Types of Variables	83
11.2.1	Different Types of Data Require Different Types of Variables.....	83
11.2.2	Important Data Types	84
11.2.3	Changing the Data Type: Conversion of Variables	88
11.3	Creating and Initializing Variables	89
11.4	Not So Variable: Constants	92
11.5	Ordered Fields of Variables/Arrays	92
11.6	Associative Fields of Variables/Hashes	96
11.7	Objects	97
11.7.1	The World Is Made of Objects	97
11.7.2	Classes	98

11.7.3	Inheritance.....	100
11.7.4	Methods.....	103
11.7.5	Polymorphism	105
11.7.6	Access Rights	107
11.8	Your Roadmap to Learning a New Programming Language.....	109
11.9	Solutions to the Exercises.....	109
12	How Do I Input and Output Data?	113
12.1	Forms of Data Input and Output.....	114
12.2	Graphical or Non-Graphical: That Is the Question Here.....	115
12.2.1	Graphical User Interfaces	116
12.2.2	Console Applications	124
12.3	Working with Files.....	126
12.4	Working with Databases.....	131
12.5	Your Roadmap to Learning a New Programming Language.....	134
12.6	Solutions to the Exercises.....	134
13	How Do I Work with Program Functions to Edit Data and Trigger Actions?	137
13.1	Functions.....	138
13.2	Libraries	145
13.3	Frameworks.....	148
13.4	Application Programming Interfaces (APIs).....	149
13.5	Your Roadmap to Learning a New Programming Language.....	150
13.6	Solutions to the Exercises.....	151
14	How Do I Control the Program Flow and Make the Program React to User Actions and Other Events?	153
14.1	Why Program Flow Control Is Necessary.....	154
14.2	Forms of Flow Control.....	155
14.3	If-Else Constructs.....	155
14.4	A Closer Look at Conditions.....	161
14.5	Complex Conditions with Logical Operators (AND, OR, NOT)	163
14.6	Checking Similar Conditions Efficiently with Switch-Case Constructs (Switch/Select...Case).....	166
14.7	Events.....	168
14.8	Your Roadmap to Learning a New Programming Language.....	172
14.9	Solutions to the Exercises.....	173
15	Loops: How Do I Repeat Program Instructions Efficiently?.....	177
15.1	Loops and Their Variants.....	178
15.2	Counting Loops.....	179
15.3	Conditional Loops.....	185
15.4	Your Roadmap to Learning a New Programming Language.....	189
15.5	Solutions to the Exercises.....	189

16	How Do I Search For and Fix Errors in a Structured Way?	193
16.1	Errors at Development Time	194
16.2	Errors at Runtime	195
16.3	Testing	196
16.4	Debugging Methods	197
16.5	Your Roadmap to Learning a New Programming Language	199

Python

17	Introduction	203
18	Tools & Help: What Do I Need for Programming?	207
18.1	Installing the Python Interpreter	208
18.2	Installing the <i>PyCharm</i> IDE	210
18.3	Getting Help with Python	211
18.4	Summary	212
19	Program Execution & Hello World: What Do I Have to Do to Get a Program Running?	213
19.1	Developing and Executing Programs in Python	214
19.2	The Python console: Python in interactive mode	221
19.3	Get to Know <i>PyCharm</i>	222
19.4	Some Alternatives to <i>PyCharm</i> for Python Code Execution	223
19.5	Summary	224
20	Syntax, Comments, Code Style & Documentation: How Do I Make Sure That I (And Others) Still Understand My Program Later?	225
20.1	Design of the Program Code and Naming Conventions	226
20.1.1	Indentation and General Code Formatting	226
20.1.2	End of Statement Without Semicolon, Statements Over Several Lines	228
20.1.3	Case Sensitivity and Choice of Identifiers	229
20.2	Comments	230
20.3	Documentation with Docstrings	232
20.4	Summary	234
21	Variables & Objects: How Do I Store Data to Work With?	235
21.1	Creating and Assigning Variables	236
21.2	Deleting Variables	238
21.3	Basic Types of Variables	238
21.3.1	Numbers (int, float)	239
21.3.2	Strings (str)	239
21.3.3	Truth Values (bool)	241
21.3.4	None	242

21.3.5	Other Data Types.....	243
21.4	Variables as Objects	244
21.4.1	Attributes and Methods of Variables.....	244
21.4.2	Creating Variables with the Constructor Method.....	248
21.5	Converting Variables	249
21.6	Complex Data Types	251
21.6.1	Lists.....	251
21.6.2	Tuples	258
21.6.3	Dictionaries	259
21.6.4	Sets	262
21.7	Self-Defined Classes	264
21.7.1	Defining and Using Classes	264
21.7.2	Deriving Classes from Other Classes.....	266
21.7.3	Avoid Ambiguity: Name Mangling	268
21.8	Summary	268
21.9	Solutions to the Exercises	271
22	User Interfaces: How Do I Input and Output Data?	277
22.1	Input and Output in the Console	278
22.2	Graphical User Interfaces with Tkinter	280
22.2.1	Overview	280
22.2.2	Hello Tkinter!.....	281
22.2.3	Graphical Controls (Widgets).....	282
22.2.4	Arrangement of the Controls (Geometry Managers).....	299
22.2.5	Events	305
22.2.6	Example: Calculator Application.....	308
22.3	Working with Files	315
22.4	Exercise: Development of a Simple Text Editor	317
22.5	Summary	318
22.6	Solutions to the Exercises	319
23	Functions & Methods: How Do I Work with Program Functions to Work with Data and Trigger Actions?	327
23.1	Working with Functions	328
23.1.1	Definition of Functions.....	328
23.1.2	Function Arguments.....	329
23.1.3	Return Values.....	334
23.1.4	Local and Global Variables	335
23.2	Using Functions as Class Methods of Objects	339
23.3	Working with Modules and Packages	342
23.3.1	Modularizing Program Code.....	342
23.3.2	Importing Elements from Modules	344
23.3.3	Using the Community: The Python Package Index (PyPI)	345
23.4	Summary	348
23.5	Solutions to the Exercises	349

24	Conditional Statements & Event Handling: How Do I Control the Program Flow and Make the Program React to User Actions and Other Events?	353
24.1	If-Else Constructs	354
24.1.1	Simple If-Else Constructs	354
24.1.2	Nested if-else Constructs	356
24.1.3	If-Else Construct with Composite Conditions	358
24.1.4	If-Else Construct with Alternative Conditions (elif)	359
24.2	Events	361
24.3	Summary	362
24.4	Solutions to the Exercises	362
25	Loops: How Do I Repeat Program Instructions Efficiently?	365
25.1	Counting Loops (for)	366
25.1.1	Simple for-Loops	366
25.1.2	Nested for-Loops	370
25.1.3	List Comprehension Expressions	371
25.2	Conditional Loops (while)	372
25.3	Exiting Loops Prematurely and Restarting Them	375
25.4	Summary	377
25.5	Solutions to the Exercises	377
26	Debugging & Error-Handling: How to Find and Fix Errors in a Structured Way	381
26.1	Error Handling at Runtime	382
26.1.1	Catching Errors Through Targeted Checks	382
26.1.2	Try-Except Constructs	383
26.2	Troubleshooting During Development	386
26.2.1	Breakpoints	386
26.2.2	Display of Variable Content and Use of Watches	388
26.2.3	Step-by-Step Execution	389
26.3	Summary	390
26.4	Solutions to the Exercises	391

JavaScript

27	Introduction	395
28	Tools & Help: What Do I Need for Programming?	399
28.1	Interpreter	400
28.2	Code Editors and Development Environments	400
28.3	Assistance and Documentation	401
28.4	Summary	402

29	Program Execution & Hello World: What Do I Have to Do to Get a Program Running?	403
29.1	Embedding JavaScript Code in Web Pages	404
29.1.1	The Script Element in HTML	404
29.1.2	Safety Aspects	408
29.2	"Hello World" in JavaScript	409
29.2.1	Do-it-Yourself: The (Not so) Laborious Way	409
29.2.2	With a Little Help: Fast Implementation with a Web Service	411
29.3	Summary	412
30	Syntax, Comments, Code Style & Documentation: How Do I Make Sure That I (And Others) Still Understand My Program Later?	413
30.1	Design of the Program Code and Naming Conventions	414
30.2	Comments	416
30.3	Summary	417
31	Variables & Objects: How Do I Store Data to Work With?	419
31.1	Declaration of Variables	420
31.2	Elementary Data Types	421
31.2.1	Numbers (number)	421
31.2.2	Text (string)	423
31.2.3	Logical Values (boolean)	427
31.2.4	Special Types and Values (null, undefined, NaN)	428
31.3	Converting Variables	430
31.3.1	Implicit Conversion	430
31.4	Arrays	433
31.5	Objects	441
31.5.1	Object Orientation in JavaScript	441
31.5.2	Creating Objects Directly	442
31.5.3	Accessing Properties of Objects	443
31.5.4	Creating Objects Using the Object Constructor	444
31.5.5	Creating Objects Using Constructor Functions	444
31.5.6	JSON	445
31.6	Summary	446
31.7	Solutions to the Exercises	448
32	User Interfaces: How Do I Input and Output Data?	451
32.1	Overview of Input and Output in JavaScript	453
32.2	Output Via the Console	454
32.3	Inputs and Outputs Via Dialog Boxes	457
32.4	Output to an HTML Document/Web Page	458
32.4.1	Writing HTML Code into the Web Page	458

32.4.2	The Document Object Model (DOM)	460
32.4.3	Selecting DOM Nodes by Their Properties	461
32.4.4	Selecting DOM Nodes via the Hierarchical Structure of the Document	463
32.4.5	Changing HTML Elements	466
32.4.6	Adding and Deleting HTML Elements	468
32.5	Input with Forms	470
32.5.1	Forms in HTML	470
32.5.2	Accessing Forms from JavaScript	472
32.6	Example: Simple Calculator	476
32.6.1	The Web Interface	477
32.6.2	The CSS Design Instructions	479
32.6.3	The JavaScript Code	482
32.7	Example: Color Picker	484
32.7.1	The Web Interface	484
32.7.2	The JavaScript Code	486
32.8	Summary	488
32.9	Solutions to the Exercises	490
33	Functions & Methods: How Do I Work with Program Functions to Work with Data and Trigger Actions?	493
33.1	Working with Functions	494
33.1.1	Definition of Functions	494
33.1.2	Return Values	499
33.1.3	Arguments and Parameters of Functions	500
33.1.4	Scope of Variables in Functions	504
33.2	Working with Modules/Libraries	506
33.2.1	Developing and Using Your Own Modules	506
33.2.2	Finding and Integrating External Modules/Libraries	508
33.3	Frameworks	509
33.4	Summary	510
33.5	Solutions to the Exercises	511
34	Conditional Statements & Event Handling: How Do I Control the Program Flow and Make the Program React to User Actions and Other Events?	515
34.1	If-Else Constructs	516
34.2	Switch-Case Constructs	519
34.3	Events	520
34.4	Summary	524
34.5	Solutions to the Exercises	525
35	Loops: How Do I Repeat Program Instructions Efficiently?	527
35.1	Counting Loops (for and for-of)	528
35.1.1	for-Loops with Numerical Run Variables	528
35.1.2	for-Loop with Object Run Variable (for...of)	534

35.2	Conditional Loops (while and do-while)	536
35.3	Summary	537
35.4	Solutions to the Exercises	538
36	Debugging & Error-Handling: How Do I Search for and Fix Errors in a Structured Way?	543
36.1	Error Handling at Runtime	544
36.2	Troubleshooting During Development	546
36.3	Summary	550
Index		553

About Programming



What Is Programming?

Contents

- 1.1 The Mysterious Power—or: Consciousness Determines Being – 4**
- 1.2 Algorithms – 6**
- 1.3 Limits of Classical Algorithms: The Playing Field of Artificial Intelligence – 8**
 - 1.3.1 Only Apparently Intelligent – 8
 - 1.3.2 Cat or Not Cat: That Is the Question Here – 10

1

Overview

In this chapter we will deal with what programming is and why we need programming and programmers in the first place. We will see that programming is about developing or implementing step-by-step problem-solving strategies called algorithms. However, not all types of problems that computers can solve are amenable to processing with strict algorithms; in the field of artificial intelligence (AI), other approaches are used, which we will also look at in this chapter.

1.1 The Mysterious Power—or: Consciousness Determines Being

What do your car, your mobile phone and your washing machine have in common? They are, of course, technical devices and are all marvels of engineering. But what else?

They all run on software. It is the software that gives them life, that allows you to use the devices for the purpose for which they were built. Without software, all of these devices would simply be metal and plastic constructs with no function, no value.

We can't see software, but we can see its effects, mediated by the physical devices, the *hardware*, on which it runs. If your car starts to slide in slippery conditions and you can intercept it and steer it back onto a stable course, then you feel the effects of the Electronic Stabilization Program (ESP). The ESP constantly checks whether the car is moving in the direction you have set with the steering wheel, readjusts it if necessary, and in this way helps you to keep the vehicle under control. A powerful helper that you can't see but is there when you need it.

The software controls the physical devices and tells them what to do. This principle applies equally to your car, your washing machine, and your mobile phone. Nevertheless, there are two key differences in terms of the software that runs on these devices:

First of all, the software that controls your washing machine or provides the assistance functions of your car, for example, is very limited in functionality because it is only needed for a specific purpose. You probably won't want to write your business letters or your master's thesis on your washing machine, so there is no software that allows you to do that, and the washing machine lacks the physical input and output systems that would be necessary to do that. Mobile phones, tablets, and computers (laptops, desktops, servers, etc.), on the other hand, run a much wider range of different software, because the function of these devices is precisely to run programs.

Secondly, cars and washing machines once existed without software, functioning purely mechanically and electrically. A computer without software, however, would make no sense, because its central function is to execute programs.

The simple examples of the washing machine and the car show that software is capturing more and more devices that were previously operated without software.

This increasing digitalization of the real world not only makes it possible to make more and more functions available on the devices or to execute previously existing functions more precisely, safely, quickly, or autonomously, but also to network different devices with each other. The Internet of Things is the modern buzzword here. When your refrigerator recognizes that the milk is about to run out and autonomously orders two cartons of milk online, because it knows that you need milk every morning to prepare your cereals, or when your house automatically raises the shutters because the weather forecast announces strong winds, then you get a sense the software working behind the scenes to make different devices work together for our benefit.

Despite all the networking and all the supposed intelligence of everyday devices, when we think of “software” we naturally think of the classic computer and the software permanently installed on it, such as the operating system, the word processor or the web browser. In fact, however, almost everything we encounter on the Internet every day is ultimately software. Even static websites, as we know them from the early days of the World Wide Web and with which you can do nothing but look at them, are ultimately designed with a special, descriptive programming language (more on this below). Of course, there is software behind *dynamic* websites that respond to user input. When you post a message on Facebook or start a search query on Google, software is running, partly on your own computer, partly on the web server that provides the page.

With its increasing breadth of application and its ability to solve complex problems, software has undergone rapid development in recent decades. By contrast, things did not happen quite as rapidly in the early days of software: the first device on which a real program could run was the *Analytical Engine* of the English polymath *Charles Babbage* in the 1830s, a purely mechanical predecessor of the computer. The Italian mathematician *Federico Luigi Menabrea* and the British scholar *Augusta Ada Byron*, among others, wrote the first programs for this system. This activity earned Byron a reputation as the world’s first female programmer and made her a subject of pop culture. What was most remarkable about the software the two were developing at the time was that the hardware to go with it didn’t even exist at the time. The work of the two early programmers took place with paper and ink, for Babbage’s *Analytical Engine* was never built during his lifetime because the *British Association for the Advancement of Science*, which should have funded the project, found the cost too uncertain and the prospects for success too nebulous. In fact, a fully functional *Analytical Engine* was not built until the 1990s, when much more powerful computers had been available for decades.

Punched cards were used to store the programs (and also the data with which they worked). This technique was first used at the beginning of the nineteenth century in the *Jacquard loom*, a weaving machine that could be “programmed” to weave certain patterns with the help of the punched cards.

Here you can also see the parallel to our discussion about washing machines and computers: Babbage’s *Analytical Engine* was a system that was developed exclusively to run software, whereas the loom of the silk weaver *Joseph-Marie Jacquard* was, like the washing machine, a device that can basically fulfill its function without software, but which becomes more versatile and easier to use through programs.

1.2 Algorithms

Software is ubiquitous these days. It controls the devices and tells them what to do. This is quite obvious with physical devices such as the washing machine, which masters a whole range of automatic washing programs, or your car, which maneuvers into a parking space completely independently thanks to the corresponding assistance function. But it's the same with computer software: the software instructs the computer to display a YouTube video on the screen, for example, and play the appropriate sound through the speakers. Software allows you to use the keyboard to enter your Twitter password or click a button in word processing to put some words in bold type.

The software thus issues instructions to the devices. A set of several such instructions is called an algorithm.

An algorithm is nothing more than a working instruction that explains how to achieve a certain goal step by step. We are familiar with algorithms from our everyday lives, even if we do not usually refer to the work rules and processes we encounter there as algorithms.

One example of an everyday algorithm is a recipe to make a good basil pesto:

1. You need 80 g of pine nuts, 4–5 bunches of basil (pesto alla genovese) and/or rocket, a clove of garlic, 50 g of Parmesan cheese, 50 g of pecorino sardo or pecorino romano, 180 mL of olive oil, and salt (preferably coarse).
2. Wash the basil, pat dry and pluck off the leaves.
3. Peel the garlic clove, remove the germ and chop coarsely.
4. Grate parmesan and pecorino (do not use cheese that has already been grated).
5. Place the pine nuts, garlic and basil leaves in a mortar with the salt and crush finely.
6. Mix in the cheese.
7. Slowly add the olive oil, not too much at a time so you can work it into the mixture well.

If you follow these instructions, you will end up with a good pesto made from the ingredients mentioned in the first step.

Another example of algorithms that we encounter in everyday life all the time is the following:

1. Head northwest on W 31st St toward 8th Ave.
2. Go 184 ft.
3. Turn right onto 8th Ave.
4. Go 0.7 mi.
5. Turn right on W 44th St.
6. Go 0.2 mi.

This algorithm describes how to get from Penn Station to Times Square in New York.

As a final example, here's an everyday algorithm that describes getting up and getting ready on a normal workday:

1. Get up.
2. Brush teeth.
3. Shower.
4. Get dressed.
5. Stir instant coffee in cup.
6. Drink from coffee cup until cup is empty.
7. Put on shoes.
8. If it is cold, put on a jacket.
9. Leave apartment.
10. Lock the door.

This algorithm differs from the previous two algorithms in that not all steps are always executed. Some steps (“... put on a jacket”) are only executed if certain conditions are met (“If it is cold ...”). Other steps are repeated: step 6 says to drink from the coffee cup until the cup is empty. Conditions and repetitions (or “loops,” as the programmer would say) are important concepts of programming, which we will deal with in more detail later.

You can see from the example of this simple instruction for getting up that we often use elements such as conditions and loops in our everyday algorithms, which are also used in programming. Basically, everyday algorithms and the algorithms that programmers develop are not that different. Nevertheless, there are some important differences.

But what are the differences between these everyday algorithms and the algorithms that are processed by computers? We have already said that everyday algorithms are not usually called “algorithms”—just look at the face of the chef in a restaurant when you compliment him after a delicious meal and let him know that “he really used an excellent Tartufo algorithm.”

But the differences between everyday algorithms and computer algorithms are even more profound. Computer algorithms run on the device computer; Everyday algorithms run on the “device human.” The “device human” is an intelligent system and automatically fills in gaps that the algorithm may have with what makes sense at that point. If, for example, the algorithm omits a trivial but obvious intermediate step that is vital, then we recognize this incompleteness and simply execute the missing step anyway, even though it is not specified in the algorithm. Computers don’t do that. Computers are machines that do exactly what they are told. They don’t think for themselves, and they don’t have any intelligence that would allow them to independently recognize incomplete instruction sets or errors in the algorithm and fill in or replace them with what makes sense at that point. That is why computer algorithms must be complete, with no errors. They must be incomparably more precise than algorithms designed to run on the “human device.” Everything must be described in detail so that the computer can correctly execute the algorithm.

Although everyday algorithms and computer algorithms are fundamentally different in this respect, they have one thing in common: they are ultimately just a sequence of steps to achieve a certain goal, regardless of what that goal is—whether it’s to make a fabulous pesto or to make a Facebook post.

1

This answers the basic question we ask in this chapter: Programming is nothing more and nothing less than the process of developing algorithms and writing them down so that the computer can execute them.

1.3 Limits of Classical Algorithms: The Playing Field of Artificial Intelligence

1.3.1 Only Apparently Intelligent

Computers can do many things very well that we humans find difficult or where we quickly reach our limits, as the German-born founder of PayPal, Peter Thiel, argues. They perform complicated calculations at breathtaking speed for hours on end without ever beginning to tire. Yet, Thiel continues, computers routinely fail at tasks that even toddlers can do effortlessly, such as the task of discerning whether or not the animal depicted in a YouTube video is a cat. Why is this seemingly simple problem so difficult for computers to master?

Try writing down an algorithm that checks whether a picture shows a cat. You will probably start with the obvious features of a cat, such as the ears or whiskers. But you can't just tell the computer, "See if there are pointed ears in the picture!." How is it supposed to know what a pointy ear is? So you're going to start describing pointy ears. You might approach the description problem with the geometric shape of a triangle. Color is certainly a good distinguishing characteristic as well. However, you will quickly discover that cat ears are not really quite triangular, even more so when the cat is filmed from the side. Also, your computer will very quickly mistake the gable of a gray and brown house with a thatched roof for a cat's ear. The more you delve into describing what features a cat has and how they differ from similar features of other objects that are not cats, you'll realize how incredibly hard it is to develop a real algorithm that can tell whether we're dealing with a cat or not.

Why is it so easy for us humans to recognize a cat? We always recognize the cat, even if we only see it from behind or against the light. The answer is simple: our brain does not work algorithmically. It does not execute a series of commands step by step to systematically solve the problem of "recognizing a cat." It works completely differently.

An entire branch of computer science deals with the abstract reproduction of this functioning of the human brain in the computer. The ability of a computer to reproduce performances that we typically assume to require intelligence is consequently called *artificial intelligence* (AI).

Artificial intelligence has become a buzzword, you hear about it everywhere and in all kinds of contexts. Many people are frightened by the idea that computers could be "intelligent." An extensive range of exciting science fiction films teaches us that we should be careful what we develop in the field of artificial intelligence, after all, our creations could one day take control and make us redundant.

Recent developments provide reason not to dismiss such fears entirely as the pipe dreams of imaginative screenwriters. Until a few years ago, most experts assumed that there could only be *weak artificial intelligence* (as did the author when

he developed an AI strategy for a company a few years ago). Weak AI means that a machine can perform only in some individual areas like a human. However, at the latest since developments such as *ChatGPT* by OpenAI, we have come much closer to a *strong* or *general artificial intelligence*, i.e., a system that can perform (at least) similarly to humans in practically all areas, and perhaps even shows creativity and has self-consciousness (whatever these two concepts actually mean). A strong AI would pass the famous *Turing Test* (named for British mathematician and computer scientist *Alan Turing*, 1912–1954), in which human jurors were not able to decide if they are talking to a real person or an artificial intelligence. Until now, such systems, for which the android *Data* from the popular “Star Trek—The Next Generation” TV series is an impressive example, were indeed pure science fiction.

However, applications of weak artificial intelligence are still predominant at present. Advanced chess or Go computers beat any human player, no matter how brilliant, by a mile. (Semi-)autonomous driving would not be possible without AI-powered image recognition and the recognition of road conditions, road grids and any other current road users. Voice recognition, such as that offered by Apple’s Siri assistance system, is based on artificial intelligence, as is Google’s image search, or the mechanism that suggests YouTube videos that might interest you.

Some of these applications of artificial intelligence are quite controversial in the social debate, even if they do not come with the danger that world domination by highly intelligent machines will become a reality the day after tomorrow and that humans will be downgraded to useless bio-matter.

There are at least five reasons for this:

1. Because artificial intelligence allows us to reproduce human performance, which we typically associate with intelligence, it opens up a wide field of applications that were previously reserved for human decision-making, evaluation and judgment. We do not feel comfortable delegating these decisions to basically unintelligent systems that only simulate the workings of human intelligence. After all, it is precisely the core idea of weak artificial intelligence that human intelligence performances are reproduced in a narrowly limited area, without causing real *thinking in the machine*. It is therefore not surprising that people are skeptical when they are suddenly expected to entrust their lives to an autonomously driven car, to rely on medical diagnoses made by “intelligent” software, or to accept judgments written by a specialized program as opposed to a human judge. However, the situation is put into perspective when one looks at other situations in history where technology suddenly took over the activities of humans. NASA’s Apollo 11 mission in 1969 saw the first use of a control computer, initially much to the displeasure of the astronauts involved, who simply could not imagine putting their lives in the hands of the control program developed by the young mathematician, *Margaret Hamilton* and her team. In fact, the software did not control the spacecraft fully automatically as an autopilot but acted more like an assistance system for the human pilots, who were getting used to working with the new technology. So, habituation occurs when we realize that the systems are serving their purpose and we develop confidence in their *functioning*, even if we don’t really understand how they work.

2. In addition to the necessary trust in their functionality, *ethical considerations* also pose a problem for applications of artificial intelligence. A classic, hypothetical example is the driverless car that, in a critical traffic situation, only has the choice between two situations that are, strictly speaking, morally unacceptable, either of which would result in the death or serious injury of another road user. This case can still be “solved” reasonably well by the car simply choosing the variant that gives the “affected” road user the highest chance of survival. In case of doubt, the autopilot assesses the situation considerably faster and better than the shocked and completely overwhelmed human driver. Nevertheless, there are more than a few who postulate, based on fundamental considerations, that such decisions should always be reserved for humans. The same is sometimes demanded for decisions in a social context, such as court judgments.
3. Artificial intelligence allows applications that can be used to exert *control over other people*, either through surveillance or by influencing our behavior. In the wrong hands, such technologies would be powerful tools for despots, autocrats, and even for democratic politicians in democratic political systems. The social debate about alleged vote-rigging in the 2016 presidential election in the United States and the role of social media in democratic contestation are still comparatively benign examples of such phenomena.
4. The results that artificial intelligence produces depend heavily on the inputs with which it has been trained. This can lead to artificial intelligence delivering incorrect results or presenting facts in a distorted and biased way. The former is a problem that, for example, the creators of *ChatGPT* explicitly point out to their users. The latter has been criticized in particular in connection with (actually or supposedly) racist statements and connotations.
5. Finally, not only since the rise of *ChatGPT* and its brethren it is likely that artificial intelligence will replace humans in some areas, leading to a social problem in that people will no longer be able to perform the work that they have been trained to do. This is considered a significant problem mainly because the first areas where artificial intelligence will replace humans completely will be those that are currently handled by people with rather low skills. These people, in turn, will find it more difficult to find alternative employment than higher-skilled people.

So even if some applications of artificial intelligence are not without controversy, its spread and number of areas of application will continue to grow. The possibilities offered by artificial intelligence are simply too interesting and too tempting.

1.3.2 Cat or Not Cat: That Is the Question Here

Using the example of recognizing a cat in a picture, we have established that artificial intelligence works in a completely different way to conventional algorithms. And indeed, many AI approaches do not work with an algorithmic sequence of steps, but with a form of pattern recognition that is modeled on the way the human brain works. In this process, signals are routed through several layers of artificial “neurons.” Because of the “neurons,” one also speaks of a *neural network*. These

neurons are connected to other neurons like small network nodes. Any neuron connected in the network can pass on a signal impulse to any other neuron they are connected to. Whether this happens or not, and if so, how strongly, depends first of all on whether there is a connection at all between the neurons in question. If there is, the strength of the impulse that one neuron passes on to the next neuron depends on how “thick” the connection between the neurons is and how strong the signal is that the first neuron received from its respective “predecessor.” The final layer of neurons is the output layer. This is where the result of this multi-layer processing is “displayed.”

In our cat example, the information about the image, i.e., the individual pixels with their position in the image and their color values, hits the input layer of neurons. These pass the impulses on to the next layer, according to the thickness of the connections and the strength of the impulse they themselves received. In the last layer there are only two neurons, representing the results “cat” and “not cat.” So, the outcome, the strength of the signal that ends up with the two output neurons, obviously depends on the distribution of the signals in the input layer (that is, the image we are analyzing) and the thickness of the connections between the neurons. But how does the neural network know how thick these connections must be so that in the end it can recognize whether the image we feed into the input layer represents a cat or not? Answer: It doesn’t. We have to tell it. More precisely, we “train” the network with cat images and images that don’t show cats. After all, we know which result is the correct one in the end. With the help of the result, which the untrained (or at least not yet fully trained) network delivers, and a special algorithm, the connection thicknesses between the neurons can be optimally adjusted. After that, it’s on to the next training image. Over time and thousands of images, the neural network becomes better and better at recognizing the cat images.

The remarkable thing about this technique is that in the end we can’t tell why exactly the neural network is able to distinguish the cat images from the non-cat images in the first place. The network consists of a huge set of neurons and connections between those neurons, as well as the thicknesses of those connections (called weights). If you look at these parameters, you can’t tell from the neural network that it is designed to recognize a cat. In fact, we don’t have the slightest idea *why* the network works. The parameters simply emerged from the training sessions, they were determined by a very classical algorithm based on the difference between the result calculated by the network and the desired, that is, correct result, or readjusted step by step with each training image. Thus, the neural network is a black box. Whereas with a conventional algorithm we can understand exactly how it arrived at this or that result, with the neural network we see only a confusing set of parameters that cannot be interpreted in any meaningful way.

For the sake of completeness, it should be mentioned that not all artificial intelligence approaches work with neural networks. For decades, there have been systems that provide the knowledge of an expert in a certain field on the basis of if-then rules and thus of conventional algorithms, and that practically support people in a question-and-answer game, for example in making a complicated medical diagnosis or in understanding the malfunction of an engine. These systems are also called *expert systems* because they have the knowledge of a human expert in this field documented in explicit rules.



Why Learn Programming?

Contents

2.1 Many Good Reasons – 14

- 2.1.1 Make Your Everyday Life Easier! – 14
- 2.1.2 Speak “IT”! – 15
- 2.1.3 Make Money! – 16
- 2.1.4 Understand What Holds the World Together At Its Core! – 17
- 2.1.5 Train Your Logical, Problem-Solving Thinking! – 17

2.2 Clichés and Prejudices – 18

- 2.2.1 Programming is Only for Nerds – 18
- 2.2.2 Software and Its Programming Are Just a Fashion, a Hype – 19
- 2.2.3 Programming is Only for Men – 19

Overview

You've decided to read this book, so you obviously don't need any more convincing to get into programming. Nevertheless, it is worthwhile to take a clear look at why it makes sense to learn programming in this chapter.

Even though (professional) programmers may sometimes seem like the superstars among knowledge workers in our modern world, they are often met with prejudices that are frequently characterized by overdrawn clichés. You encounter some of these prejudices even as an amateur programmer. We will deal with such prejudices and stereotypes in this chapter.

2.1 Many Good Reasons

Some reasons that learning programming is worthwhile are the following:

2.1.1 Make Your Everyday Life Easier!

Being able to program allows you to automate things that you would otherwise have to do in a laborious and error-prone way by hand. This not only increases your productivity, but also makes work more relaxed, because it is of course especially the boring, monotonous, repetitive, in short: annoying tasks that can be automated particularly well by programs.

Microsoft Excel is the Swiss army knife of the office worker today. With a few exceptions, such as law firms whose business model consists mainly of writing elaborate briefs, contracts and other legal documents, Excel (and to a much lesser extent spreadsheets from other vendors, such as Google) is now used virtually everywhere and for a wide variety of purposes, from budget planning and business cases to project plans and to-do lists. Many users are unaware that Excel (like the other Microsoft Office applications) comes with a powerful programming language, *Visual Basic for Applications*, or *VBA* for short. This programming language allows you to automate even complicated processes. Let's assume, for example, that you have several questionnaires of a customer or employee survey in different Excel files and now want to consolidate them into a single workbook. Of course, you can do this manually by opening each file, reading, or copying the values and pasting them back into your consolidated workbook. A faster and safer way (i.e., with less risk of error) is to write a small VBA program that simply performs the task automatically. Now you may object that it takes time to develop and test the VBA program until it works properly. And indeed, developing such a small macro, as VBA programs are also called, often takes longer than you originally estimated. Nevertheless, the challenge of automating the task is far more exciting than the dull, manual copying of data back and forth. The advantage of the VBA program becomes fully apparent when you have to repeat the task later, for example, because new survey results are available.

Try impressing your colleagues, not to mention your boss, with how quickly and effortlessly you complete many tasks that seem like time-consuming horror projects to most. Drew Houston, one of the founders of the cloud service *Dropbox* once said, “Programming is the closest thing we have to a superpower.” The motif that there is something magical about programming can also be found in the statements of many other programmers. But of course, it only seems magical to the uninitiated. The “magicians” themselves know how the “magic” works and that their “powers” are anything but supernatural.

Microsoft founder Bill Gates proved at a young age that knowing how to program can sometimes make life a little more pleasant. He and his friend and later Microsoft co-founder Paul Allen intervened in the timetable software of Gates’ high school in such a way that he suddenly found himself in a class that consisted almost exclusively of girls. The fact that he couldn’t really get along with any of the female classmates, as he himself reports, was at least not due to his programming skills.

2.1.2 Speak “IT”!

It is an observation that has been repeated a thousand times, but is no less true because of this, that IT is becoming increasingly important in companies and other organizations, including the public sector. There is hardly a modern organization whose way of doing business is not affected by the advance of digitalization. Sometimes even the business model itself is changing in the process. The consequence, of course, is that the IT department becomes more important internally, precisely because there are more and more issues and problems that can only be dealt with in collaboration with internal IT or external IT consultants brought on board for this purpose. Even if you do not work in the IT department yourself, but in a classic “specialist department,” in Sales, for example, or in Finance, you will come into contact with projects and plans in which IT makes a significant contribution. Therefore, an understanding of how “IT” basically works, what is technically solvable and what is not, and which steps are roughly necessary to solve a problem, is of immense advantage. This is especially true if you are closely involved in projects that have an IT component. Gone are the days of writing hundreds of pages of complicated business concepts, which were then interpreted by the IT department more or less poorly and translated into a technical solution. On the rise are agile methods such as SCRUM, in which the development cycles are shorter and the (internal) “customers,” i.e., the those who define the business requirements of a technical solution, are much more closely involved in its design, implementation and testing than in the past. In other words, collaboration with IT is becoming more intensive and important, and so is mutual understanding. As a programming expert, you will bring much more of this with you than others, and with this understanding it will be much easier for you to discuss things with your IT colleagues on an equal footing, even if you have no idea whatsoever about the specific technologies they are using. In fact, much more important than mastering a particular technology is recognizing what is solvable (in terms of automation, for example) and what is not, understanding the approach to problem solving, and

2

being able to think ahead about the most important steps. In this way, you become a valued technical partner to your IT colleagues and effectively advance your technical issues!

And by the way, if you aspire to higher orders: The time when management up to and including the executive board, could be ignorant regarding IT topics because of the operational unimportance of IT, are ending. In the future, few people will get away with putting wild strategies on fancy presentation slides if they have no understanding of how to implement them IT-wise.

2.1.3 Make Money!

We have just argued that IT is becoming increasingly important and is already of immense significance today.

This is also reflected in the labor market data. According to the *Occupational Outlook Handbook* of the U.S. Bureau of Labor Statistics, in 2021 there were 1,425,900 people employed as software developers in the United States. With a median annual pay of USD 120,730 employees in this occupational group are significantly better paid than US employees on average (USD 45,760). If you have not yet decided on a career, or if you want to reorient yourself, this is obviously an attractive field of activity. The Bureau of Labor Statistics expects employment here to increase by 26% in the period 2021–2031.

But of course, you don't have to aim for a permanent position to turn your programming skills into money. Freelancers offer their services on various Internet platforms. The popular website *Upwork*, for example, shows at the time of writing 27,333 projects in the categories “Desktop Software Development,” “Mobile Development” and “Web Development,” which are to be awarded to freelancers by the respective clients. Naturally, many software developers from countries with lower wage levels trade on these platforms, and are prepared to accept lower remuneration, even though they are often well trained and skilled workers. To be successful on platforms like Upwork, you need to build a good reputation, because you acquire new clients partly with the help of reviews you have received from previous clients. Of course, this makes things difficult, especially in the beginning. Here, however, one approach may be to initially accept lower fees and to demonstrate your skills in ways other than past client projects, such as private programming projects that you can publish via platforms like *GitHub* to show to your potential clients. Despite all the difficulties that freelancing via such internet platforms may bring with it, especially at the beginning, it is still an excellent opportunity to try yourself out and test your newly acquired skills on the market, practically risk-free and without having to leave your main job.

Finally, you can also break away from contract work for individual customers and develop your own software or web service that you offer to a broader audience. Whether this is still possible (both practically and legally) alongside your day job can only be decided on a case-by-case basis and will surely require some reflection on your risk preference. However, several examples show that it is possible to be

successful with entrepreneurial activity in the software and internet business, even within a limited timeframe. After all, your product doesn't have to be the next Facebook.

2.1.4 Understand What Holds the World Together At Its Core!

So far, we've talked more about the mundane things in life. About how you can make your life easier, advance your professional projects and earn money with programming. If you're not really interested in all that, and instead have the noble goal of learning more about how our complex world works, you can't avoid programming.

If Goethe's Dr. Faust lived in our time, he would not make a pact with the devil. He would learn to program.

Our world is increasingly governed by the supposedly mysterious power of algorithms. Despite this being the case, many people lack an understanding of what algorithms are and how they work, an understanding that hopefully you have already built up after reading the last chapter. According to the 2018 study *What Europe Knows and Thinks About Algorithms* published by the Bertelsmann Foundation, 48% of 10,960 European survey respondents admit they either haven't heard of algorithms (15%) or don't know what they are (33%). Interestingly, the study finds that the opportunities offered by algorithms are seen more by those who state that they have a good knowledge of what algorithms are.

So, we live in a world increasingly shaped by technologies and problem-solving approaches that many don't understand. It used to be different. School gave everyone a more or less solid knowledge of how the major technologies of the day worked. Anyone who paid attention in school knew the basics of what powered a car, where electricity came from, and why industrial production based on the division of labor offered efficiency advantages. By now, however, public education has lost its way. We'll return to this topic below.

If you want to form an informed opinion about the opportunities and risks of algorithms and participate in the social discussions around this topic, you need to understand more about algorithms than many Europeans do, according to the Bertelsmann survey. Is it necessary for you to be able to program yourself? No, certainly not. But anyone who has ever developed a small program themselves will have a far better understanding.

2.1.5 Train Your Logical, Problem-Solving Thinking!

It is often said that one develops a better understanding of grammar by studying Latin at school, the purely communicative utility of which (except perhaps in the Vatican, where you can probably get away with Italian as well) is rather limited. That the study of Latin implies a better understanding of grammar is because if you want to translate Latin texts, you must deal with grammar more than in other languages, such as English. In this way, you learn the basic concepts of grammar,

for example, the different cases and their uses, which can then be applied to other languages. Thus, learning Latin has a benefit even if you are not one of those who must have at least a minimum knowledge of Latin for professional reasons, such as historians or medical doctors.

It is similar with programming. In few other pursuits will you learn how to approach problems systematically, to break them down into simpler sub-problems, to develop an approach to each of the sub-problems, and finally to assemble the solution to your original problem from the solutions to the sub-problems. The ability to approach problems systematically will help you not only in programming itself, but *everywhere* in life. If you learn to program, you will see the world differently and approach problems of whatever kind in a more structured and solution-oriented way. Perhaps this school of logical thinking is the most important reason of all to get involved with programming.

2.2 Clichés and Prejudices

Now that we've discussed some good reasons why it makes sense to get into programming, let's take a moment to clear up some of the clichés and prejudices that many people have about programming.

2.2.1 Programming is Only for Nerds

Some people still have the image of the pimply, glasses-wearing, male teenager, whose staple food is pizza and coke, who hammers his keyboard obsessively and without stopping all night. Yes, those kinds of programmers certainly exist. And it's not hard to see why there is some truth to this exaggerated stereotype. For one thing, there's the "addictive potential." If you are so inclined, you can quickly get completely "sucked into" a programming problem and forget everything around you, because you are completely focused on solving this one problem that just won't go away. In this situation, time can quickly become secondary. You simply want to solve it, because it would be completely unsatisfactory to leave the problem unsolved, although, as experience shows, it would often be better to rest first and to tackle the problem again the following day with a fresh mind.

Besides the addictive potential, there is a second aspect implicit in the above cliché: programming as a substitute for real social interaction. The great thing about programming is that you develop something and then get feedback from the computer: the program runs, or it doesn't run. It's easy to imagine that this is an interesting incentive for people who are introverted and reserved in social situations. Add to this the perfect controllability: if you write the program correctly, the computer will do exactly what you tell it to do, and nothing else. Social interactions are considerably less controllable, always having elements of the unexpected, the surprising, perhaps even the random. These core elements of social interaction can be easily eliminated in programming. You find yourself in a perfectly controllable situation.

The aspects of addiction potential and controllability only play a role for a small portion of all programmers. Most programmers are completely normal people. The commercial employee who automates some work steps in Excel with the help of VBA to simplify his daily work routine is in the middle of life and may hardly differ from you. The same applies to those who are professionally involved in software development. So, you don't have to worry at all about becoming a member of a social fringe group if you get more intensively involved with programming.

2.2.2 Software and Its Programming Are Just a Fashion, a Hype

Some may think that the software industry is just the spawn of a modern hype. In the end, it's not programs that count, but a good, usable physical product, or so the argument goes. But those who think this way underestimate the important role that software now plays in all our lives. Not only have many products emerged that are entirely software (office applications and social networks are examples), but the way physical products are designed, manufactured, distributed, and monetized is changing in profound ways because of software. Think, for example, of the effects that *Amazon* has on the book trade. In some industries, such as automotive, there are strong indications that software is becoming significantly more important as part of the overall product relative to its physical components. Perhaps in the future, what your car can do, because of the software that runs it, will be more interesting than whether it has alloy wheels and sports seats. Especially for those for whom the car is primarily a means of transportation, the physical car may become more and more interchangeable as a platform on which the really interesting features run in the form of software.

Yes, physical products and non-software services will continue to be important, but they will become more dependent on software, and thus be able to offer their users more features and benefits. The way physical products are created and distributed will also change, sometimes radically, just as the importance of pure software and internet products will increase significantly.

2.2.3 Programming is Only for Men

It may well be that men are, on average, more interested in technical subjects than women. But in the case of programming, there is no reason to believe that women cannot be equally gifted programmers, or that it would be less useful for a woman's career development to be proficient in programming.

In fact, women have always played a special role in the history of software programming. In the nineteenth century, the English mathematician *Augusta Ada Byron* was the first person to describe a complete program for the punched-card machine devised by *Charles Babbage*. For this, she is widely considered the world's first female programmer. (Babbage's *Analytical Machine*, however, was never built during Byron's and Babbage's lifetimes due to funding difficulties). She also real-

ized—and this is probably far more significant—that analytic machines like Babbage’s system were fundamentally capable of more than just dealing with numbers, and could instead work with other types of symbols, such as letters. Further, she devised concepts such as the loop (repetitions of program statements), which are now standard in virtually all modern programming languages. As a tribute, the *Ada* programming language bears her name.

In addition to Augusta Ada Byron, other women have made important contributions to the development of programming. One example is the computer pioneer *Grace Hopper*, who as early as the 1940s had the groundbreaking idea of writing programs in a language that was directly understandable to humans and who played a major role in the development of *COBOL* (*Common Business Oriented Language*), one of the first modern high-level languages. She is also considered to be the inventor of the compiler, which translates the program written in a programming language (which is understandable to humans) into the machine language that the computer can execute directly (more on this in the next chapter).

Or the American mathematician *Margaret Hamilton*, who together with her team developed the control software for the Apollo 11 program and in a sense invented the assistance system with which *Neil Armstrong* and *Buzz Aldrin* landed on the moon in 1969 and returned home safely.

Women have made a decisive contribution to the development of programming and made much of what we now accept as normal possible in the first place. Throughout history, programming has always been a women’s world. Why should it be any different today?



What Is a Programming Language?

Contents

- 3.1 Languages for Humans, Languages for Machines – 22
- 3.2 Translation and Execution of Programming Languages – 24
- 3.3 From Machine Language to High-Level Language – 26

Overview

Programming ultimately means giving instructions to the computer. This is done in a special language that both we as programmers and the computer understand: a programming language. Programming languages and natural languages have a lot in common, but they also have some important differences.

In this chapter we deal with programming languages and thus with the crucial question of how we make ourselves understood by the computer, and how the computer manages to understand us and carry out our instructions.

3.1 Languages for Humans, Languages for Machines

So far, we have dealt with *what* programming is and *why* it makes sense to learn it. Only the question of *how* remains to be dealt with. How do we program? How exactly do we tell the computer what the algorithm, i.e., the sequence of steps it is to process, looks like?

Somehow, the algorithm must be written down in a way the computer can understand. Now, as we know, computers work in a binary mode, they ultimately distinguish only two states, zero and one, on and off, voltage present and voltage absent. It is therefore natural to assume that we have to write programs in a binary way in order for the computer to understand them. Fortunately, this is not the case. If you look over the shoulder of a programmer, you will see that he or she writes texts in a language that is basically readable by humans. In the text you will recognize many English terms. A typical line of program text could read something like:

```
if value < 0 then print "Value is less than zero"
```

Even though many of the terms used are familiar from English, the language used by the programmer is an *artificial* language, a *programming language*. There are other artificial languages that are not programming languages, such as the lingua franca *Esperanto*, developed by the Polish ophthalmologist *Ludwik Lejzer Zamenhof*, or *Klingon*, the native language of the alien Klingon race known from *Star Trek*.

Like human languages, programming languages also have a grammar, the *syntax*, which describes which formulations are permissible and which are not. Nevertheless, programming languages differ significantly from human languages.

1. Programming languages are less complex, both in terms of grammar/syntax and in the terms used and their meaning (called *semantics*). The good news for you is that you don't have to learn such a large vocabulary!
2. The grammatical rules must be strictly observed, otherwise you will not be understood. That is the bad news. In normal conversational situations, the human counterpart understands a sentence even if it is not quite grammatically correct. This is important, given the fact that only native speakers and speakers

with a lot of experience can really master a language without errors. But this kind of internal error correction, or error-tolerant comprehension, requires mental performances that we commonly associate with intelligence. Now the problem—as you can guess—is that computers are simply not intelligent. They have no tolerance for error and only accept and understand correctly formed sentences. This is what makes programming difficult and, from time to time, frustrating, especially when the computer doesn't want (or just can't, because it doesn't have free will) to understand you. Because the language must be so precise, its formulations often seem awkward, which makes it completely unusable as an everyday language. This brings us to the next point.

3. Programming languages are not made to be spoken. Two programmers are not going to talk in a programming language when they meet for lunch. But of course, programmers sometimes speak strangely because they talk about parts of a program written in a programming language and quote parts of the program to do so.
4. Programming languages in the strict sense (there are some descriptive languages that could be considered exceptions) are not made to transfer information, but solely to give instructions to the computer.

Thus, although programming languages, as artificial languages, are designed to be considerably simpler than natural languages, both in terms of their grammar and vocabulary, the works of the authors, i.e., programmers, who use these languages are of considerable complexity and scope. Some examples:

- The programs that controlled NASA Apollo 11 lunar module and brought Neil Armstrong and Buzz Aldrin safely to the moon, had a length of about 40 thousand lines; that is 645 normally written pages.
- The Microsoft Windows 3.1 operating system had about 2.5 million lines; a little over 40,000 pages.
- The office application suite, Microsoft Office 2013, draws its functionality from about 44 million lines of program code, which could be printed on about 694,000 pages.
- The social network Facebook is based on about 61 million lines of program code and thus about 984,000 pages.
- According to a Google developer, all Google applications together have about 2 billion (!) lines; printed out, that would be about 32,300,000 pages.

For comparison: Goethe's *Faust* has 12,111 lines, which corresponds to 195 pages.

Comparisons between these different programs are not easy, because:

1. different programming languages need a different number of lines for the same operation,
2. personal programming styles differ,
3. the way the program code is formatted (which influences the readability and thus “maintainability” of the code) may differ.

All these factors have an influence on the program size.

However, you can see already that software programs are often extensive, and sometimes extraordinarily complex, works.

3.2 Translation and Execution of Programming Languages

In the previous chapter, we had noted that most programming languages use English terms. This is mainly because the development of programming languages has been mainly driven in the English-speaking world, although the first real programming language was the invention of the German *Konrad Zuse*, to whom we will return later. Of course, one could just as easily design a programming language whose commands are all borrowed from a language other than English, such as Spanish, French, or Chinese. However, it is obvious that this is not normally done, except possibly for pedagogical purposes, given the dominant role of English as the world's lingua franca and the understandable desire of every creator of a new programming language to see his work spread as widely as possible.

However, we have not yet solved the problem that the computer can handle neither English nor any other human language. But if the computer only understands zeros and ones, how does it cope with programming languages based on English? What is obviously missing is some step between writing the instructions of the program in a human-readable programming language, and its execution by the computer.

This task is performed by the *compiler*. The compiler translates the program into a language that the computer understands directly, the *machine language*. This machine language knows only a very limited set of basic commands, such as loading individual values into the processor's registers, i.e., the small memory elements whose contents are accessible to the processor to work with.

It's a bit like having to explain a route to someone who doesn't know the place. Whereas a person who knows the area would easily find his way to the meeting point from his current location with the instruction "Go to Time Square, we'll meet there," you have to break down the directions for the person who doesn't know the area to more elementary components. For example, you could say "Go straight ahead to the second traffic light. Then turn left. Then go straight until the next intersection. There, turn right and go straight until you hit the square." It is the same with machine language. It is the stream of instructions, broken down to elementary processor operations, that result from a program written in a programming language. The human-readable program may contain many "shortcuts" and references to already known sequences of instructions, but for the processor, all of this must be reduced to the basic operations that it can perform. This machine code runs directly on the computer's processor. If you work with Windows, you're probably familiar with the *.exe* and *.com* files. These are the programs translated into machine code that the computer understands and can execute directly.

An *interpreter* works somewhat differently. The interpreter (which is itself a program) executes the program step by step; it "interprets" the program code. It is clear that the interpreter ultimately has to pass the program to be executed to the computer in machine language, otherwise the computer would not be able to understand it. But in the case of the interpreter, this translation takes place while

the program is running (“at runtime,” as programmers say). So, the program itself does not run *directly* on the computer, but is mediated by the interpreter. However, this also means: You can’t just run the program everywhere, you always need the interpreter to get the source code of the program running on a computer.

Compiling, or the translation of the human-readable program code into a machine language understandable to the computer, usually has speed advantages because the translation itself takes time and is done by the compiler in advance. At runtime, there is then no need to spend time translating. For our purposes, considerations of code execution speed are not quite so important, but when it comes to computationally intensive or time-critical applications (think, for example, of monitoring systems in intensive care medicine), such considerations are certainly highly relevant.

There is still a middle ground between compiler and interpreter. It works like this: First, a *bytecode compiler* generates an “intermediate code” from the source code of the program. This is very similar to machine language but is independent of the processor architecture. The problem with machine language is that it depends on exactly how the processor is built. Different processor types understand only “their” machine language. Bytecode, on the other hand, is machine-independent and so can easily be generated on one type of system and then executed on other types of systems. The code is said to be “portable”. The generation of the bytecode is done during the development of the program. At runtime, a just-in-time *compiler* (JIT compiler) generates the machine-specific code from the bytecode. To a certain extent, the JIT compiler is a bytecode interpreter. It reads the “precompiled” bytecode and passes instructions to the processor in the respective machine language of the system used for execution. This is faster than executing the untranslated source code directly, as a classical interpreter would do.

The most prominent example of this approach of using a combination of bytecode compiler during development and JIT compiler at runtime, is the popular Java programming language. (Java’s JIT compilers are also referred to as “runtime environments” because they form the environment in which the Java program is executed.) Thanks to its JIT compilers which are adapted to the respective system, Java not only runs on computers with very different operating systems such as Windows, MacOS or Linux, but also on various other devices such as cars or refrigerators.

Typically, it is considered a property of a programming language whether it is compiled or interpreted. However, there are also languages that are basically interpreted, but for which there is then nevertheless the possibility of bytecode compilation. Interpreters, too, may compile parts of the code that are particularly computationally intensive or are often traversed, and sometimes even all of the code, directly into executable machine code; this is also called JIT compilation. So, the world is indeed still a bit more colorful, and compiler construction in computer science is an art (and discipline) in itself.

3.3 From Machine Language to High-Level Language

All well and good with compiling and interpreting. But the question arises: Couldn't you just write a program directly in machine language? Of course, this would be possible, but it would not be easy at all, on the contrary, it would be extremely tedious, and the result would be very difficult to understand. In addition, the program would have to be written in such a way that it takes into account the peculiarities of the processor on which it is to run. If you suddenly want it to run on a completely different system architecture, you might have to modify it considerably in many places. Programming in machine language is therefore not recommended. In the early days of computers, however, that was exactly what was needed. Ultimately, even the decoy card programs of *Charles Babbage* and colleagues are software written in the machine language that Babbage's *Analytical Engine* would have understood directly had it ever been built during his lifetime.

In order to at least simplify the trouble of writing directly in machine language, *assembler languages* appeared from the 1940s onwards. Assembler languages are ultimately machine code in a readable form. For the (manageably) numerous machine commands, short commands (*mnemonics*) were created, which can be translated one-to-one into machine code. In other words: the assembler instructions correspond exactly to the instruction set of the processor. Programming in assembler is still incredibly tedious. For example, an assembler instruction is:

```
MOV AL, 6Fh
```

This tells the computer: load the value 111 (6Fh means 111 in the hexadecimal number system, that is, in the sixteen-based number system) into the processor register AL. To use a simple instruction written in a high-level language, such as:

```
print "Hello World!"
```

in assembler, it requires a whole series of such operations. Above all, however, with assembler one is still dependent on the respective processor architecture.

It is easier with the *high-level languages*. High-level languages abstract from the processor architecture because their interpreters or compilers take over the consideration of the specifics of the target processor, so that the developer is relieved of this task. He writes his program only once; afterwards it can be translated into the machine language of completely different processor architectures.

The high-level languages, unlike assembler code and (even more so) machine code, are easier to read because they use English terms for the instructions. Many of these commands combine a whole series of machine commands, which means that one line of high-level language code may generate many different machine code commands when compiled. The programmer, of course, no longer needs to worry about these details. No wonder, then, that practically all programming languages in use today are such high-level languages.

3.3 From Machine Language to High-Level Language

The first simple high-level language was developed in the 1940s by the German engineer *Konrad Zuse* and was called *Plankalkül*. With *Plankalkül*, Zuse achieved a groundbreaking development that should have paved the way for modern software development in the first place. Due to World War II Zuse's work remained unnoticed for a long time. The first implementation (i.e., usable technical realization) of Zuse's *Plankalkül* took place in 1975 and was done by *Joachim Hohmann* in the context of his dissertation.

The breakthrough of high-level languages, on the other hand, came with *ALGOL* (short for *Algorithmic Language*), which was developed in the USA in the late 1950s/early 1960s mainly for academic purposes and largely without reference to the pioneering work of *Konrad Zuse*. Even before that, in the 1950s, the American computer scientist *Grace Hopper* had designed the first working compiler. *Hopper*, who died in 1992 and was posthumously honored with the Presidential Medal of Freedom, the highest civilian award of the United States, was also involved in the development of the programming language *COBOL* (*Common Business-Oriented Language*). *COBOL* was a language widely used in a business context. Together with *FORTTRAN* (*Formula Translation*), developed by *John Backus* in 1957 and mainly used in technical and scientific work, it completes the collection of early, truly operational high-level languages.



Why Are There So Many Programming Languages?

Contents

- 4.1 Different Fields of Application – 30
- 4.2 Different Paradigms – 31
- 4.3 Further Developments and Dialects – 32

Overview

There are literally hundreds of different programming languages and new ones are being developed all the time. Why is that? Wouldn't it be enough to have one programming language that all programmers understand and all computers can be programmed with?

In this chapter, we explore the question of why the world of programming languages is so colorful, multifaceted, but unfortunately also cluttered.

4

The three early languages ALGOL, COBOL and FORTRAN that we mentioned in the last chapter only form a small part of the universe of the programming languages that exist today. There are contradictory statements about their exact number. This is partly because it is a matter of definition about when two languages should be considered “different.” Should a slight variation of a language (a “dialect”) really be counted as a separate programming language? And what about the declarative (descriptive) programming languages (such as HTML and CSS, which are used to design web pages, among other things), i.e., languages that do not contain any real flow logic? So, it is not easy to decide what you want to count as a new language.

At the time of writing, Wikipedia lists 713 different programming languages. This confronts anyone who wants to learn programming with the question of which language he or she should learn. We will deal with this in the following chapter. Here, we will look at the question of why there are so many programming languages in the first place. The answer to this question lies primarily in the different *fields of application* for which the languages have been designed, the different ways in which the languages fundamentally work (the so-called *programming paradigms*), and the *further development of existing programming languages* in a separate branch.

4.1 Different Fields of Application

Programming languages have been developed for different purposes and are therefore well suited for “their” main purpose, but possibly less so for other purposes. Examples include:

- the *R* language, which is particularly strong in the field of statistics,
- Apple's *Swift*, which was developed specifically for app programming,
- the *Structured Query Language (SQL)*, which is the standard for database queries,
- *PHP*, whose main purpose is to develop dynamic websites (websites that respond to user input).

The languages have special features for their respective “main purpose,” which make it possible to solve problems in this area easily. The statistics language *R*, for example, can handle tables and data columns (or vectors) very well, which may be difficult to accomplish in other languages. In the database query language *SQL*, it

is very easy to query databases for information and, for example, to determine “all customers who have bought our new product via the homepage at least twice in the last 12 months.” The web language PHP makes it very easy to output the results of a search in a product catalog or to process the login data of the user of a social media platform and then grant or deny the user access.

Not all programming languages have a special field of application though. These languages are therefore not called *special-purpose languages*, but *general-purpose languages*. These include the better-known heavyweights such as Java, C/C++, Python and VisualBasic. Nevertheless, due to the characteristics of these languages, areas have often developed in which they have become particularly popular. For example, C/C++, is the standard for system-oriented programming (i.e., the development of operating systems or device drivers), Python for data science (i.e., complex analyses of data, including the application of artificial intelligence methods).

However, the distinction between special-purpose and general-purpose languages is somewhat fluid. The widely used Java programming language, for example, is based on the idea of compiling the same source code once (into bytecode, see the previous chapter for more details) and then being able to execute it on all possible devices. In a sense, this is also a kind of special purpose. Nevertheless, Java is used today in many places where the portability of the developed programs from one system architecture to another is not at all important.

The purposes and fields of application for which programming languages are needed are changing, and so is the range of languages that is particularly favorable for the currently popular purposes. In the first half of the 1990s, web applications played a completely subordinate role. Instead, languages for programs that were permanently installed on the computer, for example from floppy disk or CD-ROM, were important. The key requirements were to design attractive and easy-to-use program interfaces in a simple way (preferably by drag & drop) and then to be able to react to the user’s behavior on these interfaces in an event-driven manner; and perhaps to be able to provide a convenient installation program without much effort. This made developers happy at that time. Then internet applications became more important and with them completely new languages that had to run in web browsers. They also had to be able to generate or modify HTML pages and communicate with web servers, for example to read data from databases on the server and display it for the user. Thus, languages for stand-alone applications permanently installed on a computer became less important relative to the new “upstarts.” To some extent, this also applies to Java, whose cross-platform approach was no longer so crucial now that the new languages were interpreted directly by the browser or ran on the server and simply returned a complete, dynamically generated website to the browser.

4.2 Different Paradigms

Programming languages are based on different *programming paradigms*. A paradigm in this sense is the fundamental principle of how programs are constructed and function in the particular language. The paradigms used in programming are

not without overlap, although some combinations are mutually exclusive; in fact, most languages follow multiple paradigms.

A particularly important paradigm (because it is followed by many programming languages) is *object-oriented programming (OOP)*. Simply put, OOP allows you to recreate objects as they occur in the real world, with their properties (attributes), and to work with them by applying certain actions (methods) that are suitable for this kind of object. Let us use a car as an example. A car is an object with various properties such as color, manufacturer, acceleration, and so on. In an object-oriented language, you could easily create a data construct representing a car, and set its properties to: color = midnight blue, manufacturer = BMW, acceleration = 9.1 s from 0 to 100 miles/h. There could also be properties that reflect the current driving state of the car, such as its current speed, how many people are in the car, and whether the turn signal is currently on. For many purposes, such a representation of objects is very useful. We will look at object-oriented programming in more detail later.

Another example of a (completely different) programming paradigm is the so-called *logical programming*, which is followed, for example, by the language *Prolog* (Programmation en logique). In the logical programming paradigm, the programmer defines rules that contain knowledge about a certain domain. For example, one could define as a rule that two children are siblings if and only if they have the same parents. Then one could tell the Prolog interpreter that Paul's father is Mark, and his mother is Julia; and further, that Beverly's father is Mark and Beverly's mother is Julia. Fed with this knowledge and the previously defined rule about the relationship between parents and children, the Prolog interpreter would now be able to draw the logical conclusion that Paul and Beverly are siblings completely independently and without the need for any further programming.

Like so much in the world, programming paradigms are subject to fashions and trends. When a paradigm becomes fashionable, more languages are created that follow this paradigm.

4.3 Further Developments and Dialects

Part of the diversity of programming languages can also be explained by the fact that “dialects” of languages have emerged, i.e., new languages that are closely based on another language but are further developments or (at least from the language inventor's point of view) improvements over the original language. Examples of this are C++ and C# (or “C sharp”), both of which are evolutions of the C programming language and have converted the original C language to the object-oriented programming paradigm.

But why not just adapt the original language instead of opening a new “branch” with the dialect? Reasons for this could be that the developer of the original language does not support the intended changes; or to ensure that code written in the original language is still executable, even if serious changes in the new language would make all programs written in the original language no longer being able to execute without errors.

Practically all programming languages (even if they are not further developments of other languages) are influenced by other languages. This can be seen quite clearly in the Wikipedia articles of the languages, that show systematically which other languages the language in question has been influenced by, and which other languages it has influenced.

If a language is a dialect of another language, it often has a name that is based on the name of the original language, as we have just seen with C++ and C#. Other examples of this phenomenon are VisualBasic (an evolution of BASIC) and Object Pascal (an object-oriented variant of Pascal). The latter is a good example of how a conversion of the original language to the object-oriented programming paradigm often leads to an extension of the name by “Object” or “++” (based on the “original pair,” C/C++).

Otherwise, the naming landscape of programming languages is very colorful and varied. Some programming languages have names that are based on the names of persons (the name of the developer or of persons honored by the naming), for example Ada (after the programming pioneer Augusta Ada Byron), Eiffel (after the French engineer and builder of the Eiffel Tower), Gödel (after the Austrian mathematician), or Wolfram Language (after the British mathematician and computer scientist Stephen Wolfram). The names of some other languages are abbreviations, as in the case of SQL (Structured Query Language), VBA (Visual Basic for Applications), FORTRAN (Formula Translation), COBOL (Common Business Oriented Language) or Prolog (“programmation en logique”), while still others are purely fanciful names, such as Delphi or Python.

Many (especially professional) programmers today are somewhat language-agnostic: they simply use the language that is best suited for their current project. In doing so, they look at the problem to be solved and then decide what they want to work with. They will not know all languages, of course, but they do know a certain range from which they can then choose. And this range of languages is usually much wider than that of professional translators. This works because programmers have internalized the basic concepts of programming and can therefore pick up new language easily. That’s what we hope to learn with this book. But before that, in the next chapter, we’ll look at how to choose a suitable language.



Which Programming Languages Should You Learn?

Contents

- 5.1 Subject of Your Project – 36
- 5.2 Customer Preference and Employer Preference – 37
- 5.3 Pedagogical Aspects – 38

Overview

In the last chapter we saw why there are so many different programming languages. But this diversity presents us with the problem of deciding which language or languages to learn. In Parts III and IV of this book, you will have the opportunity to familiarize yourself with two extremely popular programming languages, Python and JavaScript; to that extent, the author has already made a choice for you. Independently of this, however, in this chapter we want to address the question of what criteria you can use to select languages that might be of interest to you. After all, if you are just starting out in programming, you will certainly not remain with the first language with which you started programming.

5

Given the immense range of programming languages, the question naturally arises as to which of them you should bother with. As you probably already thought, or even feared, the answer turns out to be inconclusive: It depends.

5.1 Subject of Your Project

We have seen in the previous chapter that languages are differently suited for different application areas. So, it makes sense to choose a language for your project(s) that can comfortably achieve the goal. If you mainly want to build websites with dynamic elements (for example, if you want your website to validate address data when a customer creates a new user account), then you're most likely to go with JavaScript, because JavaScript runs in the web browser and it's great for working with user input before it's sent to the web server. In the next step, however, you won't be able to avoid involving the web server itself if the customer data is also to be written to a database. To do this, a script runs on the web server, for which PHP will probably be the language of your choice. PHP was developed for exactly this purpose. It runs on the web server and makes it easy to work with databases in the background and dynamically display the results of database queries as a web page in the user's browser. However, if you want to develop a real Windows application, neither PHP nor JavaScript will be on your shortlist, because they are made to run in a web environment. For a true Windows application, you're more likely to turn to Java, Visual C/C++, VisualBasic, Delphi, or something similar. These languages allow you to develop classic stand-alone applications, design pretty interfaces for them, and respond to the user's actions. If you want to develop programs for analyzing large amounts of data, you'll probably turn to R or Python. And for mobile apps, you might use Objective-C or the relatively new Swift developed by Apple.

We have always talked about a language being particularly good for a particular purpose. But what makes a language better or less suitable? There are two things in particular:

1. The environment in which the language runs, such as a PC operating system (for stand-alone applications), a website in the user's web browser, a web server, or a mobile operating system (for mobile/tablet apps).

2. The tools that the language provides to accomplish the task you set out to do. For many languages there are libraries and frameworks, which are ultimately extensions that give a certain functionality to the language. For example, there might be a library for searching other websites for content and extracting it (“web scraping”). Now, if you want to write a program that keeps track of the price of a particular online shop item and notifies you when it changes, you should choose a language that features a strong web scraping library. When it comes to libraries and frameworks, open-source languages with an active, engaged user community regularly have an advantage over their commercial competitors.

But languages may also come with important out-of-the-box features that make the language more suitable for your purpose. For example, this could be the language’s data structures, or the way data is processed in the language. If you want to work with data tables and columns from these tables, for example, the language R, which is primarily used for statistical purposes, is a good choice, since it offers extensive out-of-the-box support for these data structures in the form of data frames and vectors.

5.2 Customer Preference and Employer Preference

If you work as a freelancer, the client may have their own ideas about which programming language to use. This may be because he wants to smoothly integrate your program into his larger overall software, which is written in that language. Or because he and his internal developers simply know the language best. If your customer doesn’t understand how your software works, he won’t be able to develop it further.

Also, if you are looking for a job as a developer, it is important to know one or more of the languages that are in demand in the market. But how do you find them?

One way is to use the *TIOBE index* as a guide. This list, maintained by the *TIOBE* software company, evaluates the relevance of programming languages based on the number of hits they get in various search engines when you search for the name of the language.

The number of question and answer posts in the *Stack Overflow* forums, which are extremely popular among programmers, is also a good indicator. The *Stack Overflow* operators publish quantitative evaluations of the programming languages to which the posts refer. In addition, *Stack Overflow*’s regular user surveys offer a clue, as among other things, they regularly ask about programming languages used.

But what if you don’t trust all these statistics? After all, there are reasons to at least critically question the validity of such numbers. Could it be that the many questions about a particular programming language on *Stack Overflow* only come from the fact that the language is very complicated and therefore difficult to use? The same argument could be used to question the validity of the search hit based

TIOBE popularity statistics. Further, one might question whether Stack Overflow’s user surveys are representative—perhaps certain groups, such as nerds, who engage with a language for reasons other than its practicality are more inclined to participate in such surveys.

These are all legitimate considerations. Nevertheless, high numbers of questions and search hits at least show that there are people who are concerned with the language. If you really want to do without statistics, you can look for anecdotal evidence for the importance of certain programming languages. Go to the major job portals, search for programmer/developer jobs, and see what languages are required in the job ads. Studies such as those of the programming course provider *Coding Dojo* examine job ads in a systematic way (again statistics!), particularly those on the search engine ► [indeed.com](https://www.indeed.com). If that’s still not enough evidence for you and you’d rather hear the opinion of (mostly self-appointed) experts, go to *YouTube* and search for “Programming languages to learn in [insert year here].” You’ll find dozens of YouTubers with more or less good insights into the market situation who willingly explain which languages they prefer and for what reasons. Stack Overflow

5

5.3 Pedagogical Aspects

At the beginning, you might deliberately choose languages that take different paths in terms of their approach to programming (i.e., their *programming paradigms*, which we will look at in more detail in the third part of the book) in order to learn about a wide range of possibilities. This sounds a bit easier than it is, because most languages do not follow a single paradigm at all, but are *multiparadigmatic*, picking elements from different paradigms. However, a good mix still creates a sound understanding of the different approaches.

In the same way, you could start with languages that are relatively easy to learn. After all, achieving initial learning successes quickly is a key motivation to stick with it and keep going. We also follow these pedagogical points of view in the present book.

You have probably noticed that we always assume here that you will ultimately learn not just one but several programming languages. But is it realistic to really master several languages, and to do so sufficiently well?

How is your English? How’s your French? How is your Spanish? Can you make yourself understood to a native speaker in all three languages? Probably not (unless you’re really gifted at languages). It’s apparently quite hard to do well in natural languages. Programming languages are a bit different. Once you understand the basic concepts of programming, you can learn other languages with manageable effort.

As with natural languages, a developer with many years of practical experience in that language, a “native speaker” so to speak, will of course be far superior to you and know tricks you have never heard of. In this way, he will be able to solve a given task with a program that is shorter, looks more elegant and runs faster than one you can write. Still, learning the basics of a new language, at a level that will enable you to write programs successfully, is not difficult. While this statement may

5.3 Pedagogical Aspects

seem like a somewhat bold assertion to you at this point, we will prove it in this book.

After extensively covering the basic concepts of programming in the second part of the book, you will learn *two* programming languages, *Python* and *JavaScript*, in the third and fourth parts. These two languages are not only very useful, but also extremely popular by all measures. In the July 2022 *TIOBE Index*, they rank 1st (Python) and 7th (JavaScript). The leading languages in the index—Java, C, and C++ – are a bit more complicated and not quite as well-suited for entry-level programmers as Python and JavaScript. As of May 2023, JavaScript and Python were the languages with the highest number of questions on, with the number of Python-related questions in particular increasing in recent years.



Some Tips

Contents

- 6.1 Learn the Basics First! – 42
- 6.2 Play! – 42
- 6.3 Don't Lose Heart! – 42
- 6.4 Start Small, and Let Your Program Grow Incrementally! – 43
- 6.5 Let Beauty Be Beauty! – 43
- 6.6 Document! – 43

Overview

Before we get into the basic concepts of programming in the next part of the book, you'll find some encouraging tips for getting started in the fascinating world of programming in this chapter.

6**6.1 Learn the Basics First!**

Start small when learning a new language. With very few basic elements you can usually already develop executable programs. You don't have to know every detail. You will never know the language "completely" anyway, there are always more features, more libraries. That you don't know yet. The important thing is to gain a basic understanding of the language, and this is usually much easier than it first appears. Therefore, focus your efforts on the central concepts of the language first. The basic questions we cover in Part II of this book will help with this. Don't try to read tons of literature to theoretically understand every aspect of the language before you venture into your first program. Quite the opposite! Start writing your own small programs as early as possible. Your knowledge will expand naturally over time.

6.2 Play!

Dare to try things out. Unlike in driving school, nothing can break in programming if you play around a bit. When you try things out, you quickly learn what works and what doesn't. As we all know, we learn especially well from mistakes. By trying things out, you learn things that are not described in any book.

6.3 Don't Lose Heart!

Don't be discouraged if your program doesn't do what it's supposed to do at the beginning, or if you keep getting an error message and haven't the slightest idea what the problem might be. Of course, we try to avoid that sort of thing with the approach we take in this book. But the truth is, the best programmers make mistakes all the time. Ever wonder why the apps on your phone update so often? Many of those updates are just bug fixes. Making, finding, and ironing out bugs is part of the craft of programming, and by no means the least important. This is sometimes tedious and nerve-wracking, but no reason to throw in the towel. By the way, a little break sometimes works wonders.

6.4 Start Small, and Let Your Program Grow Incrementally!

When writing a program, think about what the actual task is that the program should accomplish, and what features your program really needs to accomplish it. Develop these first. Once the basic functionality is in place, you can add more complexity, step by step, to increase usability or to make the program more robust against user input errors.

6.5 Let Beauty Be Beauty!

It is important that your program works, that it does what it is supposed to do, and that it is robust, that it is not easily upset if, for example, the user makes an entry error. Far less important is that the program is a model of elegance and efficiency. You sometimes hear programmers talk about a piece of code being “elegant” or “beautiful.” Such epicurean statements are expressions of the commonly held view that programming is both a science and a (craft) art. Don’t go overboard with the art, though. A professional with years of experience may find your programs a bit clumsy and see some room for optimization. That’s not a bad thing, though. It’s better to try something new than to fiddle around with an already functional program to make it even more elegant. That way you learn a lot more!

6.6 Document!

Right at the beginning of your programmer life, get into the habit of documenting your program code, especially by annotating it with comments that explain how the code works. This is important so that you can understand your program later. Commenting is probably the most hated but at the same time one of the most valuable activities in programming. We will look at commenting in more detail in ► Chap. 10.

The Basic Concepts of Programming



Nine Questions

Overview

In this chapter, we'll get an overview of the 9 questions that we'll use to learn the basic concepts of programming in this part of the book. Based on these basic concepts, we will then work our way through the Python and JavaScript programming languages in the following two parts of the book. But understanding these basic concepts is not only useful for learning Python and JavaScript, but for any programming language.

This second part of the book is devoted to the *basic concepts* of programming. The basic concepts are implemented in virtually all programming languages in one way or another. Once you understand these concepts, you will discover many similarities between different programming languages. These similarities are what make learning new programming languages much easier.

We will summarize the basic concepts of programming under *9 big questions*. If you are learning a new programming language, you can use these questions to guide your learning process. Of course, you can work on the questions in a different order than the one proposed here. And, at some points, when you are dealing with one question, you have to “anticipate” another question. For example, if you are dealing with the storing of data in your language, you must also be able to—at least in a rudimentary way—output data (which is the subject of another question) in order to try something out practically. But even though the questions are not (and cannot be) entirely self-contained, they form a useful framework and a roadmap to understanding the new language, from the ground up.

Even if you do not follow the 9 questions explicitly you must be able to answer the 9 questions if you want to understand the basics of a programming language that is new to you.

Now in this part, we will first learn the basic concepts of programming using the 9 questions. At the end of each chapter, you will find a section called *Your Roadmap to Learning a New Programming Language*, which summarizes the key points of each chapter that you should familiarize yourself with in order to learn a new programming language.

In the next two parts, we will apply the basic concepts to learn the fundamentals of Python and JavaScript. These two parts of the book are also logically structured based on the 9 questions. Therefore, at any time while you are learning Python and JavaScript, you can flip back to the corresponding Basic Concepts chapter to refresh one or two basic considerations.

The 9 questions are as follows.

What Do I Need for Programming?

Here we will first deal with the *tools* you need to write and run programs in the programming language. We will also look at where you can get more information and help if you get stuck.

What Do I Have to Do to Get a Program Running?

Next, we develop a very first, very simple program. In doing so, we already learn certain *basic rules of program structure* and the *grammar of the programming language* and understand how programs are *written* and *executed*. To be able to do this is a basic requirement for all further learning steps, during which we also want to work very practically and write executable programs.

How Do I Make Sure That I (And Others) Still Understand My Program Later?

Your programs must at least be understandable to you, but sometimes also for others. It helps to adhere to certain *conventions* as to how program code should look, and to *comment* your program, that is, to provide explanations. If your program code is also to be used by others, you must *document* how exactly this can be done.

How Do I Store Data to Work With?

All software works with data. Therefore, we will next look at how data is stored and processed in programs. In doing so, you will learn about the central concept of *variables* and see how *objects* can be used to represent things in an elegant way.

How Do I Input and Output Data?

The data that is held in the program for processing must come from somewhere. In addition, the processing results are to be communicated “to the outside” somehow. These are the two issues in this question. *Input* and *output* of data in all its forms, be it by the user via the program’s interface, or through files or databases, is a core requirement that all programs must serve in one way or another.

How Do I Work with Program Functions to Work with Data and Trigger Actions?

In the context of this question, we deal with the actual processing of the data. This is done in most programming languages by *functions*. The input and output of data (previous question) also makes use of such functions. We can develop functions ourselves, or we can use functions that the programming language, or the community of developers, provides for us. Being able to work with functions safely is a key to success when working with any programming language.

How Do I Control the Program Flow and Make the Program React to User Actions and Other Events?

Our programs should not be rigid, but should react to their environment, for example to the wishes of the user. In the context of this question, we will deal with how we can let our programs react to external influences and events and branch into alternative sections depending on events in the program flow.

How Do I Repeat Program Instructions Efficiently?

Often, parts of a program are repetitions of the same pattern. Repetitions can be realized elegantly and efficiently in most programming languages with *loops*.

How Do I Search for and Fix Errors in a Structured Way?

Writing a program and making it perfect on the first try is an illusion—even for professionals. Unfortunately, errors are an inevitable part of everyday programming. Therefore, to conclude our tour through the basic concepts of programming, we will deal with how we can systematically find, diagnose, and eliminate errors, and which tools can help us to do so.



What Do I Need for Programming?

Contents

- 8.1 Tools – 52**
 - 8.1.1 Compiler and Interpreter – 52
 - 8.1.2 Code Editors – 53
 - 8.1.3 Integrated Development Environments (IDEs) – 54
 - 8.1.4 Simple Online Development Environments – 59
- 8.2 Help and Information – 60**
- 8.3 Generative Artificial Intelligence Like ChatGPT – 62**
- 8.4 Your Roadmap to Learning a New Programming Language – 63**

Overview

Before we can start writing programs, we first need to get the right software tools ready. These will include a compiler and/or interpreter, which translates the program source code you have written into machine language and thus makes it executable for the computer. This should also include code editors, with which the source code of the program is written in the first place. Integrated development environments combine these and other tools under the umbrella of a common user interface.

In addition to such “technical” tools, you will also need help with your programming from time to time. Therefore, in this chapter we will also look at how and where you can find information and support related to your programming language.

In this chapter you will learn the following:

- How to get a compiler/interpreter for your programming language
- What functions code editors offer, and how they differ from normal text editors
- What features Integrated Development Environments (IDEs) offer, and how they differ from pure code editors
- What popular code editors and Integrated Development Environments are available
- How and where to find information and support for your programming language on the internet
- How you can use artificial intelligence (AI) tools like ChatGPT for programming.

8.1 Tools

- » How many times do I have to tell you? The right tool for the right job!
(Montgomery “Scotty” Scott in Star Trek VI—The Undiscovered Country)

8.1.1 Compiler and Interpreter

From the previous chapter, you already know that—depending on the programming language—you need an interpreter or compiler to translate your programs into executable machine code that the computer can understand, or to have your program source code interpreted and executed at runtime. For many programming languages, compilers or interpreters can be downloaded free of charge from the internet. This is especially true for languages where further development is done by a de facto non-profit organization, as is the case with Python or R, for example. Even for proprietary languages that are only made available by a specific commercial company, such as the Object Pascal dialect Delphi, developed and distributed by Embarcadero, there is often a Community Edition, or a free version with a limited, but for the private user, sufficient range of functions. However, the use of the Community Edition for the development of commercial applications might be subject to legal restrictions. So, if you plan to sell your self-developed software,

familiarize yourself with the license conditions in advance. For some programming languages, no separate interpreter is required at all. If you want to program with JavaScript, for example, the web browser takes over the interpretation and execution of your program code. If you want to develop server-side applications with PHP, the interpretation of the code runs directly on the server, only its results are returned to the client and made visible in the browser.

8.1.2 Code Editors

To be able to use compilers or interpreters meaningfully at all, you must first write a program. As we have already seen, a program is ultimately nothing more than a text of instructions written in a special language. The emphasis here is on text. Because the source code of your program is simply text, you can use any program that allows you to edit unformatted text (that is, text without special formatting such as bold or italics). If you use Windows as your operating system, for example, you can use the very simple Windows Editor. However, it can do little more than open or create a file, type in some text, and save the file after editing. Although these rudimentary functions are in principle sufficient to write computer programs, it is nevertheless advisable to use an editor that has been specially developed for programming, or at least comes with functions that make programming easier.

One particularly important function is called syntax highlighting. As you will recall, syntax is the grammar of the programming language. Syntax highlighting is, as you can already guess from the name, a functionality that highlights certain parts of the program code to make the code more readable. For example, keywords (i.e., special, reserved “words” of the programming language) or the names of variables are each marked by different font colors and/or font styles (such as bold type). Programming without the visual support of syntax highlighting is of course always possible, but considerably less comfortable, because without the highlighting it is more difficult to direct the eye to the right places in the program code and to quickly grasp certain structures in the code. Since syntax varies from programming language to programming language, syntax highlighting must work differently in each language. A number of text editors support syntax highlighting for a variety of different programming languages, either out of the box or through extension packages. Text editors that fall into this category include Atom, Notepad++, Sublime Text, Vim, or Visual Studio Code. Many of the editors are available free of charge, either completely or at least in a somewhat limited version that can work very well, such as the popular Sublime Text.

Given the large number of text editors on the market that offer special features for working with program code, such a list cannot be exhaustive. So, it’s no surprise that there are tons of articles, blog posts, and videos on the internet about which is the best code editor. This decision depends on personal preferences, especially with regard to the functions offered and certainly also to the visual appearance; after all, the eye does the programming. A special aspect of the visual appearance are the dark themes: Maybe you have already looked over the shoulder of an experienced programmer and seen that he is sitting in front of an editor with a dark, almost

black background, from which the program code, colored by syntax highlighting, clearly stands out. Why is this dark background (“working in *dark mode*”) so popular? Maybe it’s because it’s just “cool” to set your editor this way, showing that you belong to the secretive community of programmers who are constantly hammering code into their keyboards that is almost incomprehensible to the average person. This assumption may be partially correct. However, another factor is much more important. A dark background is much more pleasant for the eyes than a bright, even white background. If you have to concentrate on your program code for hours on end, you will quickly come to appreciate the discreet and less dazzling background against which the program code shifts into the foreground in high contrast. In this book, the screenshots of code editors are always kept bright, but only because this makes them easier to reproduce in print. In fact, the author usually works with a dark background, which can be set in most editors, and is the default in some, such as Sublime.

8

8.1.3 Integrated Development Environments (IDEs)

In addition to code editors, there is another group of tools, the *Integrated Development Environments* (IDEs). These tools go beyond pure code editors in their functionality.

Important features that most IDEs combine are:

- Direct call of the compiler/interpreter
- Functions for efficient source code editing that go beyond syntax highlighting
- Functions for the graphical construction of user interfaces if the language supports graphical user interfaces
- Troubleshooting functions (debugging)
- Functions for managing entire projects with multiple files.

A frequently found function in the area of source code editing is, for example, auto-completion: When you type a few letters, possible instructions/keywords matching your input appear directly as a suggestion. This can increase the speed of code writing and at the same time reduce the error rate. Also, an automatic syntax check is often available, which already shows you possible errors while you are still working on your program code. For example, it alerts you if you have opened a bracket but did not close it later. In this way you can detect and correct errors in the program at an early stage and are not surprised by error messages only when you call the compiler or interpreter. Practical features in the area of source code editing often include deep help integration. For example, it is often possible to call up help information on the statement you are currently working with directly from the IDE.

Quite often, you not only have to write code for your program, but also design a graphical user interface (GUI). Some IDEs offer extensive support for the developer. Often, you can piece together your interface from standard elements such as buttons, input fields, or list boxes using drag & drop and then adjust the properties of the standard elements according to your wishes, such as, give a button a new

label, or change its size or even color. After that, all you must do is connect the UI elements to your program code and address them from within the program so that something happens when, for example, the user clicks on the button. IDEs that support UI design in this way include Embarcadero's RAD Studio or Microsoft's Visual Studio.

In addition to the direct call of compiler or interpreter and practical functions for source code editing and interface design, IDEs typically also offer functions for debugging, i.e., the systematic search for and correction of errors. An important feature in this area is, for example, the creation of breakpoints in the source code. If you start a program after you have set up a breakpoint, it will only run up to the point in the code where the breakpoint is located; later, you can manually let it continue running beyond the breakpoint. Working with breakpoints allows you to check whether a program runs without errors up to the point marked by the breakpoint. Also, you can look at the contents of variables at the breakpoint location. This monitoring of the contents of variables is another important debugging feature. It allows you to “look inside” variables while the program is running and to see their current contents, and even to change them if necessary.

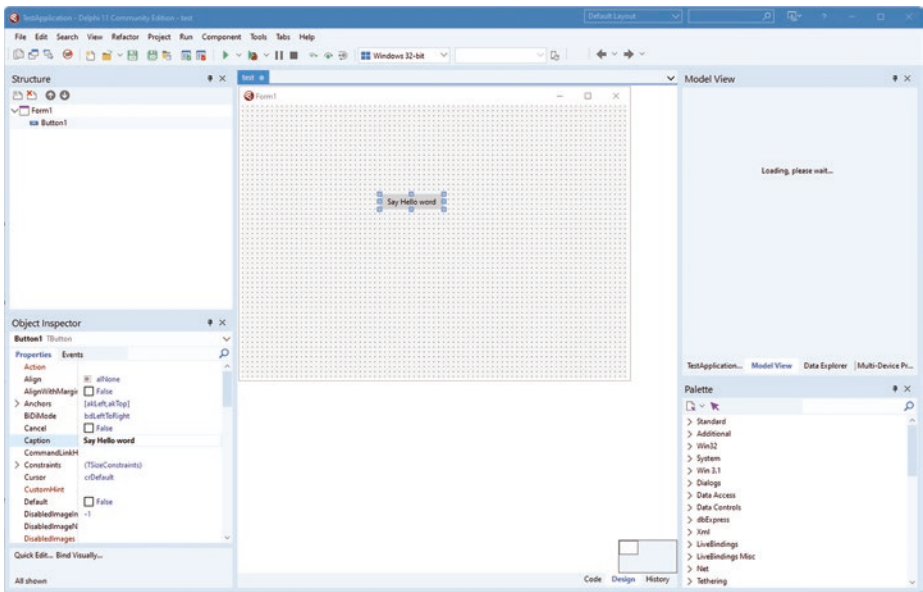
If your program consists of several source code files, you can often save them in an IDE as a coherent project. This means that all you have to do is open the project and all the code files are available to you. You can also save certain settings, such as those of the compiler, on a project-specific basis.

As in the case of code editors, there is also an almost unmanageable number of IDEs on the market, some free of charge (open source or as a community edition), some with costs. Some focus on one programming language (for example RStudio or PyCharm from JetBrains for Python), others are built to handle a variety of different languages, sometimes through appropriate plug-ins (for example Microsoft's Visual Studio or the open-source solution NetBeans).

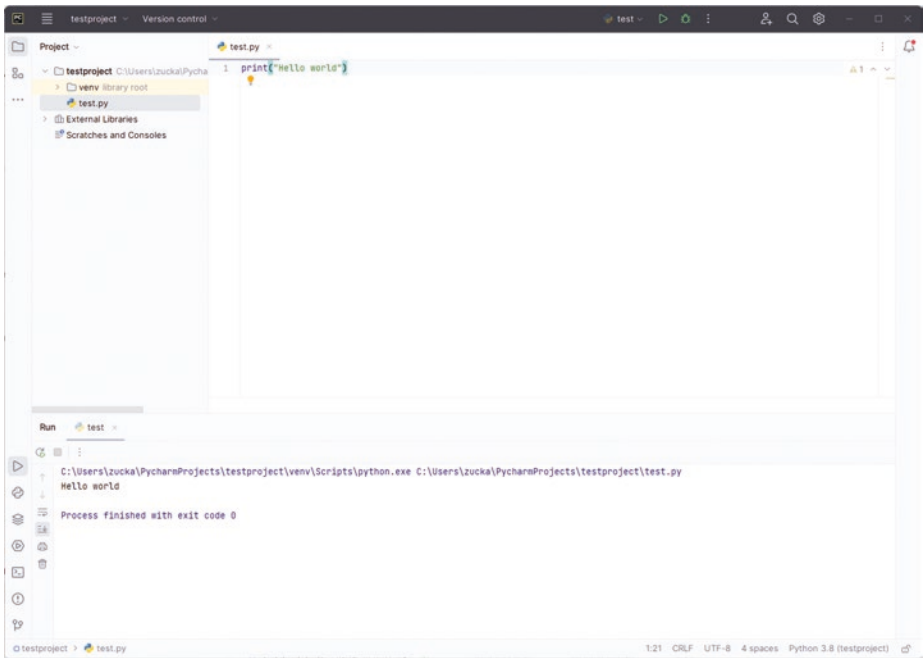
IDEs for mobile applications, such as Google's Android Studio, also allow you to simulate the operation of the developed app on a mobile environment with certain parameters (for example, hardware equipment, configuration) and to estimate the demand on system resources, such as processor load or mobile data transfer. These IDEs are thus designed for a specific purpose, namely the development of mobile applications, rather than around a specific programming language and therefore often support different languages; in the case of Android Studio, for example, C/C++, Java and Kotlin.

The transition between code editors and IDEs is somewhat fluid. Many code editors allow a compiler or interpreter to be attached and thus already have the core functionality of an IDE, but with the exception of syntax highlighting they offer little support in the area of language-specific code editing, debugging or interface design.

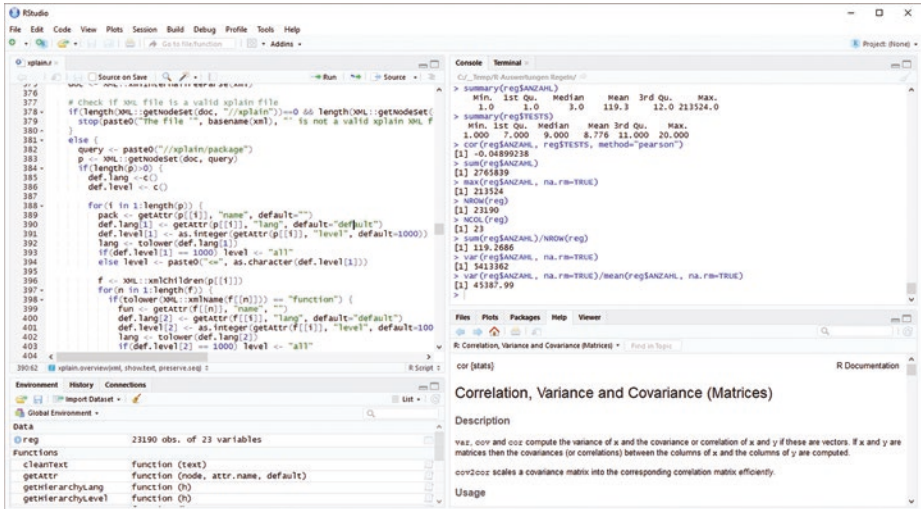
■ Figures 8.1, 8.2, 8.3, and 8.4 show different IDEs. ■ Figure 8.4 shows a very old IDE for C/C++ from Borland, which ran under MS-DOS. However, you can see very clearly important IDE functions in the menu bar, such as Run, Compile, Debug and features for project management.



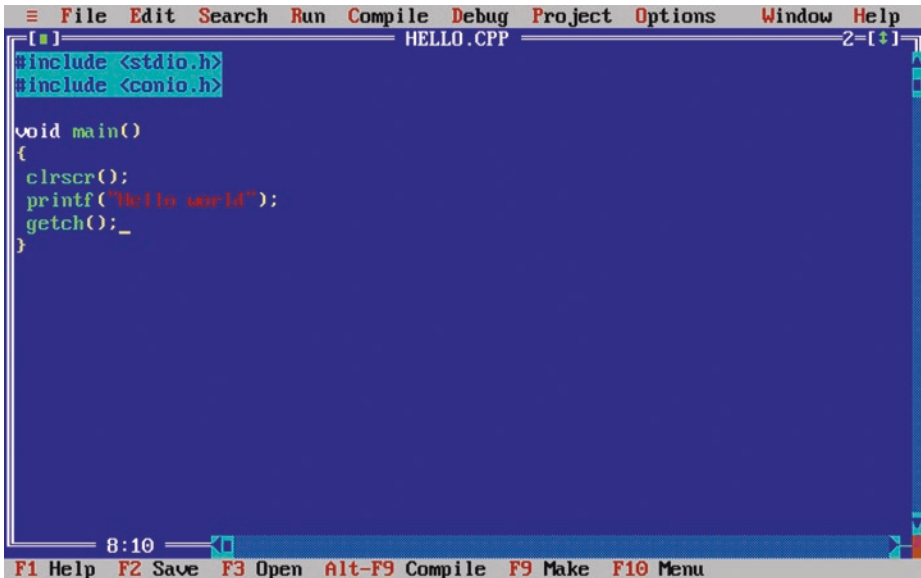
■ Fig. 8.1 The Integrated Development Environment (IDE) of Delphi



■ Fig. 8.2 The PyCharm Integrated Development Environment (IDE) for Python



■ Fig. 8.3 The Integrated Development Environment (IDE) RStudio for R



■ Fig. 8.4 The TurboC Integrated Development Environment (IDE) for C/C++

■ Table 8.1 shows a selection of IDEs that support some common programming languages natively. Several IDEs, such as Eclipse or NetBeans, can be extended by add-ins so that they support a whole range of languages.

When deciding whether to use a code editor or an IDE, it is natural to consider how much you want to use tools that only IDEs provide (such as debugging features or interface design functions). But even if you don't plan to do that, the big

Table 8.1 Selected Integrated Development Environments (IDEs)

Programming language	Selected from IDEs
C#	SharpDevelop, Visual Studio
C/C++	AppCode, C++ Builder, CLion, NetBeans, QT Creator, Visual Studio
Java	AppCode, Eclipse, IntelliJ IDEA, JBuilder, NetBeans
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, PyCharm, Rodeo, Spyder, Thonny
R	Rcommander, RStudio
Ruby	Aptana Studio, Komodo IDE, RubyMine
Swift	AppCode, Xcode
VBA	Microsoft Office, Visual Studio Tools for Office (VSTO)

advantage of IDEs to provide all features under one roof is still a weighty one, especially if you think about the central tools like compiler or interpreter that you need all the time during the development process. On the other hand, IDEs are often quite complex programs themselves, with a plethora of buttons, different toolbars/ribbons, windows and tabs that you first have to find your way around. A good impression of the complexity is given by the  Figs. 8.1, 8.2, and 8.3. Since the IDEs have been developed for professionals, the manufacturers are obviously not so concerned that you should be able to learn how to use all the functions and options in a few minutes. After all, the IDE is not intended to be used temporarily or only now and then, but permanently as the control center for all programming work. Therefore, it may take some time before you have a complete overview of the possibilities of the new tool. But once you have found your way around the IDE, you can work productively, because that is exactly what these tools are developed for.

Tip

Try different code editors and IDEs and see what you get along with best. Don't be confused by the many features. You need very few of these features to write and run programs. You will learn to use and appreciate more and more features of the tools as time goes by. Don't try to understand everything right at the beginning, but concentrate on the really essential features, i.e., those for editing and compiling/executing the code.

Ultimately, you must decide for yourself which tools you want to use. It's a good idea to try out different code editors/IDEs and only then commit. For the purposes of this book, we will be working with both a full-fledged IDE, PyCharm for Python, and a classic code editor, Sublime Text, when dealing with JavaScript. Writing these paragraphs, it occurs to the author that he himself seems to unconsciously follow a simple rule—if the programming language does not require a special, stand-alone compiler or interpreter on one's own computer (i.e., for JavaScript or PHP, for example), a code editor is the tool of choice. For languages that require an installed compiler or interpreter, a language-specific IDE comes into play. Ultimately, however, there is no golden rule that always applies to everyone. Only one thing helps: try it out!

8.1.4 Simple Online Development Environments

If you want to try out a language without installing all the necessary tools on your computer, you can often use special websites that allow you to enter and execute code directly. All the features necessary for execution, such as compiling and interpreting the code, are provided by the website. Some examples of such “online IDEs” are:

- ► <http://cpp.sh/> for the C++ programming language,
- ► <https://www.compilejava.net/> for Java,
- ► <https://js.do/> for JavaScript,
- ► <http://phptester.net/> for PHP,
- ► <https://www.pythonanywhere.com/> for Python,
- ► <https://rextester.com/> for a whole range of languages, including C#, Haskell, Kotlin, Ruby, Pascal and Visual Basic.

More such platforms can be found very easily by typing “try *programminglanguage* online” (where *programminglanguage* is replaced by the name of the language you want) into a search engine. In most cases, you don't even need to create an account; you can start writing code right away. Sites that require a (free) account, such as ► <https://www.pythonanywhere.com/>, usually also allow you to save files in the cloud and reuse them later.

Web services like the ones mentioned are of course no substitute for a real development environment, since they usually have very limited functionality and the execution of programs may also be subject to restrictions (in some cases, for example, a program may only take up five seconds of computing time). If you want to get serious about the language, there is no way around installing the necessary tools on your own computer. Nevertheless, these websites are an interesting opportunity to try out a language without risk and unnecessary effort.

8.2 Help and Information

Having the right tools at your disposal is not everything. From time to time, you will need help. Therefore, it makes sense to think about where you can get more information about your programming language in advance. For example, if you have no idea how to tackle a certain problem, don't know what certain commands of your programming language do or how to use them, or you don't understand the sometimes-cryptic error messages of the interpreter or compiler. In all these cases, it is helpful to have an immediate contact point where you can find support. Such contact points are a significant resource in practical work, not only for programming beginners. Besides books like this one, of course, the internet offers a myriad sources that hardly leave any information need unsatisfied—provided, of course, that you can find them.

Many programming languages, both open source and proprietary (vendor-specific), come with extensive web documentation, where you can find detailed information about specific commands of the programming language. Examples of these help offers are PHP's Function Reference, Python's Library Reference or Microsoft's VBA Reference for Visual for Applications (VBA).

Part of the official documentation is usually not only such a function reference, i.e., a dictionary-like reference book that describes what certain commands of the programming language do and how they are used, but often also a language reference. Language references explain the grammar of the language in question, the syntax, and thus describe how to correctly formulate instructions in the language. However, such language references are not always easily digestible for novice programmers. That's why some official documentation also offer tutorials for beginners and "Getting Started" articles as a complementary component.

Putting a browser bookmark on the official language documentation, especially on the function reference, is recommended to anyone who wants to get serious about a programming language. It is usually the first place to go when you want to understand what a particular command does and how to use it.

In addition to the official documentation, there are of course also unofficial information and help channels that are not operated by the organization responsible for the programming language. In practice, the most important such channel for many programming languages is the internet platform Stack Overflow (► <https://stackoverflow.com/>). With over 17 million questions on a wide variety of programming languages, it leaves little to be desired. If you search there for a solution to a concrete problem, you often get the impression that every conceivable question has already been asked by someone before. But not only do the large number of answered questions and the high coverage of different programming languages make Stack Overflow an incredibly useful source of information, the quality of the answers is also generally very high.

The high quantity and quality are achieved on the one hand through gamification, i.e., the playful setting of incentives by crediting forum participants with points ("reputation") for certain actions. These points are not only visible to everyone as a proof of competence next to the respective username, but also allow the

use of certain functions that are not available to all users. One important such function (which can be accessed even with a relatively low score) is the up- and down-voting of answers, i.e., the rating of other users' answers. This helps readers to better assess the quality of the answers. In turn, the authors of up-voted answers receive reputation points, which increases the incentive to provide high-quality answers. Also, the questioner can mark one of the answers as the best. Other users will then recognize by the large green checkmark next to the answer that it is the one that ultimately helped the questioner solve their problem. The author of the answer is also credited with reputation points for this. In this way, not only is a system of mutual quality control operated, but also a strong incentive is set for investing in work for others, from which you have nothing but the good feeling of having helped another user. With Stack Overflow, in addition to the good feeling, you also gain reputation and rights, which is obviously beneficial to the ego of many participants.

In addition to gamification, the strict rules in the forum also play a role in the high quality of the information. For example, questions that are duplicates of other questions or that are off topic in the respective forum area are immediately closed by the moderators. The questioner is required to provide a minimal, but executable code example for his problem and to formulate his question exactly already in the title of the post. Stack Overflow's own Code of Conduct calls for friendly interaction, but the tone in the forum sometimes seems a bit rude to Stack Overflow newbies.

Although the tone sometimes takes a little getting used to, Stack Overflow is a first-class source of information. Even incomprehensible compiler or interpreter error messages—a common annoyance in programming—can usually be found on Stack Overflow. Since Stack Overflow supports many different programming languages, it is important to always include the name of the language in the search query.

If you search for a question on Google, Stack Overflow's search hits are regularly listed high up. By adding “site:► [StackOverflow.com](https://stackoverflow.com)” to the Google search query, results from Stack Overflow are listed exclusively; of course, ► [StackOverflow.com](https://stackoverflow.com) also has its own search. In addition to Stack Overflow, there are a number of other forums, including on well-known platforms such as Facebook or Reddit. Numerous blogs also provide good tips and how-tos for specific problems.

But when should you use the official documentation, and when should you use other sources like Stack Overflow? If you already know which command to use and now it's just a matter of understanding how to do it correctly, then the official function reference for the language is usually the best place to start. But if you don't know which command or approach to take to a problem, Stack Overflow (or a similar forum) is your first choice. Here, you will find good solutions to many common problems. Similarly, if you can't decipher error messages from the compiler or interpreter, or just can't figure out why a command you looked up in the function reference behaves the way it does, Stack Overflow is the place to go. The nice thing about Stack Overflow is that due to the large number of questions already asked, it's quite likely that yours will be there, and you'll get your problem solved immedi-

ately without having to post a question yourself and then wait for hours or even days for an adequate answer.

8.3 Generative Artificial Intelligence Like ChatGPT

Artificial intelligence tools such as ChatGPT from OpenAI, Gemini from Google, Claude from Anthropic, or Llama from Meta are a very different kind of tool. They are built on top of powerful Large Language Models (LLMs) and are dialog-based, meaning they allow the user to have a “conversation” with an artificial intelligence. This allows questions to be asked that are not yet found in this form in any internet forum. In addition, the tedious transfer work of applying someone else’s question and the answers they received to your own situation is eliminated, because with tools like ChatGPT, you can ask exactly based on your own situation and receive an answer that is tailored precisely to your own problem. The AI tool is easy to use because it communicates in natural language; usually, a whole range of natural languages are supported in addition to English, including Spanish, French, German, Italian and Chinese.

Tools such as ChatGPT are examples of generative artificial intelligence, which means that they are able not only to compile and present known information, but also—seemingly creatively—to generate new content and develop new solutions. Together with the ability to work in numerous programming languages, these tools are very useful in programming. Areas of application can be such as:

- Understanding foreign code: If you don’t understand a piece of code that you didn’t write yourself (e.g., from an internet forum or blog), just have it explained to you (“Explain what the following code does”).
- Looking things up: You can use tools like ChatGPT like a language reference or textbook. You can ask very specifically how to use a certain function in a programming language, for example, but also more generally about which functions (and external add-on modules) are available at all to solve a certain problem. The good thing (especially in the first case) is that if you don’t understand the answer, you can ask for a more understandable explanation (or examples) at any time.
- Writing code: The great strength of generative artificial intelligence is to write code itself (“Write a Python program that ...”). There are numerous examples circulating on the internet in which developers have used tools such as Chat GPT to develop large applications (for example, whole web platforms) in a short time and with little effort and have (allegedly) made a lot of money from their commercial exploitation.
- Finding and eliminating bugs in programs: Feeding code and error messages to AI tools often helps you figure out the bug faster than if you were to laboriously dig through the code yourself. ■ Figure 8.5 shows an example.
- Comment and document code: The tedious work of commenting and documenting code (i.e., writing explanations of how the code works and how it can be used), which is rather unpleasant for most programmers, can usually be done well by generative AI tools.

8.4 Your Roadmap to Learning a New Programming Language



Anonymous

I have the following Code (in backticks):

```
`
# Global counter
action_count = 0

def perform_action():
    action_count += 1
    print("Action performed.")

# New action
perform_action()
`
```

Running this code I get the error message: "UnboundLocalError: cannot access local variable 'action_count' where it is not associated with a value". What does this error message mean?



ChatGPT

The error message "UnboundLocalError: local variable 'action_count' referenced before assignment" occurs because of how Python handles variable scopes. In your code, `action_count` is defined outside the `perform_action` function, making it a global variable. However, when you try to modify `action_count` inside the function with `action_count += 1`, Python treats `action_count` as a local variable within the scope of that function. Since you haven't defined `action_count` inside the function, Python raises an error because it tries to update a variable that it considers local but hasn't been initialized within the function's scope.

Fig. 8.5 Debugging with ChatGPT

The quality of generative artificial intelligence results is often really good and very impressive, especially for first-time users. Nevertheless, tools like ChatGPT do not work without errors. The results always need to be checked, code written by AI tools needs to be tested thoroughly. Own knowledge of the programming language used is mandatory for this reason alone. However, because the systems are dialog-based and can refer to previous inputs and outputs, you can point out errors and ask ChatGPT to correct them. Even if the tool has not made any real mistakes but the result does not yet correspond to your ideas and requirements, you can have it "sharpened" again in dialog with the AI tool. In addition, due to the dialogue nature of the process, you can have complex programs developed step by step, which increases the probability that the desired result will come out in the end. Since Chat GPT is free of charge in the basic version, it is highly recommended to try it out. Do not worry that this will make you look like an amateur, many professional software developers also use such tools to increase their programming productivity.

8.4 Your Roadmap to Learning a New Programming Language

When you learn a new programming language ...

- First obtain the necessary tools, especially the required compiler or interpreter and a tool (code editor or IDE) to work with your code.
- If you already use a code editor or IDE for another language, consider using it for the new language as well as you are already familiar with its operation. There might be an extension package for your tool that makes it fit for the new language as well.
- If you do not yet have a code editor or IDE in use, try out different ones. If the work with your programming language requires special features, such as for the design of graphical interfaces, or if you want to work intensively with tools for debugging, you should prefer an IDE to a pure (code) editor. Otherwise, a good code editor, from which you are also be able to call compilers or interpreters, may be sufficient.
- When you install a new code editing tool, don't try to understand all the complexity of features right away. You only need a relatively small part of it anyway—at least in the beginning—to really be able to work. Focus first on understanding how to open, edit, and save code files, and how to run your program. After that, you can start programming right away. You will learn most of the rest about your integrated development environment or code editor en passant.
- Before you really get into the programming language, find out what official help and information channels are available; in particular, set browser bookmarks for the official function and language references.
- Check, if ChatGPT supports your language and try out a few things.



What Do I Have to Do to Get a Program Running?

Contents

- 9.1 All Beginnings Are Easy – 66
- 9.2 Hello, World! – 68
- 9.3 Your Roadmap to Learning a New Programming Language – 69

Overview

After you've acquired the necessary tools and explored sources of help and information, it's time to work out exactly how you're going to make a program work. Nothing is more frustrating than being excellently prepared with extensive theoretical knowledge about your programming language, but ultimately failing to even start your first small program.

To avoid this unpleasant experience, you should write your first real program as early as possible, even if you know almost nothing about your programming language.

In this chapter you will learn the following:

- Why it is important to produce a first executable program early on, and thus celebrate your first small success as a programmer in the new programming language
- What the famous “hello world” program is, and why it is a good first step into the new programming language
- What the difference is between interactive and script/batch mode when executing code and when which mode is preferable.

9

9.1 All Beginnings Are Easy

In the following chapters we will deal with many basic concepts whose implementation in your new programming language you need to understand to be able to develop really good, useful programs. Nevertheless, it is helpful to start with a very simple program right now.

You use this simple piece of code as the basis for step-by-step extensions that let you try out every additional element of the language in a practical way. You will also immediately learn some specifics of your new programming language that can cause you many headaches if you don't understand them and pay attention to them from the beginning. A good example of this is the need to end statements with a special character, often a semicolon, as required by a syntactically correct statement in many programming languages. Such subtleties are best learned using a program that is as simple as possible, because the simpler the program, the fewer alternative sources of error there are, and consequently the easier it is to troubleshoot. This is important, because you don't want to start your first steps in the new programming language with a long, frustrating error search. It is always better to have a first small feeling of success! At this point, it is not at all important that you understand absolutely everything you do with your first program. The only important thing is that you can write a small, but executable program and run it.

“Execute” is the keyword, which immediately brings us to another argument for immediately getting started with a small program. By doing so, you will learn how to handle the most important functions of the new tools, in particular how to compile source code (if this is necessary) and how to execute it. If you are working with an IDE, then the relevant commands will probably be accessible via menus or tool-

bars. However, you may decide to work with a combination of a normal code editor and a command line interpreter/compiler. In this case, you need to understand exactly how to call the command-line program that compiles or directly interprets your source code, and what arguments to pass to it. This is true even if you can include the compiler or interpreter in your code editor because this is regularly done by telling the code editor the command line statement in question. Information on this can usually be found in the official documentation of the programming language.

In the case of compiled languages, another step is regularly added to the compilation of the program, namely *linking*. A special tool, the *linker*, produces the executable file of your program. This consists of your program (the source code you wrote, translated into machine language by the language's compiler), compiled program libraries and other compiled program components that you access from within your program. Linking combines all of these components into a single executable file that can then run on its own. If you are working with an IDE, you will often find not only separate menu commands for compiling and linking, but also a command called "Build". This does the compiling and linking (the entire generation of the executable program file from your source code) with a single click.

In some cases, however, it is not done with a few clicks on the interface of your IDE or the call of a command line interpreter or compiler. One example is JavaScript, which you regularly have to embed in a web page, i.e., in an HTML document. This also needs to be learned. A first try with a small example program can't hurt.

However, this does not exhaust the possibilities for executing source code. Some programming languages support an *interactive mode* in addition to the execution of entire programs. In interactive mode, you first enter a program statement. Then, by pressing the <ENTER> key, or a special button, this one statement is executed, and the result is displayed immediately (if there is a displayable result). After that, you can enter another statement and again get its result presented immediately. This procedure, which is also called a read-eval-print-loop (REPL), is interesting if, for example, you want to take an exploratory look at a dataset and perform statistical analyses on it. No wonder that R and Python, two languages that have considerable strengths in this area, have an interactive mode. However, interactive mode can also be used for debugging by feeding the interactive interpreter, often called a *console*, instructions one by one and then looking at the result each time before executing the next instruction. In this way, it is easy to determine exactly where the program that you pass to the interpreter for step-by-step execution behaves differently than you expect. For this reason, it is not only languages used for statistical analysis of data that support an interactive mode.

An interactive mode regularly exists only for interpreted languages. If your language requires a compiler, each interactive instruction you enter would have to be translated by the compiler into a separate, self-contained program in machine language. But then you would not be able to use the next instruction to access data that you have previously processed in memory, because after the first program has finished, its memory area is released again by the operating system. You would not achieve true interactivity this way. Compiled languages therefore normally only

offer the execution of whole programs. This way of executing programs, which also exists in interpreted languages, and which is the more common way of execution, is also called *batch* or *script* mode in distinction to interactive mode.

So, familiarize yourself with exactly how to run programs in the language you want to learn.

9.2 Hello, World!

By writing a minimal program, you will learn everything you need to try out the programming principles you will learn later, and their implementation in your programming language, and to execute programs at will.

But what does such a minimal program look like? The best-known minimal program, which probably every programmer knows, is the “Hello, world” program. This is a program that does nothing more than display the simple phrase “Hello, world” on the screen. The origins of this classic go back to the early 1970s.

The following are some examples of what a hello world program looks like in different programming languages. First in C:

9

```
#include <stdio.h>

main()
{
    printf("Hello, world!");
}
```

As you can see, the actual output statement, the **printf()** function, is surrounded by curly braces preceded by a **main()**. If you study C more closely, you will learn that a C program is ultimately a function itself, namely the **main()** function, which is called automatically when the program is started. In our example, this function does nothing more than call another function, **printf()**, which then displays the phrase “Hello world!” on the screen. Without the **main()** function, your C program will not run. You’ll also run into a problem if you forget the trailing semicolon after the **printf()** statement. And in order for **printf()** to be callable at all, you have to include a standard library of functions for input and output with **#include <stdio.h>**. So, getting a C program to work is not all a pleasure. Importantly, the hello-world program teaches you to apply some basic rules, even if you may not fully understand their meaning at this point. Nevertheless, you would now be able to extend your hello-world program if you went deeper into the C language.

Considerably simpler in structure is the hello-world program in Ruby on Rails:

```
print 'Hello, World!'
```

In Delphi, where you develop a graphical user interface, you learn, in the course of the hello-world program, to display a message box with an information icon and Okay button on the screen:

```
MessageDlg('Hello World!', mtInformation, [mbOK], 0);
```

In practice, you would have to make the display of the message box dependent on some user action, such as the user clicking a button. So, your hello-world program would also include a minimal graphical user interface, consisting of a window with a button. In this way, you would immediately learn to use the basic tools for creating graphical user interfaces.

Finally, in PHP, it makes sense to learn how to embed a PHP script into a website using the `<?php>` tag. Here a hello-world program could then look like this:

```
<html>
  <body>
    <?php echo 'Hello world!'; ? >
  </body>
</html>
```

9.3 Your Roadmap to Learning a New Programming Language

If you are learning a new programming language...

- start by writing a very simple program, even if you have not yet jumped into the depths of the new programming language. This way, you will learn some important specifics of the language. A good way to do this is to implement the classic “Hello, World” example, which can be found on the internet for practically all languages.
- deal with the question of what possibilities there are to execute your programs (compilation, interpreter in script/batch mode, or interactive mode) and exactly how to accomplish this. It is nice to be able to write programs, but it is only useful when you can execute them!



How Do I Make Sure That I (and Others) Still Understand My Program Later?

Contents

- 10.1 Understandable Program Code – 72**
- 10.2 Design of the Program Code and Naming of Program Elements – 73**
- 10.3 Comments – 76**
 - 10.3.1 Explain Your Own Program Code – 76
 - 10.3.2 What Else Comments Are Useful For – 77
 - 10.3.3 Documentation Outside the Program Code – 78
- 10.4 Your Roadmap to Learning a New Programming Language – 80**

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

(User *johnc* on *Stack Overflow* “What is the best comment in source code you have ever encountered?”)

- » There are only two hard things in Computer Science: cache invalidation and naming things.

(Phil Karlton)

Overview

That you should program in a way that allows you to understand later exactly how you proceeded and how the solutions you developed work, is a trivial demand that should be obvious. But, in fact, many programmers, even professional ones, do far too little to put this into practice. So next we’ll look at how you can design your code so that it is understandable to you and others who need to work with it later.

In this chapter you will learn:

- How to make program code readable
- What comments are and why they are useful
- How to use comments cleverly to explain your program code
- What role documentation of the program code plays if other programmers are to use it for their own developments.

10.1 Understandable Program Code

Developing understandable program code is tedious and exhausting, and what it takes to do it is one of the least favorite tasks of many programmers.

For one thing, it takes more effort to program in a comprehensible way, because you have to stick to conventions and write notes, *comments* in programmer jargon, so that later on you can quickly understand how the program you wrote works. When you are mentally deep in the details of a problem solution, most people—including the author of this book—tend to massively overestimate their ability to understand their own actions later on.

But you should keep in mind, and this may be an additional motivation to invest the necessary work, that this “later” does not necessarily have to be in five years. It can also be the day after tomorrow that the lack of readability of your program code becomes your undoing and causes unnecessary effort that could have been avoided with a little more effort when originally writing the code.

So, it makes sense to start looking at what you can do to counteract this early on, and to make sure that you work in such a way from the start that you can always get back into your program code without too much difficulty. Three factors can contribute significantly to this:

- The *visual design of the program code* has an influence on how quickly you can orientate yourself in it and recognize connections between different parts.
- The way various elements of a program are *named* (such as variables and functions, which we'll discuss in detail later) goes a long way toward determining how quickly you can grasp the content of what certain parts of the code do.
- Comments, such as explanations written by the programmer and placed directly in the program code, can, if they are well written, explain a lot of things that cannot be read out of the code itself.

10.2 Design of the Program Code and Naming of Program Elements

One important factor that will help ensure that you still easily understand your program code later is how you format it, especially how you use *line indentation*. An example:

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

Even without understanding what this code, written in the C programming language, does you will see a whole series of curly brackets that open after certain statements and close again sometime later. Everything in between is a code block that belongs to the preceding statement. The code blocks here are obviously nested within each other, meaning that there are code blocks that in turn contain other code blocks. In the above notation, however, this nested structure is difficult to see. This is especially obvious at the end, where a whole series of blocks is closed in a cascade. Which bracket now belongs to which block, and therefore which statement is in which block, is not easy to find out. But this question is essential if you want to understand what the program does and how it works exactly. (For those interested in what this code does: for a matrix with 100 rows and 100 columns, it writes all the field coordinates to the screen in the form (row, column), but omits the odd rows.)

The code becomes clearer if you insert indentations, for example with the tab key:

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

```
    }  
    printf("\n");  
}
```

Now it is much easier to see where a code block starts and where it ends. The code becomes more *readable*.

Most programming languages allow you to format code arbitrarily using indentation. However, there are some exceptions, such as Python, where indentations have a content-related meaning and therefore cannot be set arbitrarily. However, the all-clear can be given for Python as well: The way Python requires indentations virtually forces the programmer to make his code readable anyway. Or to put it another way: a correct syntax (i.e., executable program code) in Python automatically brings with it a minimum level of readability. We will take a closer look at this in the third part of the book, when we learn the fundamentals of Python.

But not only the formatting of the source code plays a big role for the readability and comprehensibility of the code. When programming, you have to give many things a name, for example variables or functions. The user of your program will not see these names (also called *identifiers* in programmer jargon), but that is not a justification to give you free rein to your wildest fantasies when choosing the identifiers.

As an example, consider the following program written in Python that calculates body mass index (BMI):

10

```
a = input("Your weight [in kg]: ")  
b = input("Your height [in m]: ")  
c = float(a)/float(b)**2  
print("Your body mass index is: ", c, "\n")  
print("Your input was:")  
print("Size:", a, "m")  
print("Weight:", b, "kg")
```

Without knowing Python, do you notice anything? The program first asks you for your weight and height. If you enter a height of 1.78 m and a weight of 80 kg, for example, the body mass index is output as 25.25 (a value that indicates slight overweight). Afterwards, the initial values with which you fed the program will be displayed again; it will look like this:

- Your input was:
- Size: 80 m
- Weight: 1.78 kg

With these values you would be clearly underweight, but with a height of 80 meters this would be a comparatively insignificant problem! What happened? This is not a program bug, but a programmer error! In the program, the variable names were swapped when outputting the weight and size parameters. As a result, the wrong

values are shown on the screen. Since the variables are called **a** and **b** here, such mix-ups in the program code are not easily to spot at first.

It is much easier to detect the error if you choose *speaking* variable names:

```
weight = input("your weight [in kg]: ")
height = input("Your height [in m]: ")
bmi = float(weight)/float(size)**2
print("Your body mass index is: ", bmi, "\n")
print("Your input was:")
print("Size:", height, "m")
print("Weight:", weight, "kg")
```

So, use easy-to-understand, meaningful identifiers that convey an idea of what the thing you are naming contains (in the case of a variable) or what it does (in the case of a function). When you're programming quickly, it's tempting to use short, often one-letter variable names (for example, **x**, **y**, **i**, **f**) without much thought, because they're easy to type quickly. But you should resist this temptation. Your "later self" who tries to read your program code will thank you!

Sometimes, however, it is not enough to use just one word for the identifiers, such as **size** or **weight**. It is with compound identifiers that we see differences in the naming conventions of programming languages. Suppose we wanted to distinguish height and weight according to whether the user of the program is a man or a woman and introduced different variables for both sexes to do so. Some programming languages prefer the notation **heightMan** for the height of a man, others **height_man**, still others **height.man**.

Of course, in the end it doesn't matter at all whether you follow the convention that most programmers follow in your programming language or not. For the syntactic correctness and thus the executability of your program, this is of course irrelevant, at least as long as you use only the characters permitted in variable names (in some languages, for example, **\$** is a reserved character; **exchangerate\$** would then be an invalid variable name and would lead to an error). However, it is recommended to stick to *some* standard. This saves thinking, because you don't have to rethink every time, and makes your program more readable later.

On the question of formatting code, and choosing appropriate identifiers many people have done a lot of smart thinking. *Google* has compiled these thoughts for a whole range of programming languages in its *Google Style Guides*, and you will also find style guides (sometimes slightly deviating from *Google's* guidelines) in many other places on the internet. Feel free to look at such a style guide for your programming language. Of course, you don't have to follow it slavishly. More important than following a certain style is to have a style in the first place, which you can implement more or less consistently, in order to save yourself mental work and to keep your code readable and thus understandable.

10.3 Comments

10.3.1 Explain Your Own Program Code

It is not only good formatting and a sensible choice of identifiers that help you to write code that you will understand later. Another important tool on this path are *comments*. Comments are texts in the program code that are *ignored by* the compiler or interpreter. Unlike in the rest of the source code, where you have to stick scrupulously to the syntax rules so that the computer understands what you want, in comments you can write to your heart's content. The trick is to make it clear to the compiler or interpreter that your comment is not part of the actual program code, and that it can therefore safely disregard it.

In order for the compiler or interpreter to know what is program code and what is comment, comments are always introduced with a special symbol. Depending on the programming language, the comment then extends either to the end of the line or to the point where it is again terminated with a special symbol.

Some languages only support the first mode, which means that there are only single-line comments. If a comment is to extend over several lines, the comment must again start with the comment symbol in each line.

Let's look at some examples. First, an example in the C programming language:

10

```
// Counter reading has now run up completely.  
// Output counter reading.  
printf("Counter reading: ", counter, "\n");  
counter = 0; // Reinitialize counter
```

The symbol that introduces comments here is `//`. Everything between `//` and the end of the line is a comment and is ignored by the C compiler. Therefore, comment and code can also be on the same line, although the code must of course be to the left of the comment symbol `//`. Interestingly, the C programming language and many languages based on it know not only the single-line comment with `//`, but also multiline comments. These are enclosed between the symbols `/*` and `*/` (Note, the asterisks always face the comment text). This would allow the example above to be written as follows:

```
/* Counter reading has now run up completely.  
Output counter reading */  
printf("Counter reading: ", counter, "\n");  
counter = 0; /* Reinitialize counter */
```

As you can see, the second line of the first comment does not need a separate symbol to start the comment, because we are still in the comment area that was opened by `/*` in the previous line. Only with `*/` is the comment closed again. Since this comment has a clearly defined beginning and end, it can also be placed in the middle of the code, although this is not recommended since it is detrimental to the readability of the code:

```
counter /* the loop counter */ = 0;
```

Of course, every programming language has its own symbols for comments. However, the aforementioned `//`, the combination of opening `/*` and closing `*/`, and the `#` symbol (usually for single-line comments) are frequently encountered.

Using comments can help you document how your program works directly in the code. Everyone who has the code sees the documentation at the same time. Of course, this is especially interesting if you are working with someone else. Your collaborator will certainly appreciate a hint or two in particularly complicated places.

But you yourself, at least your “later self”, will also benefit immensely from the comments. Nothing is nicer than looking at an old piece of code and being happily surprised to find that the developer at the time (yourself) left a little help that makes it considerably easier to understand the difficult passage.

However, because commenting your solution takes time, you should only use comments where it is really necessary, that is, at points in the program where you can expect that you will not easily understand them later. So don’t comment every triviality but use comments economically. Commenting is probably the least popular part of programming work. So, write your comments promptly, ideally right after you have written the code in question. This way, you will not only remember exactly how your solution works, but you will also avoid having to think about the solution again later to write the comment, a mental effort that may discourage not only die-hard fans of procrastination from adding comments. Artificial intelligence tools such as ChatGPT can help here. Ask ChatGPT to comment your code once, for example with the following prompt: “Comment the following code so that it is understandable for beginners”. You will be surprised how much tedious work ChatGPT, Gemini, Claude & Co can do for you.

10.3.2 What Else Comments Are Useful For

With comments, you can not only document the procedure you used, but also, for example, note unfinished tasks. A clear **TODO** marks the comment as an open task for you. In the same way, you can use **REVIEW** to highlight code segments that you want to take a closer look at or assign any other meaningful tags that mark specific types of comments.

You can also use comments in a much more mundane way, namely to better structure your source code and make it clearer for you. In this sense, you can use comments to visually separate parts of the source code from each other, for example like this (in C++):

```
/* ----- START READING IN THE DATA ----- */
```

When you start programming, you will see how helpful it can sometimes be to “comment out” a program statement. This means to put the instruction into a

comment and thus “switch it off”. After all, the compiler or interpreter does not execute what is written in a comment. So, by enclosing a statement in a comment, you can make that statement temporarily ineffective. A small example in C:

```
printf("This text is printed on the screen!");  
/* printf("This text does not appear because the printf statement is  
   commented out"); */
```

Comments are very helpful when you are just starting to deal with a new programming language and want to record the knowledge you gain from trying it out in the code of the test programs right away. It is well known that you learn best when you write down what you have learned (the acknowledged most effective method for this, handwritten notes, is naturally ruled out for program code).

10.3.3 Documentation Outside the Program Code

Comments are a kind of documentation *in the* program code. This is advantageous, since the explanations and what is explained are right next to each other. However, there is nothing against storing documentation *outside* the source code. This is useful, for example, if you want to write more comprehensive explanations, or if you want to include other information in the text (such as a table or a photograph of a handwritten note or sketch). The same is true if you want to document overarching contexts, such as the overall structure of your program or the interrelationship of different parts of the program. In all these cases it is okay to separate at least part of the documentation from the code and to place it to another document. Whether you write this external documentation with a word processor or record it in your electronic notebook is completely irrelevant. The principle applies: Praise be to that which helps understanding!

A special way to generate documentation outside of the source code are special *documentation generators*, which are available for many programming languages. In their basic function, these tools do little more than automatically read out certain comments specially marked in the program code and prepare them in a visually appealing form. The result is then usually output as an HTML document or in PDF format. Examples of such tools are *Javadoc* for Java, *phpDocumentor* for PHP, *roxygen* for R or *Doxygen*, which supports several different languages, including Python and C++. Generating documentation in this way is particularly interesting when writing program code that will later be called by others (we’ll see below how to package parts of programs into modules that can then be called from elsewhere). While those who will call the program code do not necessarily need to understand in detail how it works, they do need to know how exactly to call the program, and how to control its operation for their purposes. Since these developers are not interested in the program code as such, but only in its *interface*, it would be impractical if they had to rummage through the code to find the relevant hints. That’s why the documentation is placed outside the code. This way, it stays “clean” and concise, and is easy for the user to use. For the developer of the code, it is also

10.4 Your Roadmap to Learning a New Programming Language

easy to write because it is embedded directly in the program code, just like conventional comments. In Java, for example, this could look like this:

```
/**
 * Reads the name of the user
 *
 * @author Marc McIntosh
 * @version 1.3
 *
 * @param prompt Text of the prompt shown to the user
 * @return Name the user has entered
 */

public String getUser_name(prompt) {
    /* Here comes the actual program code */
}
```

Here, a special part of the program, called a function, is documented using *Java-doc*. The function can be called with a prompt, for example `getUser_name("Enter your name: ")`, then reads a user input and returns the user's input as a value to the person who called the function. (We'll deal with functions in more detail later.) The important thing here is that this function is a part of a module that can be called by other programs. To understand what the function needs to be called (the text of the prompt to be displayed to the user) and what it returns (the name the user has entered), the function is documented here in such a way that the documentation tool Javadoc can automatically generate appealing HTML documentation. Javadoc uses special comments for this purpose.

Normal comments in Java are placed between the symbols `/*` and `*/`. However, if a comment is prefixed with `/**`, Javadoc knows that this is a comment that should become part of the documentation. Special predefined tags prefixed with `@` can be used to address specific fields in the documentation. For example, `@param prompt` is used to explain what the **prompt** parameter means when calling the **getUser_name** function. These predefined fields can then be presented in a special way in the documentation, for example in a special formatting or at a special position within the documentation.

Often a stricter distinction is made between documentation and commenting than we do here. Based on the target audience, comments are aimed at the person who wants to understand and edit your code (usually yourself, but possibly others as well). Another developer, however, who simply wants to use your code in his own programs, is not interested in its inner structure and workings, and will not read your comments in the code at all but will only look for information on how exactly to use your code properly. Similarly, the user of a word processor is not interested in exactly how it works inside, he wants to know how to change the font color for a selected text! The provision of this information about the use of the code is done by the documentation in the narrower sense, which of course may be generated via documentation generators from special comments in the code.

10.4 Your Roadmap to Learning a New Programming Language

When you learn a new programming language ...

- to determine whether the programming language allows you to format your code in any way you want (for example, with indentations), or whether the formatting has a meaning in terms of content, take a look at a style guide that gives information on how to format your code in the programming language and how to choose the identifiers/names of variables, functions and other elements,
- decide on a way to format your code and name the variables, functions, and other elements. You don't necessarily have to follow a style guide, but you should remain consistent,
- determine how comments can be built into the code, in particular whether comments can be multiline, and whether they are bounded at the front and back, or always extend to the end of the line,
- get into the habit of commenting your code so that you can understand it later, but do so economically and concentrate on the difficult parts where there is a real risk that you will not understand them later, or only with great difficulty,
- determine whether there are documentation generators that allow you to create attractive HTML or PDF documentation from the comments integrated into the program code.



How Do I Store Data to Work With?

Contents

- 11.1 Variables as Placeholders for Data – 82**
- 11.2 Data Types of Variables – 83**
 - 11.2.1 Different Types of Data Require Different Types of Variables – 83
 - 11.2.2 Important Data Types – 84
 - 11.2.3 Changing the Data Type: Conversion of Variables – 88
- 11.3 Creating and Initializing Variables – 89**
- 11.4 Not So Variable: Constants – 92**
- 11.5 Ordered Fields of Variables/Arrays – 92**
- 11.6 Associative Fields of Variables/Hashes – 96**
- 11.7 Objects – 97**
 - 11.7.1 The World Is Made of Objects – 97
 - 11.7.2 Classes – 98
 - 11.7.3 Inheritance – 100
 - 11.7.4 Methods – 103
 - 11.7.5 Polymorphism – 105
 - 11.7.6 Access Rights – 107
- 11.8 Your Roadmap to Learning a New Programming Language – 109**
- 11.9 Solutions to the Exercises – 109**

Overview

Programs work with data. These are read in from the user or from files, databases or other sources. We will look at exactly how this happens in the following chapter. This chapter is about something else: all data must be stored in the computer's memory so that it can be processed. This is done in the form of variables. But don't worry: even if these variables have a lot in common with their namesakes from mathematics, you don't need to delve into the depths of mathematics to understand how to work safely with variables when programming.

In this chapter you will learn the following:

- What variables are and how to create them
- What types of data variables can hold
- How to combine many variables of the same kind into variable fields (arrays and hashes)
- How to combine different variables reflecting the properties of a certain real-world object (e.g., a car with the properties brand, maximum speed and list price) into one object and how to then manipulate these properties (object-oriented programming paradigm).

11.1 Variables as Placeholders for Data

11

Variables are tools with which we record data in programs. As in mathematics, variables act as *placeholders* that we can address by their name, as with the *identifiers* discussed in ► Chap. 10. As the name suggests, their content is variable, so we can store different data in the variable (one after the other), but no matter what the current content of the variable may be, it is always accessible via their identifier.

A variable is comparable to a box that we label. The label, or the name or identifier, always remains the same, even if we change the contents of the box. The contents of the box should match the label to some extent, otherwise confusion arises. It is exactly the same with variables.

In general, we are quite free in choosing the name of the variable. However, as you have already seen in the previous chapter, there are some language-specific basic rules that must be observed. These rules usually dictate which characters may appear in the name of a variable (certain special characters such as % or # are usually not allowed). In addition, many programming languages also specify which characters may or even must appear at the beginning of a variable name. In R, for example, variable names may not begin with a digit, in PHP variable names *must* begin with the dollar sign (\$). However, as you will remember from the last chapter, it is important not only to follow these “hard” rules, the violation of which will inevitably lead to the compiler or interpreter refusing to obey you, but also to name variables meaningfully and *consistently*. Meaningful in this context means that you can tell the content or purpose of the variable from its name. Consistent means that you ideally always form variable names in the same way, for example, always capitalize and lowercase the parts of compound variable names in the same way.

Let's say you are developing a web-shop and want to store the number of orders a customer placed in the last 12 months in a variable. This variable could be called, for example, **CountOrders12Months**, but also **CntOrd12M** or **cntOrd12M** or **Count_Orders_12_Months** or ... The possibilities are numerous, and there are few limits to creativity. For the readability and comprehensibility of the program code, it is important that you think about a system, how you want to form variable names, and then follow this system as consistently as possible.

11.2 Data Types of Variables

11.2.1 Different Types of Data Require Different Types of Variables

So far, we haven't talked about what *kind of information* a variable can hold, such as a number or a text. For the compiler or interpreter of your programming language, this makes a difference, for at least two reasons.

For one thing, the compiler or interpreter of your language must reserve memory for the variable. It is obvious that a long text (for example, a street name) requires more memory than a number (for example, a house number). If you first assign a number to a variable, let's say the house number 58, your compiler or interpreter then makes sure that enough memory is reserved to store a number. If then you assign a long text to the same variable, for example the address "Times Square, Manhattan, NY", the space initially reserved is no longer sufficient. Additional memory space must be found, possibly at a completely different location in memory.

Secondly, very different operations can be performed with numbers and texts. For example, you can multiply a number by another number. With a text, these operations make no sense. In the worst case, the program will even crash if you perform an operation that is not allowed for the kind of data your variable contains. Therefore, it makes sense to check at times whether the data in a variable is of the "right" kind, has the right *data type*, in programming terms.

The data type of a variable describes what kind of information can be stored in it. In this respect, a data type specifies, for example, whether a variable should hold integers, fractional numbers, or texts. In practice, however, data types differ not only in the type of data they cover. They also differ in terms of *the range of values* or *length of information* they can hold: A variable that is intended for a text and provides 10 characters for it, will store "Peter Miller" as "Peter Mill" (the space counts also as a character here). In an integer variable whose value range is from 0 to 65,535 (that's a value range you can cover with two bytes), you won't be able to store a negative account balance of -254 USD. In the same way, an integer variable with the value range from -32,768 to 32,767 (which also requires two bytes of memory) will not be able to handle an account balance of 50,000 USD. Data types therefore have different value ranges, which – just like the basic type of information for which they are intended – limits the data they can hold. In the case of data types for floating-point numbers, such as fractional decimal numbers like 3.1415926,

another characteristic comes into play in the form of *precision*, or the number of decimal places. Accuracy matters. Whether you receive the gold medal at a skiing or speed skating world championship together with a competitor, or come in second behind this competitor, may depend on whether the result is measured in hundredths of a second (and both athletes have the same result and would therefore both be declared winners) or whether thousandths of a second are also taken into account, and a small difference between the two times then becomes apparent.

11.2.2 Important Data Types

The basic data types are quite similar in most programming languages. Usually there are data types for:

■ Integers

These data types take integers such as -4 , -3 , -2 , 0 , 1 , 2 , 3 , 4 . Optionally, there are data types that take only positive integers (including zero), that is, natural numbers. Integer data types in many programming languages have a name that includes the word *integer*, or an abbreviation of it. Classical names of such data types are **integer** or **int**. For integer data types with a particularly large value range, i.e., the possibility of storing very large numbers, they are also often called **bigint**, **long int** or simply **long**.

■ Floating-Point Values

Floating point data types take fractional decimal numbers, such as 1.7 or 3.141459 . Of course, integers can also be represented as floating-point numbers, e.g., 4 as 4.0 . Accordingly, all integers can also be stored in floating-point variables.

If this is so, the question naturally arises why one needs integer variables at all and does not simply always work with floating-point variables. The reason is mainly the larger memory requirement of floating-point values, because they must store the fractional part of the decimal point as well as the digits before the decimal point, or they have to keep capacity for its storage, even if the fractional part after the decimal point is always zero, as is the case for integer numbers. The problem is that our compiler or interpreter does not know in advance that we only want to store integer values. Therefore, it always reserves as much memory as it needs to store a fractional decimal number. If you need a lot of such floating-point variables at once, this will result in a noticeably higher memory requirement.

There are a number of names for floating-point data types in different programming languages, such as **real** (for real numbers) or **float**. In most languages there is – analogous to **integer** and **long integer** – another floating-point data type, which offers higher precision (i.e., more decimal places) and a larger range of values. In most cases, names such as **double** or even **long double** already indicate the higher precision.

■ Characters and Strings

Individual characters, such as letters, are not dissimilar to integers, because every character that the computer understands, i.e., that is part of its *character set*, can also be encoded as a number. Well-known character sets are ASCII or Unicode. Although there is a close connection between characters and numbers, most programming languages have a special data type for individual characters, often with a name based on the word *character*. Accordingly, many languages have a character type **char** for individual characters.

Entire texts are ultimately sequences of characters, or *strings*. Some programming languages therefore do not know a separate type for character strings but build character strings from a string of individual character variables. Here, a more complex data type is created from a simple data type, in which many variables of the simple data type are practically connected in series. This kind of series connection is called an *array*, a construction we will deal with more intensively later.

Other languages do have a special data type for strings, which is often simply called **string**. This somewhat shields the programmer from the nature of the string, namely that it consists of different characters strung together. For him, it then “feels” as if the text is a single variable and not composed of individual variables arranged in a field.

However, strings can often contain not only letters, digits and special characters (such as punctuation marks, #, < , > , * , ~), but also so-called escape sequences. These are special control instructions. In practice, the most important one marks a line break and is represented as **\n**. The backslash (\) tells the interpreter or compiler that the following character is not a letter but a control statement; the **n** itself stands for *new line*. Thus, the string **"Mr./Mrs.\nFirstName Name\nStreet\nZIP City"** would be interpreted as

```
Mr/Mrs
FirstName Name
Street
ZIP city
```

Wherever the escape sequence **\n** is encountered, a line break is inserted. While the escape sequence marks the position of the line break within the string, certain program functions – such as those for outputting strings to the screen – “understand” the encoding and implement it accordingly. Besides **\n**, there are several other escape sequences, for example **\t** for a tabulator jump. Escape sequences solve another common problem: In most programming languages, strings are enclosed by single (') or double (") quotes. But what if a string is supposed to contain a quote, for example, so how do the quotes need to be *real characters* within the string? Consider the following string:

```
"He said, "I love you!""
```

It is enclosed in double quotes, but contains a quote that itself uses double quotes. Normally, the interpreter or compiler would refuse to follow us here. It would first recognize a string **"He said: "**. However, this is followed (without this being understood as a string) by **I love you!** followed by an empty string (**"**). The **I love you!** would probably cause problems, because it will not be a valid statement in the respective programming language, so outside of a string the interpreter or compiler would try to interpret it as a statement. So what to do? Apart from the trivial solution of simply using single quotes for the quote inside the string, the “inner” double quotes can also simply be *escaped*, as follows:

```
"He said: \"I love you!\""
```

The backslashes tell the interpreter or compiler that the following quotation mark should be understood as *part of the string*, not as its delimiter.

But what if there should be a backslash in the string? For example, in this string:

```
"\n is an escape sequence."
```

The escape sequence `\n` would normally lead to an undesired line break within the string. Masking by the backslash helps here too, only this time the existing backslash itself is masked, because it should be understood as part of the text and not as the start of a control statement. So:

```
"\\n is an escape sequence."
```

This string would result in the correct output. Escaping the backslash itself is especially important when you code paths on Windows systems, for example `C:\\Home\\My source codes`. A common cause of errors is that escaping is forgotten in strings like this.

Escape sequences are used in many programming languages, for example in C, Python, Perl or R.

■ Truth/Logical Values

Practically all programming languages have a special data type for the truth content of statements, i.e., *true* or *false*. In this context, we often speak of *boolean* (after the nineteenth century English mathematician and logician *George Boole*) or *logical* variables. Unlike the data types we have considered so far, boolean variables allow the storage of only two different values, true and false. This is completely different from, say, integer variables, which can hold *any* integer. In most programming languages, the two truth values have special identifiers, usually **true** and **false**, so that boolean variables can be easily assigned values or compared, and these assignments and comparisons are easily readable in the program code. Because there are the easy-to-understand identifiers **true** and **false**, programming languages

usually store only 1 and 0 as values of boolean variables. In this sense, boolean variables are mostly just integer variables, where the compiler or interpreter takes care that they have only one of two possible values, which can conveniently be accessed with special identifiers, mostly just **true** and **false**.

In programming language, boolean data types are often called **bool**, **boolean**, or **logical**.

■ Other Data Types

These data types can be found in practically all modern high-level languages, albeit with different names, different value ranges, and different precision. In addition, most programming languages have several other, more complex data types, which are often based on the simple types you have just learned about.

Some languages, for example, have (at least) one special data type for the date or the time. If you want to store the date and time, there seems to be no way around simply storing days, months, year, hours, minutes, and seconds as integers and then “assembling” them into some complex data type. This is indeed one possible solution that is often used. But there are others. For example, time on systems using the UNIX operating system, which dates to the 1970s, is measured using only a single integer representing the number of seconds that have passed since 0:00 a.m. GMT on January 01, 1970. Then, for example, the turn of the millennium, i.e., January 01, 2000, at 0:00 a.m., in New York City corresponds to the Unix time 946,702,800. This time concept is also called *UNIX Epoch*. So, the date can also be represented in a single number, but in many programming languages it is treated as a separate data type, although it is ultimately an integer.

This example also nicely illustrates the role of the value range in data types. Systems running on UNIX did not have a year 2000 problem. However, they will run into trouble for this on Tuesday, January 19, 2038 at 3:14:08 UTC. This is because the data type used by UNIX (at least on the older systems) to store the time will then reach the value 2,147,483,647 and thus the limit of its value range. This data type cannot store larger numbers. So, what happens one second later, at 3:14:09? The date value measured in seconds will jump back to its smallest possible value, -2,147,483,648. According to Unix chronology, this corresponds to a time in December 1913. The developers of Unix, of course, knew this, but accepted it; the year 2038 was still far away from the perspective of the 1970s, and data types with a larger value range were not available.

But the example shows very well that you have to think about whether the data type you intend to use is really sufficient for the intended purpose, or whether you need a data type with a larger range of values (if you have one available, unlike the UNIX developers).

Date/time is an example of a data type that exists in many languages alongside the other data types discussed above. In addition, there are often other data types, such as for enumerations. In variables of these types, categorical data can be stored, i.e., data that can only have certain distinct values, for example the sex of a person, their highest school-leaving qualification, or a car color. In this sense, they are like boolean variables, except that the number of possible values may well be more than

two. Often, such *enumerations* (*sets* or *factors*) are special data types in a programming language even though “under the hood” they are essentially stored as integers (each category/value is represented by a specific number).

Date values, strings, and enumerations are all examples of data types that are based on simpler data types. When we discussed strings which can be thought of as consisting of a sequence of characters you already became acquainted with the field as a way of combining variables into a more complex data type. Besides fields, we will later take a closer look at another possibility, the core idea of which is to combine variables of *different* types into one object. Such an object then often represents a real object with its central properties, for example a car, which can be described by its age in years (integer), its brand (string), and its color (enumeration), among other things.

11.2.3 Changing the Data Type: Conversion of Variables

Sometimes you need to change the data type of variables. This is called *converting* the variable. For example, suppose you have written the software for an online book store. A user of your program has just entered the information about how many of which book he wants to buy (usually just one, but sometimes maybe two, one for himself, one as a gift). On the check-out page, you want to show the user how many books they are ordering in total. This is easy if you have read the quantities into integer variables. However, it becomes problematic if you have used string variables. Your addition will not succeed. Why is the difference between the two variables so important?

Data types, as you have seen in the previous sections, are defined by the kind of information the range of values that variables of this type can hold, and, if applicable, also by the precision with which (floating-point) values are stored. But there is another characteristic that distinguishes data types: The question of what kinds of operations can be performed on variables of a data type. That you can add integers is obvious. But what about strings? Can you add the two strings “**apples**” and “**pears**” in the mathematical sense? No, of course not. But what about the strings “**2**” and “**1**”? These could be the quantities of two books from our online shop example. Can these two strings be added together? The answer is, possibly to your surprise: No.

The compiler or interpreter that processes your program code is simply not interested in what exactly is contained in the variable. It is simply a string of characters, completely without meaning for the computer. Whether the string contains letters, numbers, or any special characters like the dollar sign or the underscore is irrelevant. The compiler or interpreter is not interested in the content. Therefore, addition is not an operation that is allowed for strings. But if you now know that the string variables you read in via your online store’s website definitely contain numbers, of course you still want to calculate with them. What to do?

The key to solving this is to change the type of the variable. Many programming languages provide special statements that allow you to do just that and perform an *explicit type conversion*. Some languages, however, also have *implicit type*

conversion. In a certain sense they are not as ignorant as described before, but if you want to calculate for example `"2" + 1` (where `"2"` is a string value), they look at the variables and their content quite closely and would recognize that in our example the calculation would be possible if the string `"2"` was converted to a number. The conversion is then performed automatically, without you having to intervene with a special statement.

Another example of this is the following calculation: `TRUE - 1`: Here, an integer value is subtracted from a Boolean value. Languages that follow a very strict type concept would reject this calculation with an error message. Languages that support implicit conversion would recognize that `TRUE` is ultimately (in most languages) represented by the value `1`, because it is this that is stored internally as the truth value. Thus, the value of `TRUE - 1` can be determined. In this sense it is even true: `TRUE - 1 = 0 = FALSE`.

Languages that look very strictly at the adherence to the data types, convert little implicitly and possibly even allow little explicit conversion, are called *strongly typed*. Here the data types of variables play a big role. At the other end of the spectrum are languages where the programmer does not have to specify the type of the variables, but where the data type always adapts automatically by implicit conversion in such a way that the desired operations can be performed as far as possible. Such languages are “weaker typed”. In extreme cases, even operations that make no sense at all can be performed without an error message: The addition `3 + "My name"` (where `3` is a number, `"My name"` is a string) would then perhaps simply result in `3`. The type conversion from `"My name"` to a number fails, of course, but the programming language is so weakly typed that it just “keeps on calculating” as best it can.

What may sound tempting is at the same time dangerous. Because obviously it would be bad if the user of our online shop enters the quantity `3` for one book and the quantity `"Tom Peterson"` for the other book. We can't really work with this information. In the worst case, our program gets into trouble and produces implausible results or even crashes completely. A little more “control” over whether the variables are good for the operations you want to perform should not be seen as a restriction on your own programming freedom. It is simply an aid to writing safer, more stable program code and to detecting errors at an early stage (ideally during development and testing, and not at runtime).

11.3 Creating and Initializing Variables

In the last section, you saw that variables can be distinguished according to their data type, i.e., according to what kind of information they can store and what range of values can be covered with them. Now the question arises - how do you create a variable in order to be able to work with it. The “birth” of a variable is done differently in different programming languages. Roughly, however, two types of languages can be distinguished: Those in which you have to create a variable explicitly before using it for the first time, and those that create the variable automatically as soon as you use it for the first time.

Languages of the former kind include C, Visual Basic for Applications (VBA), and JavaScript. Suppose we wanted to assign the value **10** to an integer variable called **piececount** in both languages. Before that happens, however, the variable must be created. Programmers refer to this as *declaring*, which is the process of telling the compiler or interpreter that you want to use this variable from now on. The compiler or interpreter then takes over the technical part of variable creation.

The declaration of the variable including the assignment of the value **10** would look like this in C:

```
int piececount;
piececount = 10;
```

int piececount not only declares a new variable **piececount**, but also specifies its type as **int**, which in C is the integer data type. After this declaration, the compiler knows that there is an integer variable called **piececount**, and it can be used in the program from now on. Without the declaration, the assignment **piececount = 10** would lead to an error message, with which the compiler points out that it does not know the variable **piececount** and thus cannot assign a value to it.

The same code section in VBA would look like this:

```
Dim piececount As Integer
piececount = 10
```

11

Unlike in C (where the declaration was introduced with the type of the variable), a special keyword is used here to declare the variable, namely **Dim**, from *to dimension*. And that's actually very good, because what happens when the variable is created is that memory is reserved, as much as the data type of the variable requires. In this sense, the memory is *dimensioned* according to need.

Some languages like JavaScript require a declaration of variables, but no type is specified:

```
var piececount;
piececount = 10;
```

Here the compiler or interpreter decides based on the use of the variables, which data type it needs. By the assignment **piececount = 10** it becomes clear that this must be an integer variable. If the variable is later assigned a value that requires a different data type, for example by the assignment **piececount = "Not specified"**, then the data type of the variable changes accordingly in the background without you as the programmer noticing anything. Unlike in C or VBA, the typing is *implicit*. In C and VBA, on the other hand, the type must be specified *explicitly*; this is why we also speak of *explicitly typed* programming languages.

You may now ask yourself whether a declaration in JavaScript is really that useful, since the type of the variable is not specified at all. If the type has to be speci-

fied, it can be argued that the compiler or interpreter can check whether a variable is unintentionally assigned values that it cannot accept due to its data type. This additional check provides security and may prevent troublesome errors and strange behavior of the program. After the above declaration, the C compiler will refuse to work with an assignment such as **pieccount = "Not specified"** and abort with an error message. This way, the programmer is made aware of the fact that he has somehow worked inconsistently in the code.

Does the necessity to declare a variable the way it is done in JavaScript have any benefit for the programmer? Or is it simply a chicanery devised by the inventor of the language to drive the users of his invention crazy? As you can imagine, this is not the case. There is a point to forcing the programmer to register his variables, because that way, the compiler or interpreter knows *which variable identifiers* are allowed. If you then make a typo, as happened to the author several times while writing these lines, and for example write the assignment **pieccount = "Not specified"** (note the missing *e*) in your code, the JavaScript interpreter will recognize that you are trying to access a variable that does not exist. This is because the variable you declared has a different name. This way you will quickly get to the root of the problem and be able to fix it. This would be much more difficult if the programming language did not require a declaration. Then the statement **pieccount = 10** would simply create a new variable called **pieccount**. In that case, you'd probably have some trouble determining what's causing your program not to behave the way you want it to. Uncovering the real cause, namely that you are actually working with two different variables, is then considerably more time-consuming than if the compiler or interpreter already gives you a "hint".

So sometimes it can be quite helpful to accept a stricter regime. This strictness makes it easier to detect and locate problems. If your programming language offers the possibility to switch to a stricter mode (which can be achieved in VBA, for example, by a special option that forces variable declaration), you should accept this offer, even if it looks like more control and less freedom at first glance.

The strictness that some programming languages impose on their users sometimes includes that the declaration of variables is only allowed at the beginning of a program, which improves the structure and thus the readability of the code.

In our examples above, we assign a value to the variable directly after its declaration. But what if we do without this, but still use the variable later anyway, for example output its content on the screen? What would be displayed then? In other words, what is the value with which variable is "born"? In the past, variables often had a somewhat random value after their declaration, namely the one that was currently in the memory area released by the operating system, which was then reserved for the variables. This value was ultimately a residue of the program that had previously used the same memory area but had not "cleaned up" after itself. Therefore, a very important recommendation to any programmer was to always load his variables with a value at the beginning, to *initialize* them, or to give them a clearly defined, known content. This prevents the program from crashing or reacting in an unpredictable way due to "strange" variable contents.

Today, most programming languages initialize variables automatically, numeric variables with **0**, strings with an empty string. Some languages even have a special

value for variables not explicitly initialized by the user, such as **undefined** in JavaScript. This value signals that the variable does not yet have a real value because it has not yet been initialized. Many languages also have another value that indicates that the user is deliberately leaving the value of the variable open (think of an unanswered question in a poll, for example). In JavaScript, for example, this value is **null** (not to be confused with number 0), in Delphi/Object Pascal it is **nil**, in R it is **NA** (for *not available*). If the variable has such a value, this indicates that the variable is indeed used, just that it does not contain an explicit value. In other words: No value is also a value!

Even though initialization is strictly speaking no longer necessary in many programming languages today, it is good practice. Often, initialization can be done directly in the declaration, for example in C. Our example above would be shortened to:

```
int piececount = 10;
```

11.4 Not So Variable: Constants

Another type of language elements, which are usually initialized directly at declaration, are *constants*. Constants behave similarly to variables in the sense that they are values that can be addressed in the program under a certain name, their identifier. Unlike variables, however, their value cannot be changed in the further course of the program once it has been set. This protects the value of the constant from being accidentally overwritten. Usually, constants must be initialized with their (henceforth constant) value already at the declaration. An example from Pascal:

```
const pi = 3.14159;
```

In C, the constant declaration looks just like the declaration of a variable, with the addition of the **const** keyword:

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

11.5 Ordered Fields of Variables/Arrays

So far, we have always created single variables. However, most programming languages also support *arrays* of variables. An array is an ordered collection of variables of the same type that can be addressed under the same name. The individual values of the field are accessed via an index.

In the context of our hypothetical online shop example, we could, for example, store the click history, i.e., the sequence of products that the customer has looked

at, in an array. This is important information when it comes to analyzing customer behavior in more detail and making tailored suggestions to the customer as to which products might also interest him, as is the case with many online shops today.

In this example, our field could be called **history**. In this field, we would sequentially store the IDs of the products that the customer has viewed. Let's assume here that these IDs are integer values. Then we have a field of integer variables. We can now use an *index* to access the individual *elements* of the field. We could access the fifth element in the click history this way through **history[5]**. In the square brackets you see the number of the element that is to be accessed, in this case the fifth product that our current customer has looked at.

Of course, we could have implemented the whole thing with single variables: Thus, we could have created variables **history1**, **history2**, **history3**, **history4**, **history5**, **history6**, and stored the sequence of products under consideration in these variables. However, this has some disadvantages. For one thing, in many languages the variables must be explicitly declared. If you envision a click history with, say, 30 products, you would have a lot of typing to do just to declare the variables in the first place. On the other hand, in practically all languages that support fields, there are very efficient mechanisms for traversing those fields, namely by moving the index that identifies a field element one position at a time. This way you can go through the whole field step by step. The whole thing can be solved very elegantly from a programming point of view, so that you do not have to write much code. It would be much more complex and maintenance-intensive (think of the case where you just want to quickly increase the length of the history from 30 to 100 products) if you were to work with individual, independent variables.

Although virtually all modern high-level languages provide fields, the languages differ in one important respect; namely, what the index value of the *first* field element is. In many languages, field indices start at 0. Then **history[0]** would be the product ID of the first product the user looked at.

Fields/arrays can also be multi-dimensional. In our example, we could store the click history of all our visitors in one field; we would use a two-dimensional field, which we can think of as a table or matrix. The rows would contain the users, and the columns would contain the IDs of the products they viewed. Then **history[3][1]** would be the ID that website visitor number 3 looked at first (assuming our field indexing starts at 1). To access elements of a field, we now need two indexes as coordinates that describe exactly where we want to reach in our two-dimensional tableau.

The dimensionality of fields is, of course, by no means limited to two dimensions. We could easily add a third, fourth, fifth dimension. All no problem, as long as we still have in mind which index (and thus which dimension) stands for which "coordinate" we use to store the information in our field. For example, in addition to the user, we could also store the day (from 1 = Monday, to 7 = Sunday) and thus have a third dimension. Our field would have the structure **history[day][user][productid]** and with **history[2][3][1]** we would get the ID of the first product that user 3 looked at on Tuesday.

Without going into it in detail, we have introduced a notation in the past paragraphs to access the individual elements of fields with indexes: We put the index number in square brackets. This is actually how many programming languages do it, but not all: some put the indexes in *round* brackets. However, the possibilities of indexing are not exhausted by simply specifying an index number. Many programming languages allow further methods of indexing, for example indexing by exclusion: Here, not the index or indices of those elements of the field are specified that you want to select, but just the other way around, those that you do not want to select. This is often done by prefixing the index with a minus sign. **history[-5]** would then be the entire field except for the fifth element. In some programming languages, however, a negative value means something else, namely that the field is indexed from the end. In this case, **history[-5]** would be the fifth element *from the end*. Some languages also offer the specification of a whole range of indices: so **history[5:9]** would grab the fifth, sixth, seventh, eighth and ninth element of the field. A notation in the form **history[from:to]** not only saves typing work but makes it easier especially when the selection limits **from** and **to** are not yet known, but variables are used in their place whose values are first determined by the program (for example, by a user input).

The world of fields is relatively colorful across programming languages. However, most of them have the following principles in common:

- Fields are combinations of several single variables (however, there are exceptions to this fundamental definition): In the statistical language R, one-dimensional fields, so-called vectors, are even the standard; a single variable is then only a special case of such a vector, namely a vector of length one).
- The variables in a field all have the same type (for example, they are all strings, or all numbers).
- Fields can be one-dimensional or multidimensional.
- Individual elements of the fields can be accessed via a numerical index by specifying the element to be selected.
- When creating a field (if variables have to be declared at all) its dimensions as well as the type of the field variable have to be specified.

Beyond that, however, working with fields can differ greatly in different programming languages. We had already learned about some of these differences. In summary, among other things, the following questions may well be handled very differently in different programming languages:

- Whether the numeric indices start at 0 or at 1.
- Whether round or square brackets are used in indexing.
- What indexing options are available beyond simply specifying the element number.
- Whether, and if so, which functions are available to work with fields (for example, to determine the length of a field or to delete an element from a field).
- Which data types are allowed for fields.
- The maximum size of fields.

Next, let's look at how fields are actually declared and used in some programming languages. In all cases, we want to create a field of six variables that can hold the names of a group of people. We then want to set the second name to "Sophie".

The whole thing first in Visual Basic for Application (VBA):

```
Dim names(6) As String
names(1) = "Sophie"
```

The same in JavaScript:

```
var names = [];
names[1] = "Cathy"
```

As you can see, in JavaScript the size of the array does not have to be specified in advance. The indexing also starts here at 0.

And finally, Delphi:

```
var
names: array[1..6] of string;
names[2] = "Cathy";
```

Here, the value range of the indices can be set explicitly. We set it so that it runs from 1 to 6. The second element thus has the index 2.

Finally, it should be mentioned that in some programming languages string variables are understood as fields of individual characters. The individual characters within the string can then be accessed by normal indexing. Consider the following examples from C and Python; first the C version:

```
char my_name[] = "Thomas";
printf("Third character: %c", my_name[2]);
```

Then in Python:

```
my_name = "Thomas"
print("Third character: ", my_name[2])
```

In both cases we grab the character with index 2, i.e., the third character (because both programming languages start the field indexing at 0) and display it. Both languages understand strings as fields, but C is much stricter than Python.

11.6 Associative Fields of Variables/Hashes

In addition to the classical ordered fields/arrays, some programming languages know a second type of fields, the *associative fields*, sometimes also called *hashes*, *dictionaries* or *maps*.

Associative fields consist of an *unordered* set of *key-value pairs*. The key can be used to access the respective value. Such key-value pairs could, for example, consist of the name of a customer (key) and the order value of his last order (value). The entries in an associative field, unlike those in an ordered field/array, do not have a natural order; they don't need one, since the individual elements are accessed via a clearly defined key.

Let's look at the example of two languages that support associative fields: Perl – associative fields are called *hashes* here – and in Python where they are called *dictionaries*. First the Perl variant:

```
my %ordervalue = (
    "Thomas_Schultz" => 43.99,
    "Jim_Scott" => 19.49,
    "Mary_McGregor" => 68.99,
);
$ordervalue{" Mary_McGregor"} = 8.99;
print('Last order of Thomas Schultz was: $bestellwerte{" Jim_
Scott "});
```

11

As you can see, the upper part of the code first creates a new hash field called **ordervalue** (hash identifiers in Perl are always preceded by the percent sign when speaking of the field as a “whole”). The hash field is immediately initialized with three key-value pairs. To the left of the operator `=>` is the key, in this case the name of the customer, to the right of the operator the last order value. Further down in the code, a special element of the hash field is accessed, once to change a value, another time to display a value on the screen. As expected, access is not made with a numerical index (after all, the elements in the hash field are not sorted), but via the key, in this case the customer name.

Now the same again in Python:

```
ordervalue = {
    "Thomas_Schulz" : 43.99,
    " Jim_Scott" : 19.49,
    " Mary_McGregor" : 68.99,
}

ordervalue['Mary_McGregor'] = 8.99
print("Last order of Thomas Schultz was: ",
ordervalue['Thomas_Schultz'])
```

Even if the exact syntax differs slightly in both languages, the parallels in dealing with hashes or dictionaries, as the associative fields are called in Python, cannot be overlooked. It becomes clear how easily values can be looked up in an associative field – in fact, like a dictionary.

The languages that support associative fields usually come with a whole set of tools for analyzing and manipulating such fields. For example, functions or operators are regularly available to the programmer to extract all keys or all values from the field in one fell swoop, or to determine the size of the field, that is, the number of key-value pairs it contains.

? 11.1 [3 min]

What is meant by the declaration of variables?

? 11.2 [3 min]

Name two advantages of being forced to declare variables.

11.7 Objects

In the last two sections, we dealt with fields. Fields allow you to store a lot of similar information in an orderly way and to access it again. This is very useful in many cases, but often not the easiest or most natural way to deal with data.

In this section, you will therefore learn about an approach to working with related data that is so fundamental that it constitutes a programming paradigm of its own. Many programming languages have adopted this approach, in whole or in part. Because it shapes many popular languages, such as C++, Java and JavaScript, Python, and Kotlin, it is extremely important in practice.

We are talking about *object-oriented programming* (OOP for short). We will deal with it quite extensively in this section, and not entirely without ulterior motives, since the two programming languages introduced in the third and fourth parts of this book also belong to the broad class of object-oriented languages. In the eyes of not a few contemporaries, object orientation is associated with attributes such as “difficult to understand” and “complex”. After reading this section, however, you will find that such fears are not justified at all.

11.7.1 The World Is Made of Objects

Let’s assume we wanted to program a catalog-like display of products for an online store. Each product has a set of properties that we want to display, such as: a name, a description, an item number, a manufacturer, a price, and quantity. Using the knowledge from the previous sections, we could map each of these properties as a separate field/array. Then, for example, there would be an array **itemnumbers** and **itemnumbers[187]** would be the item number of the 187th article. If we wanted to retrieve the item description for the same item, we would use the array element **descriptions[187]**.

The leading criterion here is therefore the respective property. The product for which we query this property is indicated by the index, in our example, 187. However, this approach is a little artificial, because in reality we do not normally start from the property, but from its carrier.

When we program our catalog view, we are faced with the problem that we have a *product* in front of us and we need to display it, with *all the* relevant attributes. We start from the product and ask ourselves what attributes this product has. So, we then take the name, the description, the item number and all the other attributes that we want to display in our catalog list and display them for that particular product. We always look at one and the same *object*, namely the product, from all possible points of view (its attributes). It's the same when you go to a used car dealer and check out the cars for sale in their yard. You look at a car, check the model, color, age, condition, price and other parameters that are important for your assessment. Then you look at the next car and then the one after that. But you always start from one object, the car, and look at its respective characteristics.

You've probably noticed what this reasoning boils down to: The world is simply not organized by properties, but by objects, whether that be products, cars, houses, businesses, buttons, emails, or students. All these physical and non-physical objects are ultimately summaries of properties. In this sense, people or roles that people perform (such as the role of "student") can also be objects. We shouldn't be too particular about the choice of language here, although at first it might seem a little odd to think of a customer or a colleague as an "object". But if we define an object as a set of properties, it becomes clear that in this broad sense people, animals, plants, and gods are also "objects".

But if the world is organized by objects and not by properties, why is this not reflected in programming? Why do we work with fields that clearly focus on a single property and not on an object as a whole collection of different properties? In the 1960s, the American computer scientist *Alan Kay* and other pioneers of object-oriented programming also asked themselves this question and, in response, brought about a paradigm shift in the truest sense of the word. One of the first programming languages to follow this new paradigm was Kay's *Smalltalk*.

In the concept of object-oriented programming, the object is in the foreground; it embodies the organizing principle of this approach. It is no longer the properties that determine the way we deal with data, it is the bearers of the properties, the objects.

11.7.2 Classes

Let's follow the object-oriented approach and define an object product with the following properties:

```
product.name
product.description
product.itemnumber
product.manufacturer
product.price
```


All products that we sell through our shop have these properties. In order to recognize that the properties all describe the object “product”, we summarize them by prefixing them with **product**.

Let us now be a little more precise linguistically. What we have defined there is our view of *any* product, it is in a sense the *template for a product*; this is what products look like to us, these are their essential characteristics from our point of view. Such an abstract template, which describes which properties an object has, is called a *class* in object-oriented programming. Each real product that we offer has an individual value for each of these properties, for example the name “Garden shovel, stainless steel” and a price of 10.99 USD. The actual objects whose properties are modeled after our class are called *instances* of the class. So, in a way, an instance is the concretization of the abstract idea expressed in a class. All our products will have different values for each of the properties, so there are as many instances as there are products. But all products belong to the same class, they are just products.

In the next step, we’ll get a little more formal and define our class as we would in a programming language:

```
Class Product
Begin
    name : String
    description : String
    itemnumber : Integer
    manufacturer : String
    price : DecimalNumber
End
```

The properties of the class – we also speak of *attributes* in this context – are located between the limiting keywords **Begin** and **End**. This code excerpt here is not written in any existing programming language, but is formulated as “pseudo-code”, as we will do several times in this part of the book in order to illustrate basic principles. The point here is simply to describe, in a formalized but easily understood way, what a class definition might look like. Later, you will see how class definitions are constructed in some real programming languages and will be able to cope with them immediately once you have understood the basic concept from this pseudo-code.

Now that we have defined which attributes our class should consist of and what data type these attributes have, we can create an instance of the class, i.e., a variable that (built along the lines of the class definition) represents a concrete product. As soon as the new variable of type product is created, we can start to adjust its attributes:

```
GardenShovel : Product

GardenShovel.name = "Garden shovel, stainless steel"
GardenShovel.price = 10.99
```

To access the attributes of the **GardenShovel** instance of our **Product** class, we use the dot operator in the form *instance.attribute*. This notation is common in many programming languages.

With **Product**, we have created our own data type, which is more complex than the data types we learned about in the previous sections, because it stores different values. However, we can work with it just as with one of the “built-in” data types (such as integers or logical values). For example, we can create variables of this type and assign values to them (not to the variable as a whole, but to the individual attributes, which are of course elementary variables).

11.7.3 Inheritance

Sometimes we have objects that are special cases of other objects. A book, for example, is a special product. It has all the properties that our products have, it has a name (the book title), a manufacturer (the publisher), and of course it has a price. In addition, it has a couple of other properties that we should also show in our web shop, for example the author and the number of pages. After all, both are information that could influence the purchase decision of our customers.

Now, in order to represent our special product “book” as a class, there is a trick in object-oriented programming called *inheritance*. Contrary to what the term might suggest, nobody must die to be able to inherit. The basic idea, however, is easy: Our book as a special case of the product simply “inherits” all the attributes that a product has and gets, with the author and the number of pages, two more attributes. These two attributes are what makes the book special. So, our book is a product, but not every product is a book. There are products that have only the standard properties of products, but not the special properties of author and number of pages; these are unique to books.

Of course, we could include these additional properties that books have beyond the general definition of a product directly in the **Product** class. But what values should we then assign to these attributes for concrete instances of our class that are not books? And what happens if we give special treatment to other special cases of products, such as clothing, or garden furniture? The number of attributes that are then no longer applicable to all products, but only to a single product category, would increase considerably, making the class **Product** very confusing.

It is simpler and more elegant with inheritance. We create a new class **Book**, which takes over all properties of the above defined class product and additionally adds the attributes number of pages and author. The class definition for the class book could then look like this:

```
Class Book Inherits Product
Begin
    author : String
    pages : Integer
End
```

If you compare this class definition with the definition of the class **Product**, you will notice that the keyword **Inherits** is added here, followed by the class whose attributes our class **Book** is to take over (i.e., “inherit”). Of course, we can now create instances of our class, i.e., concrete variables, and change their attributes.

```
Grisham1992 : Book
Grisham1992.name = "The Pelican Brief"
Grisham1992.price = 8.99
Grisham1992.author = "John Grisham"
Grisham1992.pagenumber = 478
```

As you can see here, in this code snippet we edit our variable **Grisham1992** not only with regard to the special properties of books, namely author and number of pages, but also with regard to the standard attributes of products, namely name and price. These do not occur explicitly in the definition of the class **Book**. However, books inherit these properties from the more general class **Product**, from which the class **Book** is derived.

Ultimately, we build a *class hierarchy*, with **Product** as the superclass and **Book** as the subclass. Class hierarchies can of course have much more than two levels; for example, we could add novels and encyclopedias as further subclasses of books and provide them with special attributes that are not contained in the class book. In the same way, we could of course extend the hierarchy by modeling other product categories besides books (for example, clothing and garden furniture) as separate classes that are directly derived from the **Product** class.

Next, let's take a look at what our two classes, **Product** and **Book**, would look like in two real programming languages; we'll start with C++:

```
class Product
{
    char designation[30];
    char description[200];
    long itemnumber;
    char manufacturer[30];
    float price;
}

class Book : public Product
{
    char author[50];
    int pages;
}

// Later in the main program ...

Book grisham1992;
Product gardenshovel;
```

```

gardenshovel.name = "Garden shovel, stainless steel";
gardenshovel.price = 10.99;

grisham1992.name = "The Pelican Brief";
grisham1992.price = 8.99;
grisham1992.author = "John Grisham";
grisham1992.pages = 478;

```

Of course, even if you are not familiar with the particular syntax of C++, your understanding of the concepts of object-oriented programming and our pseudo-code will allow you to understand what is happening in this program snippet.

Now let's look at the same thing in Delphi/Object Pascal:

```

type
TProduct = Class(TObject)
    property Name : String;
    property Description : String;
    property ItemNumber: Longint;
    property Manufacturer : String;
    property Price : Single;
end;

TBook = Class(TProduct)
    property Author : String;
    property Pages : Integer;
end;

// Later in the main program ...

var
GardenShovel : TProduct;
Grisham1992 : TBook;

GardenShovel.Name = "Garden shovel, stainless steel";
GardenShovel.Price = 10.99;

Grisham1992.Name = "The Pelican Brief";
Grisham1992.Price = 8.99;
Grisham1992.Author = "John Grisham";
Grisham1992.Pages = 478;

```

Here we have followed the Delphi-typical notation, that classes (and in general all data types defined beyond the basic data types) begin with “T”, our two classes are then called accordingly **TProduct** and **TBook**. Behind the keyword **Class**, the class

from which is inherited is written in brackets. This is the class one level higher in the class hierarchy. The interesting thing here is that the class **TProduct** also inherits its attributes from a higher class, namely from the class **TObject**. This class is the highest in the class hierarchy, and all other classes are ultimately derived from it.

You can see from these examples that class definitions in programming languages may have their own peculiarities, but they still have a lot in common. With the few basic ideas of object-oriented programming that we have looked at so far, you can already understand what the class definitions mean in the respective language without a detailed understanding of the programming languages they are written in.

11.7.4 Methods

In the examples of the previous section, we directly changed the attributes, i.e., properties of our classes, by assigning values to them. For the “pure doctrine” of object-oriented programming, this is sacrilege. According to the “pure doctrine”, the attributes may not be edited directly, but only with the help of something called *methods*. Methods are callable subroutines to which you can pass certain values, called *arguments*, and which then process these values in some way, for example by assigning the “passed” value to a class property.

To make this a bit more concrete, let’s assume that our product class has a **set-price()** method that can be used to edit the price. The method is passed the price as an argument, and the method in turn then makes sure that the class property **price** is changed accordingly. The class definition would then look like this:

```
Class Product
Begin
    name : String
    description : String
    itemnumber : Integer
    manufacturer : String
    price : DecimalNumber
    setPrice(newprice: DecimalNumber)
End
```

Our original class has been extended by the method **setPrice()**. This method takes as argument **newprice** a decimal/floating point number, namely the price we want to set for our product. We could then create a new instance of the **Product** class later in the program and initialize the price using the **setPrice()** method, here in the example to the price 10.99 USD:

```
GardenShovel : Product
GardenShovel.setPrice(10.99)
```

Here we see now apparently a substantial difference to the fundamental data types, which we became acquainted with so far, such as integer or character string variables. The classes of object-oriented programming not only consist of data values, but, with the methods, also contain the tools to process these data. But this is only an apparent difference. In fact, even the fundamental data types in many object-oriented languages are themselves classes that offer a set of methods to the outside world. For example, the **DecimalNumber** class might provide a method **round()**; if **price** were a **DecimalNumber** object, that is, an instance of the **DecimalNumber** class, then, for example, **price.round(2)** would round the value of the variable **price** to two decimal places.

But why so complicated? Why don't we just stick to assigning values directly to the attributes of our class instances? Why is a special method necessary, which itself has to be developed? In our example above, we did without it for the sake of simplicity and assumed that the method **setPrice()** is already programmed somewhere and can be used by us; therefore, a hint (a *prototype*) in the class definition that the method should be part of the class was sufficient. But in fact, the code behind this method, the code that is executed when the method is called, must of course also be developed. So why all the effort just to change a value, which we could have done with a simple assignment?

Proponents of object-oriented programming would argue that the use of methods shields the internal data structure of the class from the outside world, i.e., from the programmer using the class. Programmers don't have to worry about how the various facts are mapped into the class; after all, they don't edit the class attributes directly, but via the methods. The developer of the class could change the class attributes, but as long as the methods available to the user of the class do not change, the user will not notice the changes. From their point of view, everything remains the same. The programmer does not have to rewrite software but can continue to work with the existing code without making any changes.

The advantage of using methods is that the modularization of code and thus the division of labor between the developer of the class and the programmer who uses the class in his programs is simplified. The developer of the class is responsible for the functionality that his class provides via the methods, the programmer as the "consumer" of this class only has to call the methods that are always to be used in the same way and does not have to worry about their exact functionality. This type of programming, in which a *programming interface* is provided externally in the form of the methods, makes the programs *more robust*, i.e., less susceptible to changes.

A second factor that contributes to the robustness of object-oriented programming is that methods can naturally ensure that *only permissible operations* are performed. Suppose our programmer wanted to assign the value -10.99 to the attribute **gardenshovel.price**. If he can easily assign values to the attribute **price**, he could also assign it a negative price. However, this could have unfavorable effects at a later point in the program, such as when the customer is supposed to "settle" a negative invoice amount and would thus ultimately receive a refund. This is where the methods can show their strengths: Our **setPrice()** method could check whether the price passed to it as an argument is greater than 0. If it is, the attribute **price** would be set to that value. Otherwise, that is, if the price is negative, the attribute

would be set to the value 0. In this way, the method would prevent invalid prices, such as negative prices, from being set accidentally. By validating the price, the method contributes to the stability of the program; in other words, it is no longer easy to “upset” the program, it becomes more robust against erroneous data entries.

A special method that exists in practically all object-oriented languages is the *constructor*. The constructor is called automatically when a new instance of the class is created. It can be used, for example, to initialize certain important attributes of the class, either with default values or with values passed as arguments to the constructor method. If we were to include such a constructor in our **Product** class, our class definition might look something like this:

```
Class Product
Begin
    name : String
    description : String
    itemnumber : Integer
    manufacturer : String
    price : DecimalNumber
    setPrice(newprice: DecimalNumber)
    Product(startprice : DecimalNumber, name : String)
End
```

What is new is that the class has a constructor method **Product()**. It is named the same as the class itself and takes two arguments, a price and a product name. With these two data, the constructor could now initialize the attributes **price** and **name**, when a new instance of this class is created. To do this, the constructor must of course be called with the two arguments when the instance is created; this could look like this, for example:

```
GardenShovel : Product(10.99, "Garden shovel, stainless steel")
```

As you can see, we declare here – as in the previous examples – a variable of type **Product**, but this time the constructor is called with the necessary parameters, the price and the name.

11.7.5 Polymorphism

This term may sound like a disease, but *polymorphism* is by no means a negative phenomenon; on the contrary, it is a very practical possibility offered by object-oriented programming. Polymorphism is closely related to the concept of inheritance. You remember that classes can “inherit” their methods and attributes to derived classes. In the previous sections, we defined a class **Book** that inherits all the properties and methods of the more general class **Product** and can also have its own properties and methods that are not available in the “parent class” **Product**.

Now we could define a method that displays the properties of the product, i.e., creates a kind of product profile. We could put this method in the general class **Product**. Thanks to inheritance, it would also be available for the class **Book** derived from **Product**. However, the display would ignore the special properties of books, such as the author or the number of pages, both of which are attributes of the **Book** class. These properties are only part of the **Book** class, not the **Product** class, so a display method that we place in the **Product** class naturally cannot access these properties. The garden shovel from the previous examples, an instance of the general class **Product**, has no number of pages!

However, it would be practical if we had a display method that simply returned the correct display for each product, no matter what type of product we were dealing with. Ideally, we would call the display method **showProduct()**, and it would take care of outputting the right information on the screen for each type of product.

This is exactly what polymorphism allows. Polymorphism means that classes that are derived from each other can have methods with the same name, but they all do something different. If the method is then called for a specific object, i.e., an instance of a class, the method belonging to the *respective class* is automatically executed. In our example, the properties author and number of pages would then also be displayed.

Such a polymorphic design of the method **showProduct()** could look like this:

```

Class Product
Begin
    name : String
    description : String
    itemnumber : Integer
    manufacturer : String
    price : DecimalNumber
    showProduct()
    setPrice(newprice: DecimalNumber)
    Product(startprice : DecimalNumber, name : String)
End

Class Book Inherits Product
Begin
    author : String
    pages : Integer
    showProduct()
End

```

As you can see, both classes, the “parent class” **Product** and the derived “child class” **Book**, each have a function **showProduct()**. Which one is executed when we call the method depends on whether the object for which we call the method is an instance of **Product** or an instance of its derived class **Book**.

So, if we declare two objects

```

GardenShovel : Product
Grisham1992 : Book

```


and then call the method **showProduct()** for each of the two objects,

```
GardenShovel.showProduct()  
Grisham1992.showProduct()
```

two different methods are ultimately called; for the object **GardenShovel** the method of the class **Product**, of which **GardenShovel** is an instance, for the object **Grisham1992** the method of the derived class **Book**, so that **showProduct()** can then also display number of pages and author correctly.

The handy thing about polymorphic methods is that we don't have to bother with what kind of object **GardenShovel** and **Grisham1992** actually are. We just stubbornly call the **showProduct()** method, and what happens is always what's best for the object class in question; the same names of the methods make it possible.

A term that comes up again and again in connection with polymorphism is *overloading*. One speaks of the method **showProduct()** of the class **Product** being overloaded by classes derived from **Product** each bringing their own method **showProduct()** in order to optimally consider their specifics. Both terms describe the situation well: While “polymorphism” refers to the fact that (apparently) one and the same method can have many (Greek *poly*) forms (Greek *morphía*), “overloading” describes the process by which the same function is given a different meaning several times.

11.7.6 Access Rights

To conclude our look at object-oriented programming, let's look at one last feature that reemphasizes the motivation of object-oriented programming to strictly separate the development from the use of classes.

There is a way to restrict access to attributes and methods of classes. The concrete design can vary from programming language to programming language, but usually there is at least the following gradation of access rights:

- *Private*: Methods and attributes under this access restriction can only be used by methods of the same class. They are not “visible” to the outside world; as a user of the class who uses the class in his or her own programs, you could not access these methods and attributes. However, a method that you call (which is not itself private) could use these methods and attributes. However, you cannot do this directly. Private methods and attributes are therefore shielded from the outside world. Access protection *private* is therefore well suited to defining auxiliary variables or auxiliary methods that are not called from outside but are only to be used by other methods of the class.
- *Protected*: Methods and attributes that have been declared as protected can be used by the class to which they belong, and also by derived classes, but not by the programmer working with these classes in his program.
- *Public/open*: Methods and attributes with the access restriction public can be accessed from anywhere, from within the own class, from derived classes and also by the user of the class.

Consider the following extension of our example for clarification:

```

Class Product
Begin
    Public
        name : String
        description : String
        itemnumber : Integer
        manufacturer : String
        showProduct()
        setPrice(newprice: DecimalNumber)
        Product(startprice : DecimalNumber, name : String)
    Private
        price : DecimalNumber
End

```

Here we have declared the attribute **price** as a private property. The **setPrice()** method, on the other hand, is a public method. So, as developers of the class, we don't want anyone to directly edit our attribute **price**. Therefore, we protect it as private. However, a method from the same class can access it and change its value. **setPrice()** is such a method. It is a public method that can be called from outside the class. So a programmer using our class could now edit the attribute **price** via the interface method **setprice()**, but not directly, for example by assigning a value.

In this way, it is very easy to control which parts of classes should be visible to the outside and serve as an interface to the functionalities of the class, and which should not.

? 11.3 [5 min]

Are the following statements true or false?

- Object-oriented programming is the attempt to achieve as “natural” a representation of things in the real world as possible.
- A method is a function that belongs to a class and can change the attributes of this class instance.
- All attributes of a class instance can be changed directly from the program by assignment.
- The use of object-oriented programming makes the program clearer but makes adjustments to the program more difficult.
- Inheritance means that you can reuse the definition of a class in different programs.

? 11.4 [3 min]

Describe the difference between a class and an instance.

? 11.5 [3 min]

What are the essential elements of a class definition?

? 11.6 [3 min]

Why is polymorphism a useful approach in object-oriented programming?

11.8 Your Roadmap to Learning a New Programming Language

When you learn a new programming language ... you'll discover:

- whether the language is case-sensitive for the identifiers, i.e., the names of variables and functions,
- which basic data types the language offers (especially numbers, strings, logical/truth values),
- whether you have to declare variables, and if so, how to do it
- how to assign values to variables,
- whether, and if so how, variables can be (explicitly) converted between data types and which conversions, if any, are already (implicitly) performed by the programming language itself,
- whether the language supports fields of similar variables (arrays), and if so, how to create fields and access their elements (in particular whether the indexing of the field elements starts at 0 or at 1),
- whether the language knows associative fields whose values can be accessed with keys, and if so, how to create these fields and address their elements,
- which other complex data types are common in the programming language,
- whether the language supports object-oriented programming, and if so, how to access class attributes and methods, and how to define classes yourself, especially how to derive classes from existing classes.

11.9 Solutions to the Exercises

■ Exercise 11.1

In the languages that require a variable declaration, the variable is registered with the interpreter/compiler by the declaration; it reserves the necessary memory and, if necessary, assigns an initial value to the variable. After that, the variable can be used in the program. When declaring a variable, the identifier (name) of the variable and, in some languages, also its data type is specified.

■ Exercise 11.2

The need to declare a variable allows the interpreter/compiler to indicate the use of undeclared variables. Because undeclared variables are often the result of typos in the program code, this prevents a new variable from being accidentally created and worked with, while the variable that should have been accessed remains completely unchanged. The program code thus becomes more robust by forcing variables to be declared. The same is true if the type is already specified when declaring, and this type cannot be changed afterwards. In this case, the interpreter/compiler can report an error if the variable is accidentally assigned a value of an “inappropriate” type. This also avoids errors and makes the program code more robust.

■ Exercise 11.3

- (a) True.
- (b) True.
- (c) False. In many programming languages, attributes of class instances can be shielded from external access by defining them as “private”. These attributes can then only be edited by methods of the same class but are practically invisible to the outside world and therefore cannot be accessed directly by the programmer.
- (d) False. Object-oriented programming helps program elements become more independent of each other. Because the programmer only addresses the class instance via the defined methods (and, if necessary, by directly accessing the attributes), the inner workings of the class need not interest him or her any further. Thus, as long as the *interface* of the class remains unchanged externally, the developer of the class itself can change it internally at will, and the programs based on the class remain syntactically correct. Adaptations to the code are facilitated by this stronger modularization.
- (e) False. Inheritance means that further classes can be derived from a class, which “inherits” its methods and attributes. In this way, a class can be extended elegantly, especially for more specific uses.

■ Exercise 11.4

The class is the abstract definition of an object (or object type) with the attributes and methods belonging to objects of this type and acts like a template. According to this template, the concrete objects, the class instances, are formed and therefore possess as images of the class all its methods and attributes.

■ Exercise 11.5

Essentially, a class definition consists of the identifier (name) of the class and the attributes and methods belonging to the class. These can be provided with access right restrictions by means of corresponding keywords (cf. ► Sect. 11.7.6). If the class is derived from another class, the reference to the “parent class” is also part of the class definition (see ► Sect. 11.7.3).

■ Exercise 11.6

Polymorphism allows a particular method to be offered by objects of different types (classes). This makes it possible to adapt the method to the specifics of the respective class. This is particularly interesting if a class hierarchy is created by inheritance. If the method is now called for the instance of a class from this class hierarchy, the special implementation of the method for the class of the object instance is used. If this class does not have a special implementation of the method, the method with the same name of the next higher class is used. This ensures that you always use the method that is best suited to the special features of the class in question, but that, if necessary, you use methods that belong to classes higher up in the class hierarchy. This means that different object types can be handled differently, but still offer the same interface (that is, the method with the same name) to

the outside world. The programmer does not have to deal with the question of which method of which class he or she should actually call; he or she simply calls the method for his or her object instance, and the interpreter/compiler clarifies for him or her exactly which method should be used in this case.



How Do I Input and Output Data?

Contents

- 12.1 Forms of Data Input and Output – 114**
- 12.2 Graphical or Non-Graphical: That Is the Question Here – 115**
 - 12.2.1 Graphical User Interfaces – 116
 - 12.2.2 Console Applications – 124
- 12.3 Working with Files – 126**
- 12.4 Working with Databases – 131**
- 12.5 Your Roadmap to Learning a New Programming Language – 134**
- 12.6 Solutions to the Exercises – 134**

Overview

In the last chapter we dealt with the question of how data can be stored with the help of variables in such a way that we can work with them in the program. We now turn to the question of how we can bring data into the program from the outside (in this context we speak in a broader sense of “data input”) and also get it out of the program again (“data output”). This involves both data input and data output directly from/to the user, as well as input and output in the context of files and databases. In this chapter, we deal with how a program communicates with its “outside world”.

You will learn the following:

- What distinguishes the two key basic modes of direct user input and output, graphical user interfaces and console applications, and when which form is preferable
- Which the most important control elements on graphical user interfaces are
- How to design graphical user interfaces
- How to read data from and write data to files
- How to work with databases in the basics.

12.1 Forms of Data Input and Output

Most programs communicate in one way or another with their “environment”. The “environment” includes, of course, first and foremost the user of the program, who provides information and makes decisions. But part of the environment are also other objects and phenomena whose properties and states have an influence on the program flow. For example, if we are talking about software that is supposed to proactively turn up your heating when it starts to get cold, the current (and possibly predicted) temperatures are part of the program’s relevant environment. Of course, the program can only process information about this environment if it is made known to it in the form of data. In this chapter we want to take a closer look at this “making known”. It is therefore a question of how information can be “entered” into the program in the form of data.

When the term “input” is used, the first thing that comes to mind is probably direct input by the user, and especially input with the aid of a keyboard. But of course, this is not the only way that the user can provide data to the program. Other input devices, of which microphone, webcam, mouse, joystick, or touchscreen are only a few examples, allow you to input very different kinds of data—for example sound, video, position, direction and speed data. But not all input comes directly from the user. Files and databases are also possible sources of data input that a program works with. The program for heating control, for example, will perhaps query data from the database of a meteorological service via a web interface to determine whether the heating needs to be turned on, and if so, when, and how much.

Conversely, a wide range of options are available for *outputting* data. Examples of this are, above all, the output on the screen and the writing of data to files and databases.

In this chapter, we will deal with three types of data input and output in programs—input or output by or to the user via a *user interface* of some kind, as well as working with *files* and *databases*. Since the latter is not a trivial matter and usually requires knowledge of a separate programming language (a query language) developed specifically for this purpose, we will only cover this subject area here in an overview, but at least in such a way that you develop a solid understanding of this enormously important field in the practice of modern software development. Of course, there are other ways in which data can get into your program or be made available from your program, for example via APIs (Application Programming Interfaces). As an introductory book, however, we will concentrate on the three mentioned above.

12.2 Graphical or Non-Graphical: That Is the Question Here

Since the advent of operating systems with graphical user interfaces and the triumph of the World Wide Web, we have become accustomed to entering and viewing data on attractively designed *graphical user interfaces* (GUIs, or, because graphical is the standard today anyway, simply UIs). What “attractive” means in this context depends on taste and the technical possibilities of the respective time and technological environment.

What GUIs of all times and all technical varieties have in common is that they try to make the input of data as convenient as possible for the user by means of suitable *control elements*. For example, by not having to enter numbers from a given range of values via the keyboard, but by allowing them to be conveniently defined by means of sliders.

Especially in the course of the increasing importance of mobile and web applications, completely new job descriptions have developed, those of the *user interface designer* (UI designer) and the *user experience designer* (UX designer). While the UI designer technically designs the interface and “wires” it to the program code behind it, the UX designer is intensively concerned with the end users and their behavior. In a sense, he is the UI designer’s market researcher and tries to find the optimal way for end users to interact with the application. The UI designer can then implement the UX designer’s findings in the user interface. While the UI designer is more concerned with the “inside”, with technical design, the work of the UX designer is more concerned with the “outside” and consists mainly of understanding the end users and their way of working.

If you’re not professionally involved in programming, you’ll usually be a developer, UX designer and UI designer all rolled into one. This is more work, but it also means that you can let your creativity run wild when designing your interfaces. However, you will also have to consider the needs and desires of your users, unless you are developing the software exclusively for yourself.

With the triumph of graphical user interfaces, *console applications* have almost completely gone out of fashion in the end-user area. Those who worked with operating systems such as MS-DOS in their early computer days, or who today operate a Linux system and do so not exclusively via one of the graphical front-ends for

Linux, may still remember with a pleasant shiver. Console applications offer only one form of direct input by the user, namely via the keyboard.

A central difference, however, is not only in how the interface appears visually and in terms of ease of use. The program flow of console applications is usually quite different from that of applications with GUIs. Based on Karl Marx's famous dictum that being determines consciousness, one could say: "The interface determines programming". This is because console applications are usually *linear programs*, they run step by step. For example, first something is displayed (such as: "Please enter your username"); then the program waits for user input. Once the user has made his input and confirmed it with <ENTER> or <RETURN>, the program's next output follows ("Please enter your password"); the program waits again until the user has made his input and confirmed it. Then the program processes the input (checks username and password for validity, for example) and responds again with an output ("Access granted."), and so on.

This is different with a graphical user interface. Here, the program usually does not specify an exact sequence of user actions. In our example, the user could also first enter the password and only then the username. Only a click on the login button triggers the program to check the user input. The difference becomes even clearer when you consider a graphical user interface such as that of a word processing program, where the user can use buttons to access a variety of functions or just start working on the document text. This way of working with the program is *not linear*. Instead, the program observes what the user is doing and responds to *events*, such as clicking a button or selecting a function from a menu. Programs with graphical user interfaces are thus typically *event-driven*. If the user triggers a certain event, the program jumps to the place in the program code that describes what to do when this event occurs. If the user then triggers another event, the program again jumps to the right place, no matter where exactly in the program's source code that section of code might be found. Unlike the console application, which strictly executes one line of code after another and consists of a linear sequence of statements, in event-driven programming the processing "jumps" from one block of statements to another block somewhere else, depending on what the user is doing.

In ► Chap. 14, when we talk about how to make programs respond to the user's input, we will look at event-driven programs in a bit more detail. In this chapter, we will first deal with the possibilities of data input, that is, the interface as such.

12.2.1 Graphical User Interfaces

12.2.1.1 Important Controls

The controls used on graphical user interfaces may look different depending on the operating system (for example, *Windows*, *macOS*, *Android*, *iOS*) and platform (computer, tablet, smartphone). However, many elements exist in virtually all operating systems and on virtually all platforms.

In the following section, we will take a brief look at the most important control elements and their central properties. In addition to the properties mentioned here, the controls have a number of other characteristics. Some of them are specific to the respective control element, others are common to all or at least to the vast majority of controls. The latter include the position on the surface, the dimensions (height and width), the visibility (is the control element currently visible or is it hidden?), the usability (can the user currently use the control element? Is it active, or is it inactive and thus locked and “grayed out?”), its name (so that it can be addressed in the program), and its color.

■ Edit fields

Graphical user interfaces provide edit fields for classic keyboard input. Depending on the type or setting, these allow information to be entered in one or more lines (■ Figs. 12.1, 12.2, and 12.3).

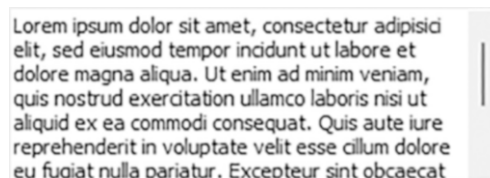
Edit fields can typically be configured extensively, or different edit fields with different properties are available. For example, the user’s password entry can be masked by simply displaying nothing or a specific character instead of the characters entered. Also, the information entered can often be validated as it is entered. For example, this can be used to force the user to enter a numeric input; if the user were to enter letters instead, that text would not be accepted as input by the edit field at all. Sometimes it is important that different parts of the input are formatted differently (for example, when entering program code that is to be displayed with syntax highlighting); then you need an input field that can handle different text formatting.



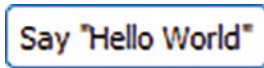
■ Fig. 12.1 Example of an input field



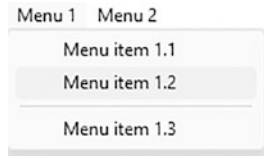
■ Fig. 12.2 Example of an input field



■ Fig. 12.3 Example of an input field



■ Fig. 12.4 Example of a button



■ Fig. 12.5 Example of a menu

■ Buttons

Buttons are defined clickable places on the screen through which the user usually triggers an action, for example saving a document or sending a message (■ Fig. 12.4). The most important feature of the button is its label and the link to the program part that is executed when the button is clicked. The visual appearance is also often dominated by a symbol image that stylizes the action triggered by the button.

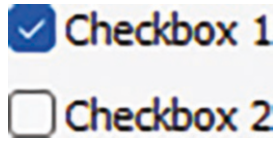
■ Menus

Like buttons, menus are used to let the user trigger actions. The focus is on the selection among various options in the form of the menu items. This also explains the name of the control element, which the user uses in a similar way to the menu in a restaurant. In addition to the names of the menu and its items, important properties—again analogous to the button—are the actions that the user can trigger by clicking on the items (■ Fig. 12.5).

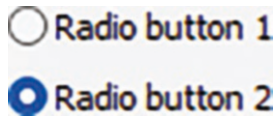
■ Checkboxes and Radio Buttons

Checkboxes and radio buttons are control elements that allow the user to make a selection between several options. Unlike buttons and menus, clicking on one of these elements does not usually trigger an immediate action. Instead, checkboxes and radio buttons are usually used to specify settings that control the exact behavior of actions that the user later triggers elsewhere, such as by clicking on a button or menu item. For example, the user could use a radio button to choose whether to open a file read-only or not. The actual action, the opening of the file, is triggered later by clicking on a button “Open file”. What exactly a click on this button does (whether the file is opened in read-only mode or in such a way that it can also be modified) is determined by the setting previously made with the help of the radio button (■ Figs. 12.6 and 12.7).

Checkboxes often appear in multiples, radio buttons always. The difference between the usually square checkboxes and the round radio buttons is that *only one option* can be selected from a group of such control elements in the case of radio buttons, but *several can* be clicked on in the case of checkboxes.



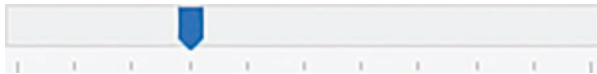
■ Fig. 12.6 Examples of checkboxes



■ Fig. 12.7 Examples of radio buttons



■ Fig. 12.8 Example of a toggle button



■ Fig. 12.9 Example of a slider

■ Toggle Buttons

Similar to checkboxes, *toggle* buttons allow you to toggle an option on or off. Unlike checkboxes, however, there are no checkmarks to indicate whether the option is currently selected or not. Instead, the appearance is modeled after a slide switch. Toggle buttons have become popular with the triumph of the mobile operating systems iOS and Android but are now also available on other platforms (■ Fig. 12.8).

■ Sliders

Sliders are control elements that allow you to make a selection along a scale, i.e., between options that can be arranged in an order based on some criterion. Like radio buttons and checkbox/toggle buttons, a change in the current selection by the user does not usually trigger an immediate action. In most cases they are used to make a setting that takes effect later when the user triggers an action, such as clicking on a button or menu item (though sometimes changes made by the user have a direct effect, such as when the slider is used to scale a graphic that automatically updates when the slider control is moved). Its most important setting is the scale, the gradation in which the user can adjust the slider control, particularly the number and designation of the expressions (■ Figs. 12.9 and 12.10).



■ Fig. 12.10 Example of a slider

Element	Property 1	Property 2	
Item 1	Property 1.1	Property 1.2	
Item 2	Property 2.1	Property 2.2	
Item 3	Property 3.1	Property 3.2	
Item 4	Property 4.1	Property 4.2	

■ Fig. 12.11 Example of a list view

■ List Views/List Boxes

List views or list boxes are “flat”, non-hierarchical lists of items that allow the user to select one or more of these items. An example of the use of list views are file managers that list the files contained in a folder. In this case, the elements shown in the list view can be provided with icons. Sometimes, in addition to the name of the element, other properties of the elements are displayed in extra columns of the list view, for example, their size or the date of their last modification. In addition to the type of display (icons, additional columns for supplementary element properties), an important property that can be specified when using list views is whether the user should be able to select only a single element at a time or several elements at once (■ Fig. 12.11).

■ Tree Views

Tree views are similar to lists in the sense that they allow multiple elements to be displayed. Unlike lists, however, they have a tree-like structure, i.e., they display hierarchical relationships between the elements. A classic example for the use of tree views are the folder views in file managers, where the hierarchy of the folders is typically displayed as a tree structure. This is a very good way to represent hierarchical relationships of any kind, such as the structure of an organization or a hierarchy of products, ranging from broad product categories down to individual products. Unlike list views, tree views typically have no additional columns for displaying further properties of the hierarchically arranged elements (■ Fig. 12.12).

■ Pickers

Pickers are control elements that allow you to make a selection from several predefined options, usually without immediately triggering any action. In this respect,

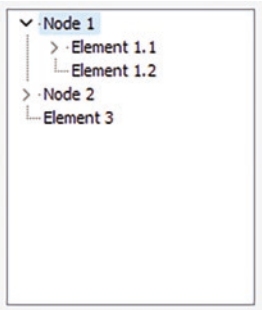


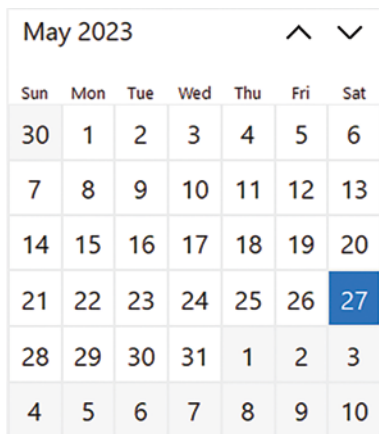
Fig. 12.12 Example of a tree view



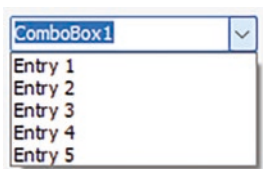
Fig. 12.13 Examples of date pickers

they are similar to radio buttons, which also allow a selection of settings from several predefined alternatives. However, the types and forms in which pickers appear are very different. Sometimes they allow the selection from a predefined, but large and complex set of possibilities. A good example of such a choice is the date picker that has become commonplace on mobile and non-mobile platforms. Two very different examples of such date pickers are shown in Fig. 12.13 and 12.14.

Of course, pickers do not necessarily have to be based on such a complex selection situation as the date. Pickers are often used to select from a manageable list of simple text options. In this sense, pickers also include the good old combo box as shown in Fig. 12.15, which presents a list of selection options that expands downwards.



■ Fig. 12.14 Examples of date pickers



■ Fig. 12.15 Example of a combo box

12 12.2.1.2 Developing Graphical User Interfaces

Now that we have looked at some popular controls for graphical user interfaces, the question naturally arises as to how exactly one technically develops an interface. Somehow the arrangement of controls and their properties have to be defined. The best interface design you have thought up and perhaps sketched out on paper or with the help of a graphics or presentation program won't help you if you can't turn it into a truly usable program interface, a real GUI.

This is often done with the help of the *Integrated Development Environments (IDEs)*, which we already learned about in ► Sect. 8.1.3. These often allow you to “click” together graphical user interfaces without programming by selecting the different controls and placing them where you want them. ■ Fig. 12.16 shows the integrated development environment of Delphi. Here you can see how a graphical user interface is being created. On the right (the “Palette” area), different control elements can be selected, which can then be placed in the program window using drag & drop. Their size can also be adjusted using the mouse. In the “Object Inspector” on the left, you can see the properties of the currently selected control element, in this case the selected button. The selected property you can see here is the “Caption”, the label of the button. Other properties are, for example, the shape the cursor should take when you move the mouse over the button (“Cursor”), the font of the button caption (“Font”) or the height of the control (“Height”). Obviously, even with a simple button you can make quite a few settings to define the appearance and behavior in detail.

tics of their properties. More convenient, of course, is editing by drag & drop in *WYSIWYG* mode (*what you see is what you get*), as seen in ■ Fig. 12.16.

Some programming languages are designed to define graphical user interfaces directly in the program code, similar to the way Delphi does it in the background. The individual elements of the interface are then usually objects in the sense of object-oriented programming. They are created by program statements, placed, adjusted with respect to their properties, and “wired” to the rest of the program code. You will learn more about this procedure using Python as an example in ► Sect. 22.2. In ► Sect. 32.4, we will see how, in the case of JavaScript, the interface is designed with program code, but in a different language (namely HTML) than the one in which the actual program is written.

12.2.2 Console Applications

■ How Console Applications Work

Console applications do not have a graphical user interface, only a text interface. They run either in the operating system’s console or terminal (for example, Linux Bash, Mac Terminal, or MS-DOS prompt) or in an integrated development environment. In the former case, they are programs that the operating system can execute directly (i.e., programs already in machine code) or programs in an interpreted language that are executed by calling the interpreter in the operating system’s console and having it execute the program code. When executing in an Integrated Development Environment, the program is called from the development environment, i.e., a graphical user interface, but it only has a text interface itself, so it practically runs like a console application. Again, the programming language interpreter is called and executes the program, except that the console in which it runs is hard-coded into the IDE. In Part III of the book, we will develop console applications with Python and call them from a graphical IDE.

The special feature of console applications is that they usually run in a strictly linear fashion. Whereas the user can decide for himself via the event-driven graphical user interface which functions he calls in which order, in the console application he follows the sequence that is permanently built into the program. Here is a simple example (where > shows something entered by the user):

```
Please enter the temperature in degrees Celsius:
> 23
Conversion to Kelvin or degrees Fahrenheit (K/F)?
> K
23 degrees Celsius in degrees Kelvin is: 296.15.
```

In a graphical user interface, you might be able to use a radio button to select which temperature system to convert your input to. And most importantly, you probably could have made this selection *before* you entered the temperature to be converted in degrees Celsius. Not so with the console application with its text interface: it dictates what you have to enter and when. In this case, the program does not run event-driven, but linear.

Here are two examples of how the first part of the above program might look in two programming languages, Pascal and Python; first in Python:

```
temp_celsius = input('Please enter the temperature in degrees
Celsius:')
target_scale = input('Conversion to Kelvin or degrees
Fahrenheit (K/F)?')
```

As you can see, it's quite simple to query an input from the user. Since you don't have to declare variables in Python, each input is ultimately just a single line of code.

Now the whole thing in Pascal:

```
program temp
var
    temp_celsius : real;
    target_scale : char;
begin
    write("Please enter the temperature in degrees Celsius:");
    readln(temp_scale);
    write("Conversion to Kelvin or degrees Fahrenheit (K/F)?");
    readln(target_scale);
end.
```

In Pascal, the variables we use must be declared, a floating-point number variable (**real**) and a variable that takes only a single character (**char**). The input as such is done here using the **readln()** function, which stands for *read line, which* means that after the input, a line break is automatically done so that the next output starts on a new line. Without the **ln**, the cursor would simply stop after the input. So, the next output would start exactly at that point. There is an analogous distinction in output, where a distinction is made between **write()** and **writeln()**.

■ Why Console Applications at All?

There is one important question we haven't answered yet: Why would anyone develop a console application at all, today, when there are so many fascinating options for graphical interfaces available? The most obvious reason is that it's easier and faster. For example, if you want to try something out, like a new function library or a certain algorithm, it would be a waste of time to put a lot of work into designing a perfect graphical interface. Most of the time, a very simple console application that allows only rudimentary text interaction with the user is enough. Concentrate your work on the essentials and not on visual attractiveness and good usability, which is not necessary for your actual purpose!

To understand even better why console applications have not yet gone out of fashion, it is worth taking a look at professional or at least ambitious software developers. They greatly appreciate command line tools that run in the console. And this for several reasons: For one, working with command-line programs is

more ergonomic, because you can perform all tasks with just one input instrument, the keyboard, and you don't have to laboriously use the mouse to call up the corresponding buttons or menu items in a graphical development environment. In addition, command line parameters, special control options that are passed directly to the console program when it is called, allow the program behavior to be controlled effortlessly and very precisely, which would be incomparably more tedious with a whole series of clicks on a graphical interface. Also, console applications are regularly less demanding in terms of system resource consumption (especially memory and processing power), which can mean they run faster. It is therefore not surprising that many developer tools come as console applications, for example the well-known versioning tool *git* or the text editor *vi*. As a rule, these tools have the same parameters and control options on all operating system platforms, making it easy for the developer to switch back and forth between different platforms. And finally, although no developer would ever openly admit it, it's also kind of cooler, nerdier to work on the console command line than to click together your commands on a graphical user interface, just like any "normal" user.

Many professional developers use a computer with *MacOS* or *Linux* as the operating system instead of a Windows-based system. Asked about the reasons, many say that these two operating systems offer the better console, so they can work more comfortably on these systems. It's no wonder then that Microsoft, whose strategic thrust is explicitly to cater especially to the needs of developers, has released a modern console app in the summer of 2019 that aims to put Windows operating systems back into the competition for developers' favor.

? 12.1

Explain two ways in which graphical user interfaces can be developed.

12

? 12.2

Explain the basic difference in program flow between console applications and those with a graphical user interface.

? 12.3

List two advantages of console applications over graphical user interface applications.

12.3 Working with Files

■ The Fastest Way: Immediate Reading and Writing of Files

Data can not only be entered by the user but can also come from files. Working with files is basically always the same and very simply structured:

- The file in question is opened
- Their content is read out (or content is written into them)
- The file is closed.

Some programming languages have functions that allow you to edit files directly without having to explicitly worry about opening and closing them. Here are three examples. First an example from F# (“F Sharp”):

```
File.WriteAllText("test.txt", "A sample text written directly")
```

The same in PHP:

```
file_put_contents("test.txt", "A sample text written directly")
```

And finally, in R:

```
cat("A sample text, written directly ", file = "test.txt",  
    append = TRUE)
```

In all three cases, the text “A sample text, written directly” is written to a file named **test.txt**. The file does not have to be explicitly opened or closed; the functions used take care of that without our intervention.

■ Modes of File Processing

However, all of these languages also have functions for executing the above-mentioned three steps in working with files independently of each other. If, for example, you want to write data to or read data from the same file several times in a row, it would be inefficient to open and close the file each time. In this case, one would certainly not work with the functions just considered, but would first open the file, then write to or read from it several times, and only close the file again after all work has been completed.

The write operations we saw in the examples above in F#, PHP, and R create a new file if the file named **test.txt** does not exist, and then write our sample text to that new file. But if the file already exists, its contents are simply overwritten, without prompting or confirmation. Only in the third example, the example in the statistics language R, is the option **append=TRUE** used to control that existing file content is not overwritten, but that our text is appended to the existing content.

Thus, writing is not the same as writing either. In fact, there are basically three modes in which files can be opened:

- Read (mostly **read** or **r**)
- Writing (mostly **write** or **w**)
- Append (mostly **append** or **a**)

The modes “Write” and “Append” regularly create a new file if no file with the desired name exists yet. They differ, however, in the way they handle existing file

contents. In “Write” mode, these are usually replaced completely. In many programming languages, mixed modes are also possible, such as **ra** (read = **r** and append = **a**).

Incidentally, for historical reasons and the way file systems have always worked, there is no “insert” mode. So, you normally can’t just open a file, go to a specific location within the file, and then just insert additional content there. Instead, although it’s awkward, you have to rebuild the contents of the file in a variable in your program, that is, read the existing first part of the file into a variable (such as a string variable), then append the contents to be inserted to that variable, and finally read the back part of the file and append that to the new contents variable as well. Afterwards you can open the file in write mode and write the content of the variable into the file. This way you have completely overwritten the old file content, but in fact, you have only inserted something.

The situation is different when reading: Here you can regularly move the “file pointer”, which indicates where the next operation in the file should take place, to any position in the file. When opening a file in write mode, on the other hand, the file pointer is automatically set to the beginning of the file (and cannot be moved away from there, at least not without actually writing something), and when opening it in append mode, it is set to the end of the file.

Besides the type of editing operation to be performed on the file, the modes for opening files also differ with respect to another characteristic, namely whether the edited files are *text* or *binary files*. The difference between the two is most obvious with numbers. The number 32,000 is 0111110100000000 in binary terms, i.e., a 16-bit sequence of zeros and ones. Two bytes (8 bits each) are sufficient to store this value. In a text file, however, the number 32,000 would be considered text. So, if it is stored, the individual characters, “3”, “2”, “0”, “0” and again “0” are stored. The resulting file would then be 5 bytes in size. Differences between text and binary files also exist in the encoding of line breaks and the signaling of the end of the file, for which text files have a special character, the *EOF* character (end of file). Text files are usually such that you see letters and numbers when you open them. For example, program code that you write is stored in a text file. On the other hand, if you open, say, a PDF file or an executable program file with a text editor, all you will see is a seemingly random pattern of strange special characters; these are binary files that your text editor is trying to represent as text. To deal with the two different basic types of files, many programming languages have separate modes for writing, appending, and reading text files and binary files.

Now let’s take a closer look at opening, editing and closing files. In general, these operations can be represented in our pseudo-code like this:

```
my_file = open("test.txt", "w")
write(my_file, "First sample text written to the file.")
write(my_file, "Another sample text.")
close(my_file)
```

In most languages, when a file is opened, a variable of a special type is returned. In our pseudo-code, we get back a variable **my_file** from the **open()** function we use to open the file **test.txt** in write mode (**w**). From now on, we work with this variable so that the functions we use to edit the file know which file our instructions refer to; after all, we could have a whole series of different files open in parallel.

In object-oriented programming languages, the file variable will regularly be an object, which then has special methods that can be used to manipulate the file (flip back a few pages to the last chapter if the topic of objects and methods seems unfamiliar). Then writing and closing the file might look something like this:

```
my_file = open("test.txt", "w")
my_file.write("Another sample text.")
my_file.close()
```

■ Examples in Different Programming Languages

Let's look at this in some actual existing programming languages.

Here is an example in C:

```
#include <stdio.h>

int main() {
    FILE *my_file;
    my_file = fopen("test.txt", "w");
    fprintf(my_file, "First sample text written to file.");
    fprintf(my_file, "Another sample text.");
    fclose(my_file);
}
```

The code looks considerably more complicated than it actually is. To make it work, the first statement must include a special standard program library called **stdio.h**, which provides the functions for input and output. The main program in C is itself a function, **main()**, which is called automatically when the program is executed. What should happen then is inside the function and is what we are actually interested in here. In C, variables must be registered before they are used for the first time. Here, a variable of type **FILE** is declared, and the asterisk is used to create a *pointer* to such a variable, but we won't concern ourselves with the implications at this point. The variable just created then takes the return value of the function **fopen()** (for *file open*), which we use to open file **test.txt** in write mode (**w**). If this file does not exist yet, it will be created. After that we write with the function **fprintf()** (for *file print formatted*) into the just opened file, which we reference with the help of our variable **my_file**. Afterwards we close the file with **fclose()**.

The same process would look like this in Pascal:

```
program FileWrite;
var
    my_file: TextFile;
begin
    AssignFile(my_file, 'test.txt');
    rewrite(my_file);
    write(my_file, 'First sample text to be written to the
        file. ');
    write(my_file, 'Another sample text. ');
    CloseFile(my_file);
end.
```

Here we first declare a variable **my_file**. This variable is then assigned the reference to our file **test.txt** with a special function called **AssignFile()**. Up to this point, it is not at all clear in which mode the file is to be opened. That is not determined until the **rewrite()** function is called, which opens the file for writing. After writing, the file is closed at the end of the program with **CloseFile()**.

So Pascal differs from C in that here the special function **rewrite()** is used when opening the file in write mode, whereas in C the general function **fopen()** is used, where a function argument, i.e. a setting we pass to the function when calling it, determines in which mode the file should be opened.

Analogously, there is a special function in Pascal for opening files in read mode: **reset()**. If the first line from file **test.txt** is to be read, the corresponding Pascal code would look like this:

12

```
program FileWrite;
var
    my_file: TextFile;
    first_line: string;
begin
    AssignFile(my_file, 'test.txt');
    reset(my_file);
    readln(my_file, first_line);
    CloseFile(my_file);
end.
```

The function **readln()** (*read line*) is used, which reads a line from the file opened in read mode (first argument of the function) and stores it in the variable **first_line** (second argument). After that, the file pointer is automatically advanced to the next line, even without our intervention. If we were to read in a line again, this time it would be the second line.

■ Work with Files Beyond Reading and Writing Local Files

The files that are read do not necessarily have to be located on your local system. Provided that the files are not subject to any conflicting access protection, an Internet address (or *URL*, a *Uniform Resource Locator*) can regularly be specified

as the file name in most programming languages. Files that are located on a web server, and that your browser reads and displays as a web page, can be read using a program you have written yourself just as if it were a file on your local computer.

In addition to the functions for opening, reading, writing and closing files, many programming languages offer functions for working with the file system, such as creating or reading directories in the file system, or copying, moving, deleting or renaming files and directories. Functions for determining the file size or for checking whether a file exists at a certain path are also part of the standard scope of programming languages. Checking whether a file exists is useful to avoid an error or even uncontrolled program crash if your code tries to access a file that does not exist.

Of course, the approaches discussed here represent only the basic functionality that practically all programming languages have, even if they may differ slightly from each other in the individual design. In addition, many programming languages offer—either out-of-the-box or through extension libraries—many other functions with which files can be processed. For example, to write or read files in special file formats (such as, image files or files in the proprietary formats of certain popular software applications like Microsoft Office), or to exchange files with servers via appropriate network protocols such as FTP (File Transfer Protocol).

12.4

Explain the differences between the various options available when opening a file.

12.4 Working with Databases

In the practice of professional software development, working with databases plays a major role. Most web services as we know them today are basically the combination of a database ("backend") and a web user interface ("frontend") that allows the user to interact with the data stored in the database. This is as true for web shops as it is for social networks. In the professional environment, reading from and writing to databases is therefore the rule rather than the exception when it comes to data input and output. In the field of non-professional software development, working with databases certainly does not play quite such a large role. Therefore, and because the subject is not a simple one, we will only cover some basics in this section, which will give you an overview of the subject, and leave the details to more advanced programming courses.

Most databases are collections of data tables that can be related to each other. They are called *relational* because the tables ultimately describe relations (relationships) between certain objects and their properties. As a stylized example, look at  Tables 12.1, 12.2, and 12.3.  Table 12.1 contains the data of customers. The individual customers in the rows (in the context of the database, we speak of *data records*) can be uniquely identified via the column (in database language: the *field*) **CUSTOMERID**.  Table 12.2 shows the products. These also have some descriptive fields as well as a unique ID, the field **PRODUCTID**.  Table 12.3 represents the orders that the customers have placed. In this table, an order is identified by its own ID (**ORDERID**), the customer, the products that the customer ordered, and

■ **Table 12.1** Example data for table CUSTOMER

Custom- erid	Name	First name	Email	Street	City	Zip- code
C00001	Collins	Ethan	ecollins@ my-emails.com	123 Oak Street, Apt 4B	New York, NY	10001
C00002	Rodri- guez	Olivia	me@oliviardri- guez.com	456 Pine Avenue, Suite 9C,	Chicago, IL	90001
C00003	Thomas	Mason	mason.thomas@ thomas-winery. com	789 Maple Drive	Los Angeles, CA	60601
C00004	Wilson	Anna	anna@thewilson- family.com	321 Elm Street, Apartment 7D	Houston, TX	77002

■ **Table 12.2** Example data for table PRODUCT

Product ID	Description	Price
P001	Garden shovel, stainless steel	10.99
P002	Balcony table, plastic, green	24.99

■ **Table 12.3** Example data for table ORDER

Order ID	Customer ID	Product ID	Date
B0001	C00001	P001	2019-06-01 0:08 AM
B0002	C00003	P001	2018-12-05 7:49 AM
B0002	C00003	P002	2018-12-05 7:49 AM
B0003	C00002	P001	2019-05-16 9:58 PM

the order date. You can see that customers and products are represented by their IDs, and thus by *references to* the **CUSTOMER** and **PRODUCT** tables, respectively. Each row represents a combination of customer and ordered product. For example, you can see from purchase order **B0002** that customer **C00003** (Mason Thomas) ordered *two* products (namely **P001**—the garden shovel, and **P002**—the balcony table). Accordingly, in the **ORDER** assignment table, this order is described by *two* records.

Of course, we could have repeated all the customer and product data in the **ORDER** table; however, this would not only make the table unnecessarily large, but would also make data maintenance much more difficult, because changes to a customer's master data (for example, his address) would then have to be made in two places in the database, the **CUSTOMER** table and the **ORDER** table. This would increase the risk of inconsistent data. These problems are avoided by the reference technique used here.

The procedure of defining the essential objects only once and mapping relationships between them by means of references to their IDs, or keys, is referred to as *normalization*. The database system ensures that these references always work, so that a customer who is referenced in the **ORDER** table cannot be deleted from the **CUSTOMER** table, because that would make its existing order data inconsistent.

By the way, our example tables here are not yet perfectly normalized. The order date is actually attached to the order itself, not to the individual combinations of customer and product that belong to an order and are stored in the **ORDER** table. Complete normalization could be achieved by removing the products from the **ORDER** table (leaving only the ID of the order, the customer and the date in **ORDER**) and storing the assignment between the order and the ordered products via their keys in a separate table.

Data stored in databases in this way can be queried. A special programming language is used for this purpose—*SQL*, or *Structured Query Language*. Queries can be formulated in this language and processed by the database system which then returns the queried data as a result. The most important SQL statement is **SELECT**. The syntax of **SELECT** is basically quite simple (although it can be extended in many places to create more complex queries): **SELECT *fields* FROM *table* WHERE *condition***.

The statement

```
SELECT FIRSTNAME, NAME FROM CUSTOMER WHERE LOCATION = 'New  
York, NY'
```

queries all data records from the table **CUSTOMER** where the field **CITY** has the value "**New York, NY**". However, not all fields are to be displayed, but only the fields **FIRSTNAME** and **NAME**. The result of this statement is therefore a table that contains only these two fields for the relevant data records, in the example as shown in ■ Table 12.4. In the case of our **SELECT** statement, the results table only consists of one record, that of Ethan Collins.

However, SQL can not only be used to query data from databases, but also to write data to databases. For this purpose, the statements **INSERT** and **UPDATE**

■ Table 12.4 Result of our SQL sample query with **SELECT**

First name	Name
Ethan	Collins

are used, which insert *new* records into a table (**INSERT**) or update *existing* records (**UPDATE**), i.e. change the value of one or more fields of a record (or several records at the same time).

From most programming languages, databases can be addressed (often with the help of extension libraries, covered in the following chapter), fed with SQL statements and their results received and processed.

12.5 Your Roadmap to Learning a New Programming Language

If you are learning a new programming language ...
you'll discover:

- which types of user interfaces (GUIs, console) you can equip your programs with,
- if console applications are supported, which functions are available for the output of data as well as for the input of data by the user,
- if graphical interfaces are supported, how they are designed (graphical WYSIWYG development in IDE, description of the interface in the code) and which controls are available to you for the design of your graphical user interface and which essential properties/configuration options they offer,
- how to open (and close) files for reading, writing or appending data, in particular what different editing modes are available when opening and how to distinguish between opening text and binary files,
- what functions are available to write data into or read data from open files,
- if you plan to work with databases, how to connect to databases as well as send SQL statements to the database and process the returned results.

12.6 Solutions to the Exercises

■ Exercise 12.1

Graphical user interfaces can be designed via an Integrated Development Environment (IDE) in WYSIWYG mode, where the individual control elements are selected and placed on the program interface with the mouse. Typically, the properties of the controls can be conveniently. The user interface is thus created without having to “program” it. But many programming languages also allow the user interface to be created as part of the program code. The individual controls are then mostly objects that can be created from within the program—just like other objects—and placed on the interface. The properties of the controls can be accessed via the attributes of the objects representing them.

■ Task 12.2

Console applications usually run in a completely linear fashion where the user is guided through the program, and the program determines the sequence of the

user's work. In applications with graphical interfaces, the user usually has more control over the order in which he or she executes which steps. For example, the user can click on buttons in any order. In this sense, the program does not determine user behavior; instead, it responds to *events* that the user triggers. For example, if the user clicks on a button, the associated event is triggered, and the program code intended for that event is executed. Although this type of event-oriented program control can, in principle, be reproduced in console applications, it is much less common there than in applications with graphical user interfaces.

■ Exercise 12.3

A console application is usually fast and easy to use if you are familiar with it. The behavior of the program can often be controlled by command line parameters when it is called from the console. The usability is also facilitated by the fact that all control operations can be performed with the keyboard, whereas graphical user interfaces often require a pointing instrument (usually the mouse) and thus the switching of the hand or hands between the input devices. Console applications also have advantages in development because they can usually be programmed with less effort than applications with a graphical user interface. In addition, a lower level of knowledge of the programming language used is usually sufficient for the development of a console application than for the development of a graphical user interface.

■ Exercise 12.4

In most programming languages, the modes for opening a file differ first by the editing operations that are allowed in each case. The file can be opened regularly for reading (**read/r**), writing (**write/w**) or appending (**append/a**). When opened in write mode, an existing file is completely overwritten. If data is to be appended to an existing file, it must be opened in append mode. A second difference is whether the file is opened as a text file or as a binary file, which mainly affects the way data written into it is encoded.



How Do I Work with Program Functions to Edit Data and Trigger Actions?

Contents

- 13.1 Functions – 138
- 13.2 Libraries – 145
- 13.3 Frameworks – 148
- 13.4 Application Programming Interfaces (APIs) – 149
- 13.5 Your Roadmap to Learning a New Programming Language – 150
- 13.6 Solutions to the Exercises – 151

Overview

In most programming languages, small subroutines, so-called functions, do the actual job. They help to read in data, change it, output it again and much more. You can access the functions that your programming language offers as standard, install additional libraries with functions from the Internet or write your own functions. That's what this chapter is about.

In this chapter you will learn the following:

- What functions are
- How to define functions in the program code
- How to call functions and specify their behavior
- What role the scope of variables plays when you work with functions
- How to use functions in object-oriented programming and the advantages of functions being objects themselves in many languages
- How functions are grouped in libraries and how you can find suitable, freely available libraries
- What frameworks are and how they differ from libraries
- What Application Programming Interfaces (APIs) are and how you can work with them.

13.1 Functions

■ What Are Functions?

As in mathematics (don't worry, we won't overuse this comparison!), functions are mapping rules that map values, called *arguments*, to another value, their *function* or *return value*. For example, the function $f(x) = x^2$ maps a value x to its square; thus, one passes a value x to the function as an argument and receives a processed value (in this case, the square of the argument) back from the function as a return value.

Functions in programming languages work according to the same basic principle, with the difference, however, that not all functions produce a return value. We have already worked in the pseudo code ► Sect. 11.7.5 with a function called **show-Product()**, which performs an output on the screen. This function does not yield a return value. It simply performs a specific action, namely displaying on the screen, but otherwise does not process the data passed to it as an argument any further. Some programming languages make a strict distinction between functions that have a return value and *procedures* that do not. However, we will not make this distinction here and will always refer to “functions” instead. In some programming languages, functions always return a value; if they do not have a “real” return value, they return a special value that signals that there is no “real” result (for example, **undefined** in JavaScript or **void** in C/C++).

■ Definition of a Function

Functions usually consist of a *function head* and a *function body*. The header regularly contains the function identifier (name) and the list of arguments that the function expects. The body is a block of code that is executed whenever the function is called. It contains the “meat” of the function; the source code it contains describes what the function does.

Consider the pseudo-code example of a simple function that multiplies two numbers together:

```
Function multiply(number1, number2)
Begin
    result = number1 * number2
    return(result)
End
```

To let the programming language know that the definition of a function begins, it starts with the keyword **Function**. This is followed by the function identifier and the two arguments of the function, **number1** and **number2**. The function body consists of two statements, one that calculates the actual result of the function, and a call to another, **return()**, that returns the function result. The two instructions form a *code block*, which is introduced with the keyword **Begin** and concluded with the keyword **End**. Code blocks exist in practically all programming languages. They are usually delimited, as in our pseudo-code, by special keywords (the English keywords **Begin** and **End** are particularly common) or symbols, such as opening and closing curly braces (**{** and **}**). Some languages, such as Python, mark a block of code without any special keywords or symbols at all, solely by indenting all lines of code in the block evenly.

But back to our function definition. Its program code does nothing per se. The function only appears when it is called from within the program. In our example, this could look like this:

```
show(multiply(3, 57.8))
```

With this call, we multiply the numbers 3 and 57.8 and immediately output the result. What happens here is that the execution of the program branches to the definition of the **multiply** function. The code contained in the body of the function definition is executed, with the function arguments **number1** and **number2** taking the value of the factors passed – in this case, 3 and 57.8. The result of the multiplication is returned with **return()**, which causes the program execution to leave the function body and return to the main program. By executing the function, the return value of the function now takes the place of the function call. This can now be passed again as an argument to another function, in our example **show()**. After passing through our **multiply()** function, the program code effectively shortens to:

```
show(173.4)
```

Because the function call is replaced by the return value after its execution, function calls can also be assigned to variables:

```
value = multiply(3, 57.8)
```

Many programming languages require that the function has been defined *before* the first call, so the function definition must be “higher up” in the program code than the first call.

Note: Functions can also be defined without arguments. For example, a function that simply clears the display on the screen does not need any further information passed to it. In most programming languages, even such functions must be called with the round brackets in which the values of the arguments are normally found. In this case, the parentheses are empty, but the interpreter/compiler of the language still recognizes that this is a function call and not an access to a variable with the same name.

The sequence of a function call from the program is shown schematically in

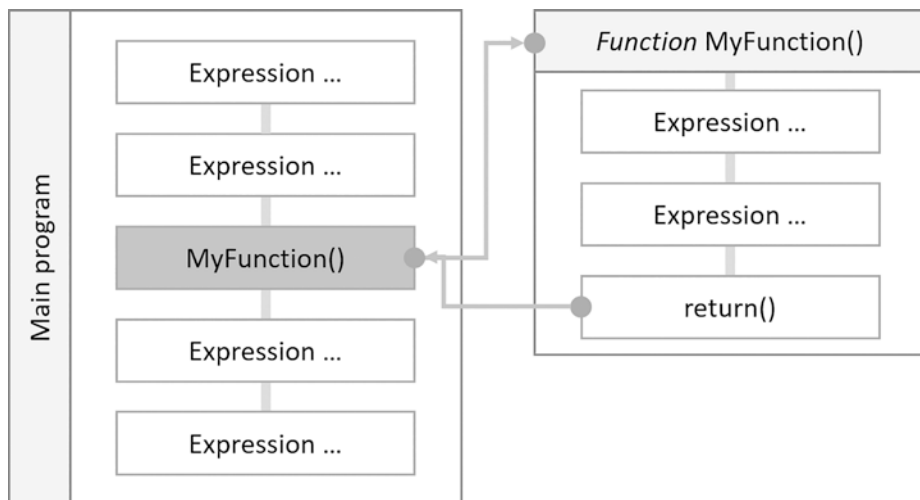
■ Fig. 13.1.

■ Optional Arguments

Sometimes you want to give the user the possibility to control the behavior of the function via a parameter but preset it with a default value. If the caller of the function then does not specify a value for the argument in question, the default value is used.

Suppose we wanted to design our **multiply()** function from above so that **number1** is always multiplied by the number pi (3.14159...) unless a value for the argu-

13



■ Fig. 13.1 Calling a function

ment **number2** is explicitly passed in the function call. Then we would have to specify this default value in the function header: **Function multiply(number1, number2 = 3.14159)**. Now a call to **multiply** could also look like this:

```
two_pi = multiply(2)
```

In this context, **number2** is called an *optional* argument, because it can be omitted when calling the function.

■ Passing Function Arguments with Their Identifier

In many programming languages, the arguments when calling functions can also be addressed via their identifiers. This could then look like this:

```
value = multiply(number1 = 3, number2 = 57.8)
```

The advantage of these *keyword arguments* is that it becomes clearer (especially when calling functions with many arguments) which value belongs to which argument; thus, one does not have to keep the order of the arguments in mind. Because now the identifiers of the arguments are available, the interpreter/compiler is no longer dependent on the order of the arguments. In this latter case, we speak of *positional arguments* to distinguish them from keyword arguments. Accordingly, you can also deviate from the actual order of the arguments; a valid function call would thus also be the following:

```
value = multiply(number2 = 57.8, number1 = 3)
```

An additional advantage is that by using argument identifiers when calling functions the program code becomes easier to understand and thus easier to maintain.

■ Scope of Variables

In most programming languages, functions lead a “life of their own”, so to speak. They are a compartmentalized world all to themselves. Think back to the definition of our **multiply()** function. Inside this function, a variable called **result** is created to hold the calculated product of the two numbers. This variable exists *only within* the function. At the end of the function call, i.e., when the function has completely run, the variable **result** also ceases to exist. It cannot be accessed from the main program. It is therefore also said that its *scope* is limited to the function body.

Suppose our main program looked like this:

```
value = multiply(3, 57.8)
show(result)
```

The statement **show(result)** would result in an error message (or, depending on the language, return a default value like **0**), because there is no variable named **result** in the scope of the main program. This variable is in the scope of the **multiply()** function. Because its scope is limited, **result** is said to be a *local variable*. The variable **value**, on the other hand, is created in the main program and is valid everywhere. Such variables are called *global variables* in distinction to local variables. They can be accessed from anywhere, even from within our **multiply()** function. Best practice, however, is to avoid such access to global variables from within a function (also called *side effects*), because it makes the function more susceptible to changes in the rest of the program, especially to changes in the identifiers of global variables.

But what if the **multiply()** function itself were to use a variable called **value**, for example, if the central instruction of the function were **value = number1 * number2**? Then we would have two variables called **value**, a local one (inside the **multiply()** function) and the global one in the main program. Now, which one is accessed when the identifier **value** is used somewhere in the program code? As a rule, programming languages look first in the local environment for a variable with this identifier and only then, if they do not find one there, in the global environment. So if the variable **value** were used within the function **multiply()**, the local variable would be used, the one that has been defined in the current scope. An access to **value** from the main program, on the other hand, leads to an access to the global variable.

Function arguments usually behave like local variables, they can also only be accessed from within the function's code.

■ Passing Arguments by Reference Instead of by Value

Consider the following slightly modified variant of our **multiply()** function:

```
Function multiply(number1, number2, AsReference result)
Begin
    result = number1 * number2
End
```

This time, our function has no return value; instead, the result of the calculation is assigned to a variable that was also passed as an argument, but, unlike **number1** and **number2**, with the keyword **AsReference**. Normally, the arguments to a function are passed as copies of the original values. Consider the following example of a call to our function:

```
number1 = 3
value = 0
multiply(number1, 57.8, value)
show(value)
```

Here the first argument of the function itself is a variable. Its value is available as argument **number1** inside the **multiply()** function, but only as a copy of the original variable. If we would now adjust the value of **number1** within the function body, this would have no effect on the variable **number1** in the main program. Only the local variable **number1** in the function body of **multiply()** would change in value.

This is different in the case of the variable **value**. This is not passed to the function as a value, but as a so-called *reference*. This means that changes made to this variable within the function affect the original variable; in the example, we use this variable to “return” the result of the calculation. Some programming languages allow variables to be passed *by value* or *by reference*, some only allow them to be passed by value.

■ Examples for the Definition of Functions

So far, we have only written our function definition in our pseudo-code. Therefore, here are two implementations of the original **multiply()** function in real programming languages, namely PHP and Pascal.

First the PHP version:

```
function multiply($number1, $number2)
{
    $result = $number1 * $number2;
    return $result;
}
```

The statement **return** is, as in some languages, not a function but a keyword (hence no round brackets around the “argument”).

Then the code same in Pascal:

```
function multiply(number1, number2: real): real;
var
    result: real;
begin
    result = number1 * number2;
    multiply = result;
end;
```

Pascal is a strongly typed programming language. Variables are always declared with a fixed type here. Therefore, both the two arguments of the function, **number1** and **number2**, as well as the function itself (at the end of the function header) each have a type specification: In all cases, they are floating-point numbers (**real**). This makes it clear not only what kind of arguments the function expects, but also what type its return value will be.

The return statement does not work in Pascal as in many other programming languages with a keyword **return** or a function **return()**, but is achieved by assign-

ing the function value to the identifier of the function (assignments are formulated in Pascal with = as assignment operator).

■ Functions in Object-Oriented Programming

In ► Sect. 11.7.4, we saw that, in the context of object-oriented programming, functions can also be part of classes. As you will remember, such functions are also called *methods*. These methods are always called in connection with a concrete object instance of the class, i.e., an object that has been created according to the “template” of the class. Most programming languages use the period (but some use other symbols, such as ->) to separate object and method when calling them. The call of a method could therefore look like this, for example:

```
object.method(arguments)
```

Depending on the programming language, the method definition may be part of the class definition or outside of it. In any case, however, there is a reference to the method in the class definition (often the method header), as we saw in ► Sect. 11.7.4 (scroll back a few pages if necessary).

Apart from the fact that functions as methods can be part of objects (or their classes), functions and objects often have another relationship. In many programming languages, functions *are* objects. There, a special class exists (often **function**) and all functions are objects (i.e., instances) of this class. As such, they may also have properties, such as their arguments or their function body, i.e., the actual code. When functions are themselves objects, this has some remarkable implications. For example, functions themselves can serve as arguments to other functions. Moreover, class definitions are logically very stringent, strictly speaking, they consist exclusively of attributes. The only difference is that some attributes, namely the methods, are *callable* and others (the “normal” attributes in the sense of value properties) are not.

13

? 13.1 [3 min]

What is wrong with the following function definitions (written in pseudo-code):

```
a.
Function exponential(base)
Start
    return(base^exponent)
End

b.
Function exponential(base, exponent)
Begin
    result = base^exponent
End
```

? 13.2 [3 min]

Suppose we had a function defined as follows:

```
Function introduce(name, age)
Begin
    show("My name is ", name, " , I am ", age, " years old.")
End
```

What is the difference between the following calls to the function, and why do the first and third calls result in the desired behavior of the function, but the second does not?

```
introduce("Mary", 25)
introduce(25, "Mary")
introduce(age = 25, name = "Mary")
```

? 13.3 [5 min]

Consider the following program excerpt:

```
age_person = 25

Function getolder(age_person)
Begin
    age_person = age_person + 1
    show("New age of the person: ", age_person)
    return(age_person)
End

age_new = getolder(age_person)
show("Current age: ", age_person)
show("Result of the function getolder(): ", age_new)
```

- What output does this program generate and why?
- What two ways are there to modify the program so that the new age, which is one year higher, is also reflected in the global variable `age_person`?

13.2 Libraries

■ Libraries as a Toolbox for Programmers

As a programmer, you can (as seen in the previous section) develop functions yourself. This always makes sense if you want to *reuse* program code, because the nice

thing about functions is that they allow you to detach a certain functionality from your actual program and call it from anywhere at any time.

Of course, programming languages regularly come with an extensive set of standard functions that you can use to perform many common tasks. Often, however, these standard functions will not be sufficient for what you have in mind. For example, functions for sending e-mails, searching web pages (web scraping), or training neural networks are usually not included in the standard language scope. In these and other use cases, if you do not want to develop the functionality in question yourself, you will have to extend the language scope by installing the required functions from elsewhere.

■ Find Suitable Libraries

Some programming languages such as Python, R and JavaScript have a very extensive and lively “ecosystem” with a large number of developers who make the functionalities they have developed available to others free of charge. This is regularly done in the form of a set of function focused on a certain topic. Such collections are usually called *libraries*, *modules*, *packages*, or *distributions*, depending on the programming language. For the sake of simplicity, we always refer here to “libraries”. In some cases, these libraries are brought together on a platform that is centrally managed by the organization that oversees the development of the programming language.

This is the case, for example, for:

- Python with the *Python Package Index* (PyPI, ► <https://pypi.org/>),
- Perl with the *Comprehensive Perl Archive Network* (CPAN, ► <https://www.cpan.org/>),
- PHP with the *PHP Extension Community Library* (PECL, ► <https://pecl.php.net/>),
- R with the *Comprehensive R Archive Network* (CRAN, ► <https://cran.r-project.org>).

13

Depending on the design, inclusion in one of these central catalogues is also accompanied by (mostly automated) quality control, which ensures, for example, that the program code of the library is executable and has at least a minimum of documentation.

For many programming languages, however, such a centrally organized library platform does not exist. Often, developers then deposit their works on *GitHub*. *GitHub* is a cross-language platform that allows developers to share source code with others by storing it in a *repository*. Underneath *GitHub* lies the version management tool *Git*, developed by Linux inventor *Linus Torvalds*, which allows changes to source code to be versioned in an elegant way. If necessary, it is then easy to revert to an old state of a single code file or even the entire repository. Developers can also use *GitHub* to create their own *branches* of the source code, develop features in them, and eventually *merge* them back into the original version, at least if the developer of the original version allows it. With these and a whole series of other features, *GitHub* facilitates collaboration between different developers on the basis of the underlying version management tool *Git*. *GitHub* is com-

pletely language agnostic; for most programming languages in use today, you will find repositories with code in that language. For developers who offer their source code under an open-source license, using *GitHub* is free at the time of writing this book. Companies and organizations that want to shield their code from the outside world pay for a private environment. While a number of comparable platforms exist alongside *GitHub*, *GitHub* is the most popular.

Of course, *GitHub* is not only used by those developers who work in a language without a central library platform. Many developers run *GitHub* repositories not for libraries but simply for their normal work on any kind of development projects, and to exchange information with other developers. If they do indeed develop libraries, they make only the finished versions available on the central library platforms such as *PyPI* or *CRAN*. Sometimes developers also don't want to subject themselves to the strict rules and automated quality checks of the central library platforms and therefore offer their libraries exclusively on *GitHub*.

It's no surprise, then, that the central library platforms and *GitHub* are both good places to start if you're looking for a library to help you solve a particular problem. That said, this search is sometimes not entirely easy, for several reasons. The search functionalities of the platforms vary, as does the information available about each library. So, it is sometimes not at all easy to find a library, and once you have found one, to judge whether it is suitable for solving your own problem. This assessment is further complicated by the fact that there will often be more than one library that might be a hit. Even on the centrally administered library platforms, there are several, to a certain extent, competing libraries for many tasks. Two approaches are then recommended. One is *trial and error*; download and work with the "candidates" to determine which is the most appropriate library for your problem. Alternately, search on the internet, where you can regularly find information on many libraries in relevant forums (such as *Stack Overflow*). This is especially useful if you want to see how to use the library by means of examples. The documentation of the libraries varies in scope and quality, so that forums like *Stack Overflow* can also be a good help in using a library that has already been found suitable.

Apart from that, forums such as *Stack Overflow* are often a good first port of call for identifying library candidates for the solution of your problem in the first place – if you cannot or do not want to search directly via a central library platform or on *GitHub*. This is because the questions in the forums are usually structured along the lines of "How do I solve the problem that..." or "How do I make it so that..." and then the answers often contain references to one or more libraries that can help solve this kind of problem. Many developers prefer this type of forum search to searching directly on central library platforms or *GitHub*, because the forums with examples and comments from users often provide valuable additional information.

Needless to say, you can also use AI assistants like ChatGPT to search for suitable libraries. For example, ask: "I want to write a Python program that reads data from a website (webscraping). Which libraries are available for this, how do they differ and which are particularly suitable for beginners?"

■ Installing and Importing/Loading Libraries

Once you have found a suitable library, all you have to do is install it (if the programming language provides for this or requires it) and include it in the program code. The inclusion is done by a statement which either imports the library as a whole or individual elements from it (such as individual functions or classes) into the program code. In some cases, a name can be specified under which the imported element can be addressed, which is useful to avoid conflicts with the identifiers of already existing variables, classes, objects, or functions.

A (hypothetical) example from Python:

```
from somelibrary import aclass as my_class
import somelibrary
```

The first statement imports only the class **aclass** into the program code and then makes it accessible under the identifier **my_class**. The second, alternative statement imports the entire library (in this case without assigning it a different identifier).

It is easier in Pascal

```
uses somelibrary;
```

or R

```
library(somelibrary)
```

where the entire library is made available in each case.

13.3 Frameworks

There is a lot of talk about *frameworks*, especially in the field of web development. The terms “framework” and “library” are sometimes not used in a very clear-cut way. Even though the functionality and use of frameworks goes beyond what we will look at in this book, at least the distinction between the two concepts should be examined in more detail. Libraries contain functionalities that the user, i.e., the programmer, can access when he needs them to perform a specific task. It is therefore the programmer who takes the initiative. Frameworks, though, are different. Frameworks control the flow of the entire application and call the programmer’s code when necessary. Frameworks thus form, as the term already suggests, a framework that the programmer “only” has to fill. This is very practical, especially in the field of web development, because there are many recurring tasks, such as authentication of user logins, the connection of databases, or the output of data in template-like pages. All these tasks can be taken over from the developer by the

framework. They are already built in as functionalities in the framework the developer is working with. Programmers only need to develop what is specific to their applications. Through this division of labor between the framework and the developer, the control of the application is practically reversed (this is why, in connection with frameworks, some people also speak of *inversion of control*): The framework controls the application, the developer supplies the application-specific program code, which the framework then calls at the right place. This allows the developer to concentrate on the important issues and leave all the rather boring “trappings” to the framework.

Well-known frameworks include *AngularJS* and *React* (for JavaScript), *django* (for Python), *CakePHP* and *Zen* (for PHP), and *Ruby on Rails* (for Ruby).

Because the focus of this book is on learning the basics of programming, we will not be working with frameworks here. However, a prerequisite for using frameworks is, of course, knowledge of the underlying language, and that’s what we’re dealing with here in the book.

13.4 Application Programming Interfaces (APIs)

Another term that is on everyone’s lips is that of the *Application Programming Interface*, or API for short. First of all, APIs are – as the name suggests – nothing more than *programming interfaces*. In this sense, a library that provides a set of functions for specific purposes also provides a programming interface that consists of the library’s functions that you can access from within your program.

When people talk about APIs today, they are often referring to specific programming interfaces, such as *Web APIs* through which you can access the functionality and data of web services. Many web services today offer programming interfaces for developers. For example, a *GoogleMaps API* provided by Google returns the geo-coordinates of a city if you pass the name of the city to the API. Here, an API is used to retrieve information provided by the web service. In the same way, however, APIs can also be used to get a web service to trigger a specific action. Twitter’s API, for example, allows you to post tweets. All you need is an account that allows you to use the API and a corresponding instruction in your program that addresses the API and sends the tweet.

So, APIs can be used to query data and trigger functions. There are literally thousands of web services that offer APIs which can be accessed via the Internet. The website *ProgrammableWeb* (► <https://www.programmableweb.com/apis/directory>) provides an overview of API providers that is certainly not complete, but at least very comprehensive. Whether you want to process payments with PayPal from your program or check the latest football results, with the web APIs available today, hardly any wish remains unfulfilled.

Technically, Web APIs mostly work with the *Hypertext Transfer Protocol* (HTTP) and can be called like web pages. For example, to access the *GoogleMaps API* to get the geo-location of Chicago, a program would simply need to make an HTTP request of the form ► <https://maps.googleapis.com/maps/api/geocode/json?address=Chicago&key=XX>, where **key** is a parameter that authenticates the

caller of the API as a legitimate user. Google would then return the coordinates, in JSON format, a data exchange format that is extremely popular with Web APIs because it is easy to generate and easy to read, and which we will learn about in more detail in ► Sect. 31.5.6. If you were to enter the request as it is into your web browser, you would also get a result in JSON format, but an error message, because **XX** is not a key that identifies you as the legitimate API user:

```
{
  "error_message" : "The provided API key is invalid.",
  "results" : [],
  "status" : "REQUEST_DENIED"
}
```

However, a program does not need a web browser to make such requests to a web service via an API. The corresponding HTTP requests can be triggered from within most programming languages (often using a library that provides the corresponding functions), and their returns can be processed without the user of the program having to know anything about it.

We won't continue discussing Web APIs in detail, because we're focusing on the basics of programming in this book. However, given these basics, it's easy to take the next step and learn exactly how to incorporate Web APIs into your programs. The most important thing to understand at this point is that web APIs are a versatile tool for extending the functionality of your programs. Ultimately, they work just like library functions; they are called (albeit slightly differently), trigger an action or retrieve information, and return a value (often a JSON object). The basic concept, which is that of a programming interface, or API, is the same in both cases.

13

13.5 Your Roadmap to Learning a New Programming Language

If you are learning a new programming language ...
you'll discover:

- how functions are defined; in particular, how optional arguments are used, how (and whether) arguments can be taken by reference instead of by value, and how functions can return values,
- how functions are called; in particular, how (and whether) arguments can be passed by name when calling a function,
- how the scopes of variables are designed and whether it is possible to access a global variable from within a function (especially if it has the same name as a local variable)
- what good reference sources for libraries exist and how to search for suitable libraries; in particular, whether a centrally administered platform exists, which is de facto the main point of contact when searching for suitable libraries,
- how to include libraries (or element from them) in your program and how to access their functionality.

13.6 Solutions to the Exercises

■ Exercise 13.1

- (a) In the return statement `return()` an argument **exponent** is used, but this argument does not appear in the argument list in the head of the function. The header should therefore correctly read **function exponential(base, exponent)**.
- (b) This time, the argument **exponent** is included in the argument list in the function header. However, the function does not return a value. A variable **result** is calculated, but it is not returned with the help of `return()`. The statement `return(result)` would have to be added as the last statement of the function.

■ Exercise 13.2

`introduce("Mary", 25)` calls the function as it is intended. The first argument in the argument list in the function header is the name, the second is the age. `introduce(25, "Mary")` flips the argument values around when the function is called. Now the first argument (**name**) is assigned the value **25**, the second argument (**age**) the value **"Mary"**. In the best case, the function then outputs a somewhat strange-looking text, but in the worst case, it terminates with an error because the expected types of the arguments and the passed values do not match. Finally, the call `introduce(age = 25, name = "Mary")` also flips the arguments around when the function is called, but by specifying the argument names it is clear how the passed values should be assigned to the arguments of the function, so the different order of the arguments when the function is called is not a problem.

■ Exercise 13.3

- (a) The program will produce the following output:

```
New age of the person: 26
Current age: 25
Result of the function getolder(): 26
```

The function `getolder()` increments the value of the argument **age_person** passed to it and returns this as the function value. The *global* variable of the same name, **age_person**, remains unaffected. Priority in the scope of the function is given to the argument, which is treated as a *local* variable.

- (b) The first possibility is to catch the return value of the function `getolder()` in the global variable **age_person**:

```
age_person = getolder(age_person)
```

Alternatively, the argument **age** of the function can be defined as an argument passed by *reference* (if the programming language used allows it). Then the function header would look like this: **Function getolder(AsReference age_person)**. This way, the **age_person** argument would still be considered a local variable (it could not be accessed outside of the function's code block), but changes to it would immediately result in changes to the variable passed in (i.e., **age**). Thus, the **getolder()** function would ultimately be able to change the global variable **age**.



How Do I Control the Program Flow and Make the Program React to User Actions and Other Events?

Contents

- 14.1 Why Program Flow Control Is Necessary – 154
- 14.2 Forms of Flow Control – 155
- 14.3 If-Else Constructs – 155
- 14.4 A Closer Look at Conditions – 161
- 14.5 Complex Conditions with Logical Operators (AND, OR, NOT) – 163
- 14.6 Checking Similar Conditions Efficiently with Switch-Case Constructs (Switch/Select...Case) – 166
- 14.7 Events – 168
- 14.8 Your Roadmap to Learning a New Programming Language – 172
- 14.9 Solutions to the Exercises – 173

Overview

Programs must be able to react flexibly to new situations, for example, when the user makes an entry or clicks on a button. Depending on the input made or which button is clicked, different program instructions are executed. In effect, the program branches off into a different path. This form of flow control makes programs dynamic.

In this chapter you will learn the following:

- How to branch to one or the other program part depending on a condition
- How to formulate such conditions and how to check whether they are fulfilled or not
- How to formulate complex conditions that are composed of several sub-conditions
- What possibilities there are to test a whole series of similarly structured conditions in a simple and clear way
- How you can react to events when you cannot know beforehand exactly when they will be triggered in the program flow (for example, the click on a button).

14.1 Why Program Flow Control Is Necessary

In the previous chapters you have seen what you need to receive data from the user, to process it, and to output the results of the processing. Nevertheless, programs that you write only with tools from that toolbox would be very limited in their possibilities, and thus in their usefulness, because the program flow would be completely rigid. It would *always* start with an input, which would then *always* be processed by the program in *exactly the same way*, and whose result would finally *always* be reported back to the user in *the same form*. In reality, it is unlikely that we would ever accept such inflexible programs. Imagine that you enter a new destination into the navigation system of your car. The navigation system calculates the optimal route to this destination, but as soon as you accidentally deviate from this route, the navigation system rigidly sticks to the original routing, even though you are no longer on that route. What you need is help to get back to the original course, or to find a completely new route, starting from your current location. The behavior of your navigation system would seem strange if you set it to avoid toll roads, but the system completely ignored this instruction and simply directed you straight onto the first toll motorway. You would, quite rightly, think this was a fault in the navigation system.

So, what we need is a program that reacts to events (“driver left suggested route”) and takes conditions into account (“driver wants to avoid toll roads”).

In both cases, certain parts of the program are only executed if the corresponding event has occurred, or the corresponding condition has been fulfilled. The program *branches* into different sections of the code, depending on the current situation and other relevant circumstances. This chapter deals with this type of program *flow control*.

14.2 Forms of Flow Control

In practice, the flow control of the program is usually realized by using *If-(then-)Else constructs*: *if* a certain condition is met, *then* do this, *otherwise* do something else.

The conditions checked can also be *complex conditions* that consist of several sub-conditions. In our navigation system example, such a complex condition could be: “If ‘Avoid toll roads’ is set *AND* the next turn leads onto a toll road, then do not recommend taking this turn under any circumstances”. Another example would be “If the next turnoff is not a toll road *OR* driving on toll roads is allowed, then recommend the next turnoff”. Here, two partial conditions are linked with each other by *AND* or *OR*, both of which are logical operators.

Sometimes you have a whole series of similar conditions to check, for example, if the user is to enter a digit and each digit causes a different reaction. The condition is structurally always the same, namely “input equals digit X” while the program code that is executed when a particular one of these conditions is met may be quite different in each case, depending on which digit was entered. Such use cases of conditions can be implemented using If-Else constructs, but these regularly lead to complicated code that is difficult to read (and thus difficult to maintain). Therefore, many programming languages know a *Switch-Case construct*, which can be used to implement the checking of similar conditions in a very simple and clear way.

The program flow is not always linear, so that after a well-defined sequence of instructions, you arrive at the point where a condition is checked and then, depending on the result of the check, you branch to one or the other part of the program. The most important cause for non-linear program flows is the user. On a graphical user interface, for example, the user can choose from a number of different program features and can decide relatively freely at which point to use which feature and in which order. This means that the user determines the flow of the program, and the program cannot simply be the stubborn processing of a long sequence of instructions but must somehow be structured more flexibly. This kind of flow control leads us to the concept of *events* and the event-oriented programming paradigm that is used in many programming languages.

We deal with these elements of flow control, the If-Else constructs, the conditions used in them, the linking of multiple sub-conditions into a more complex overall condition, the Switch-Case constructs, and event control in the remainder of this chapter.

14.3 If-Else Constructs

Imagine we were working on software for online banking, the programs that run a bank’s online banking website. Here we deal with an account balance query. The user should receive a warning if his account is in debit, i.e., the account balance is negative.

A program that does just that might look like this:

```
balance = query_balance()
If balance < 0 Then show("Attention: Your account balance is
    negative!")
show("Current balance: $", balance)
```

Here we first assign the value of the function **query_balance()** to a variable **balance**. We want to assume that this function returns the current account balance as a decimal number. After saving the account balance in the variable, we use **If balance < 0** to check whether the account balance is negative. If it is, a warning message is displayed, which we do here with the function **show()**. Then, with **show("Current balance: \$", balance)**, we output the account balance itself, i.e. the contents of the variable **balance**. Note that this statement is always executed, regardless of whether the account is in debit or not. The only statement that depends on the condition that the account balance has a negative value is the display of the warning message after the keyword **then**. Everything that follows will be executed in any case.

Now suppose that the account balance is \$1000. In this case, the output of our program would be:

```
Current balance: $1000
```

However, if the account were in debit, for example, with a balance of -\$280, the user would receive the following output:

```
Attention: Your account balance is negative!
Current balance: $-280
```

14

In this simple example you can see very clearly how the program branches and executes certain parts – in our case the warning – only if a certain condition is met.

In the next step, we extend the program so that it also displays how much is left of the overdraft facility, which we want to assume is \$500, provided the account is in debit. If the account is in credit, i.e., the account balance is greater than 0, no message is to be displayed with regard to the overdraft facility. The expansion of our program could look like this:

```
balance = query_balance()
If balance < 0 Then
  Begin
    show("Attention: Your account balance is negative!")
    overdraftrest = 500 + balance
    show("You still have $", overdraftrest, " left of your
        overdraft facility.")
  End
show("Current balance: $", balance)
```


To map the overdraft balance, we create a variable called **overdraftrest**, to which we assign the sum of \$500 and **balance**. If the account balance is negative (and only then will this part of the program be run), the sum is just the amount remaining on the overdraft facility.

Unlike the previous example, here the check to see if the account is in debit is followed by *several* statements enclosed in a code block between the **Start** and **End** keywords. We have already learned about code blocks in connection with functions in ► Sect. 13.1. All statements in the code block are executed only if the condition **balance < 0** is met. The statement **show("Current balance: \$", balance)**, with which we output the current account balance, is not part of the code block and is therefore always executed, regardless of whether the account is in debit or credit. In many programming languages, if the code block is only one line, the delimiters **Begin** and **End** in our pseudo-code) can also be omitted, as we did in the examples above.

Let's take our example again with a negative account balance of -\$280. In this case, our program now produces the following output:

```
Attention: Your account balance is negative!
You still have $220 left of your overdraft facility.
Current balance: $-280
```

In the case of a positive account balance of \$1000, the output is reduced accordingly:

Current balance: \$1000

Now suppose we also wanted to display something when the account balance is positive (or equal to 0), for example, "Your account is in credit."

This can be easily implemented with the means we have learned so far:

```
balance = query_balance()
If balance < 0 Then
  Begin
    show("Attention: Your account balance is negative!")
    overdraftrest = 500 + balance
    show("You still have $", overdraftrest, " left of your
      overdraft facility.")
  End
If balance >= 0 Then
  Begin
    show("Your account is in credit.")
  End
show("Current balance: $", balance)
```

However, this can be done more easily if we take into account that each of the conditions **balance < 0** and **balance >= 0** is the inverse of the other, i.e., both together cover all possible cases. In this situation we can do without formulating the second condition explicitly:

```
balance = query_balance()
If balance < 0 Then
  Begin
    show("Attention: Your account balance is negative!")
    overdraftrest = 500 + balance
    show("You still have $", overdraftrest, " left of your
      overdraft facility.")
  End
Else
  Begin
    show("Your account is in credit.")
  End
show("Current balance: $", balance)
```

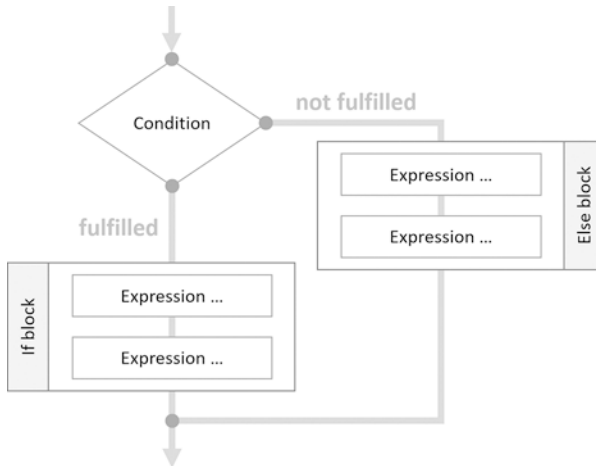
Here, the keyword **Else** tells us that now the code block begins which is executed only if the condition above (**balance < 0**) is not met, that is, if the account balance is greater than or equal to 0. We thus have two blocks of code: The block between **If account < 0 Then** and **End**, and the one between **Else** and **End**. Each time the program is executed, only *one* block is run. So, the program branches at this point. If the condition **If balance < 0** is met, it first continues, shows the warning message, calculates the overdraft balance, and displays this. Then it encounters the **Else**. Since the first condition was already met, the **Else** block is skipped and the execution is continued only with the next statement after the **Else** block, in our case **show("Current balance: \$", balance)**.

A schematic representation of the flow of an If-Else construct is shown in ■ Fig. 14.1.

Most programming languages have such If-Else constructs. The structure of these constructs is usually very similar. To illustrate this, let's take a look at how the above problem would be solved in three different programming languages.

First in C/C++:

```
if(balance < 0)
{
    printf("Attention: Your account balance is negative!");
    float overdraftrest = 500 + balance;
    printf("You still have $%f left of your overdraft
      facility.", overdraftrest);
}
else
{
    printf("Your account is in credit.");
}
printf("Current balance: $%f", balance);
```



■ Fig. 14.1 Flow chart of an If-Else construct

Next in Python:

```

if balance < 0:
    print("Attention: Your account balance is negative!")
    overdraftrest = 500 + balance
    print("You still have ", overdraftrest,
          " left of your overdraft facility.")
else:
    print("Your account is in credit.")
    print("Current balance: $", balance);
  
```

And finally, in Visual Basic for Applications (VBA):

```

If balance < 0 Then
    MsgBox ("Attention: Your account balance is negative!")
    overdraftrest = 500 + balance
    MsgBox ("You still have " & overdraftrest & " left of your
            overdraft facility.")
Else
    MsgBox ("Your account is in credit.")
End If
MsgBox ("Current balance: $" & balance)
  
```

If you compare the three examples, you will first notice the fundamentally similar structure. All three languages have an If-Else construct. The implementation differs only in detail:

- **Keywords:** In all three languages, the keywords **if** and **else** exist. However, an explicit **then** is written only in VBA.
- **Conditions:** C/C++ requires that the condition be written in parentheses, which the other two languages do not. There, however, parentheses do no harm either; we will learn the reason for this below.
- **Code blocks:** Significant differences exist, however, in how the code blocks that are to be executed when the condition is met (if-case), or when the condition is not met (else-case), are delimited: In C/C++, curly braces are used to mark the beginning *and* end of a code block. In Python, however, code blocks start with a colon and are indented. All statements at the same indentation level are considered part of the code block. Therefore, there is no need for a special keyword at the end of the **else** block anymore. Here you must pay close attention to how the code is formatted; the indentations have a meaning and therefore you can't just indent as you like. In the VBA example, there are the blocks between the **Then** and the **Else**, and between the **Else** and the **End If**. Note here that, unlike the **End** used above, the **End If** does not mark the end of the **If** block, the part of the code that is executed when the condition **balance < 0** is met. It marks the end of the entire If-Else construct. The if-block simply ends at the **Else**. Only if no **Else** branch exists, the **End If** marks the end of the **If** block.

? 14.1 [3 min]

What does the following pseudo-code program when

- a. **x** = 10
- b. **x** = 11
- c. **x** = 25
- d. **x** = -1

is entered?

```
x = enter("Please enter a number: ")
If x > 10 Then
  Begin
    If x > 20 Then show("Result A")
    show("Result B")
  End
Else
  Begin
    If x > 0 Then show("Result B")
  End
```

? 14.2 [3 min]

Modify the program section from ► Exercise 14.1 so that **Result B** is only displayed if **x** is greater than 10 and less than or equal to 20.

14.4 A Closer Look at Conditions

In the previous section, you saw how If-Else constructs can be used to branch to different parts of the program depending on whether a particular condition is met or not. In the following, we will take a closer look at the conditions that are checked in the If-Else constructs.

The great common feature of all conditions is that their result is either *true* or *false*, the condition is satisfied or not. Any logical expression that can be evaluated as true or false is thus suitable to be used as a condition. Often such logical expressions are value comparisons, as we saw in the online banking example of the previous section. There, for example, we checked whether the account balance was less than 0. Such value comparisons can be easily formulated using the comparison operators known from mathematics, such as **>** (greater than), **<** (less than), and **=** (equal to). Here, **>=** is usually written for “greater than or equal to” and **<=** for “less than or equal to” because the special characters **≤** and **≥** did not exist in the character sets of the early computer age, and today, even though they are theoretically available, they cannot easily be typed directly from the keyboard. Therefore, to facilitate fast typing of programs, the compound notation has been agreed upon. The not-equal character **≠** poses a particular challenge, because there is no natural “translation” into a compound character. And indeed, different programming languages have found different solutions for this. The two most commonly encountered are **<>** (greater than and less than) and **!=** (exclamation mark and equal sign).

In the previous section, conditions always consisted of comparing a variable with a value. In fact, any expression that can be true or false is conceivable as a condition. Imagine that we have a function in our online banking example that checks whether the current customer has a positive account balance. The return value of this function can be stored in a variable:

```
accountcredit = is_balance_positive()
```

So we assign the return value of the function **is_balance_positive()** to the variable **accountcredit**. Accordingly, the value of the **accountcredit** variable is now either **TRUE** or **FALSE**. We can check the content of the variable in a condition:

```
If accountcredit = TRUE Then
Begin
    show("Your account is in credit!")
End
```

Instead of checking the variable **accountcredit**, most programming languages allow you to simply insert the function directly into the condition:

```
If is_balance_positive() = TRUE Then
Begin
    show("Your account is in credit!")
End
```

This procedure is always useful if you do not want to continue working with the return value of the function. However, if you want to access this value again at another point, it makes sense to store the value in a variable for further use. This saves a repeated call of the function and thus computing power and time: If the program code behind the function is very complex, your program will run faster if you simply fall back on the value stored in the variable instead of having it recalculated by a function call.

Another simplification that most programming languages allow is to simply drop the explicit comparison with the logical value **TRUE**. So, the assumption is that whenever a value (either the value of a variable, the return value of a function, or the result of an expression computed in whatever way) is used in a condition without explicitly comparing it to another value, the comparison should be made to the logical value **TRUE**. In our example, we could write:

```
If is_balance_positive() Then
Begin
    show("Your account is in credit!")
End
```

If you have given the function a meaningful name, as we have done here, the condition is very easy to read and understand.

As you can see, even a normal comparison of values is in the end always a comparison with the value **TRUE** and thus a check of a condition according to the common scheme.

For instead of, say

```
If balance > 1000000 Then
```

you could also write:

```
If (balance > 1000000) = TRUE Then
```

Then the expression on the left would be evaluated first. If the account balance were now greater than one million, the expression would take the value **TRUE** and the condition would be fulfilled. Conditions are therefore ultimately always comparisons with the value **TRUE**.

14.5 Complex Conditions with Logical Operators (AND, OR, NOT)

In the examples of the previous sections, it was always a single elementary condition in an if-then block that decided whether a program part is executed or not. Of course, the condition in an if-then block can also be a condition composed of several sub-conditions.

Let's assume, for example, that in our online banking example we only want to allow a transfer if the account is not blocked *and* the sum of the account balance (which can also be negative) and the overdraft facility is at least as high as the amount to be transferred. If, for example, the account balance were \$-150, the overdraft facility \$500 and the user of our online banking wanted to transfer an amount of \$50, the online banking should allow this, because the sum of the account balance and the overdraft facility, and thus the amount available for the online banking customer, would amount to \$350, i.e. more than is actually to be transferred. If, on the other hand, the customer wanted to transfer \$400, our online banking application should reject this transaction request, because the amount to be transferred would exceed the available sum from the account balance and overdraft facility by \$50.

To make life easy for ourselves, let's assume that we have an `is_account_locked()` function available that returns **TRUE** if the account is locked and **FALSE** if it is not. Then the condition that checks whether the customer is allowed to make a transfer of **amount** dollars or not would look like this in an if-then block:

```
If is_account_locked() = FALSE
    AND balance + overdraftrest >= amount Then
Begin
...
End
```

Here you can see that we link two (partial) conditions with a logical **AND**. The total condition of the if-then block is therefore only fulfilled if *both* the one *and* the other partial conditions are fulfilled.

In addition to the **AND**, there are other logical operators that you can use to combine more complex conditions. For example, the logical **OR**, which links two (partial) conditions in such a way that the total condition is fulfilled exactly when one, the other, or both partial conditions are fulfilled. The meaning of the logical **OR** is therefore different from that of the “or” in everyday language, in which an *exclusive OR* is often meant: *either* one *or* the other, but not both together.

As you can easily see, logical operators work just like the ones you know from mathematics. Accordingly, programming languages also know an *exclusive OR* (often called **XOR**), which corresponds to the colloquial “or”.

Another important logical operator is the logical **NOT**, which reverses the logical truth content of a statement. The inverse of the truth content of the statement “The account is locked” is obviously “The account is not locked”. In programming, unfortunately, we usually cannot put the **NOT** so elegantly between the

“words” of our “sentence”. Therefore, in the program code, it tends to result in something like “NOT the account is locked”. So instead of the above condition, we could also write:

```
If NOT is_account_locked
    AND balance + overdraftrest >= amount
Then
Begin
...
End
```

As you recall from the previous section, **is_account_locked()** is a shorthand notation for **is_account_locked() = TRUE**, so if you want to check for the value **TRUE**, you can simply omit the explicit comparison to that value, because that is what is checked for by default if you do not specify a different comparison value.

The condition designed in this way thus first determines the value of the function **is_account_locked()**. With the help of the NOT operator, this truth value is then simply reversed. So, if the account is unlocked, **is_account_locked()** returns the value **FALSE**, the logical NOT reverses the value to **TRUE**. Thus, the condition shortens to **If TRUE AND balance + overdraft >= amount**. The truth value of the total condition then depends on whether the second sub-condition is fulfilled or not.

Of course, logical expressions like the above can also be nested with brackets in any complexity. The parentheses ensure that the contents of the parentheses are evaluated first, before the value thus determined is logically linked with other expressions.

Here’s a simple example:

```
If NOT is_account_locked()
    AND (balance + overdraftrest >= amount
    OR is_customerhistory_positive()) Then
Begin
...
End
```

14

The second sub-condition evaluates whether the amount to be transferred meets the usual requirement or whether the customer has such a positive history (for example, consisting of regular receipts on the account, no significant overdrafts, etc.), determined with the help of a function **is_customerhistory_positive()**. Thus, if the customer had exhibited exemplary behavior to date, the transaction would be allowed even if it exceeded the overdraft limit. The parentheses ensure that the first step is to determine whether the transaction should be allowed based on the sufficiently small transfer amount or based on the customer’s past behavior. For this purpose, two sub-conditions are linked with logical OR. This results in a logical value, which is then linked in the second step with the logical value from the check whether the customer account is blocked with logical AND.

As you might have guessed, logical operators have different names in different programming languages. Often, it remains with the English keywords **AND**, **OR**,

XOR, **NOT**, from which the meaning of the operator is immediately apparent. However, some languages like C, C++, Java or R use special characters instead of the written form: The ampersand (&) for the logical AND, the pipe (|) for the logical OR, and the exclamation mark (!) for the logical NOT. By the way, now you can easily see why inequality is represented by **!=** in these languages. It simply means “NOT equal”.

In C, C++, Java, or R, the if-then block above would read like this:

```
if(!is_account_locked()
    & (balance + overdraft >= amount
    | is_customerhistory_positive())) {
    ...
}
```

In the Visual Basic for Applications (VBA) macro language, which allows you to automate the Microsoft Office suite of applications, you would write this if-then block using descriptive identifiers for the logical operators:

```
If Not is_account_locked()
    And (balance + overdraftrest >= amount
    Or is_customerhistory_positive()) then
    ...
End If
```

? [10 min]

Consider the following program excerpt:

```
x = input("Please enter a number: ")
If x > 100 Then show ("x greater than 100!")
If x > 50 Then show ("x greater than 50!")
If x > 10 Then show ("x greater than 10!")
If x < 0 Then show ("x less than 0!")
If x >= 0 And x <= 10 Then show("x between 0 and 10!")
```

- How many conditions are checked when the user enters a value for **x**?
- Modify the algorithm so that only one condition is checked in the best case and all conditions are checked only in the worst case.

? [3 min]

Where is the error in the following program excerpt?

```
x = input("Please enter a number: ")
If x > 100 Then show("x greater than 100!")
Else
    Begin
        If x >= 110 Then show("x greater than 110!")
```

```
Else show("x less than or equal to 100!")
End
```

14.6 Checking Similar Conditions Efficiently with Switch-Case Constructs (Switch/Select...Case)

Sometimes you want to check many similar conditions at once. Imagine we wanted to classify the customers of our online banking according to how much money is received in their account each month, in order to be able to offer special services to particularly “good” customers. We want to assume that we have a function **received_3months()** available to query the average amount of money received over the last three months. In our pseudo-code language, the classification of the customers could be done like this:

```
If received_3months() < 1000 Then
Begin
    category="D"
End
Else
Begin
    If received_3months() < 2000 Then
    Begin
        category="C"
    End
    Else
    Begin
        If received_3months() < 4000 Then
        Begin
            category="B"
        End
        Else
        Begin
            category="A"
        End
    End
    End
End
```

14

Here, the different thresholds for the average monthly cash inflow are checked in a nested If-Else construct. However, this is relatively difficult to read. Many programming languages know a construct that allows you to write the check of several similar conditions more elegantly. In our example, the formulation could be written as follows:

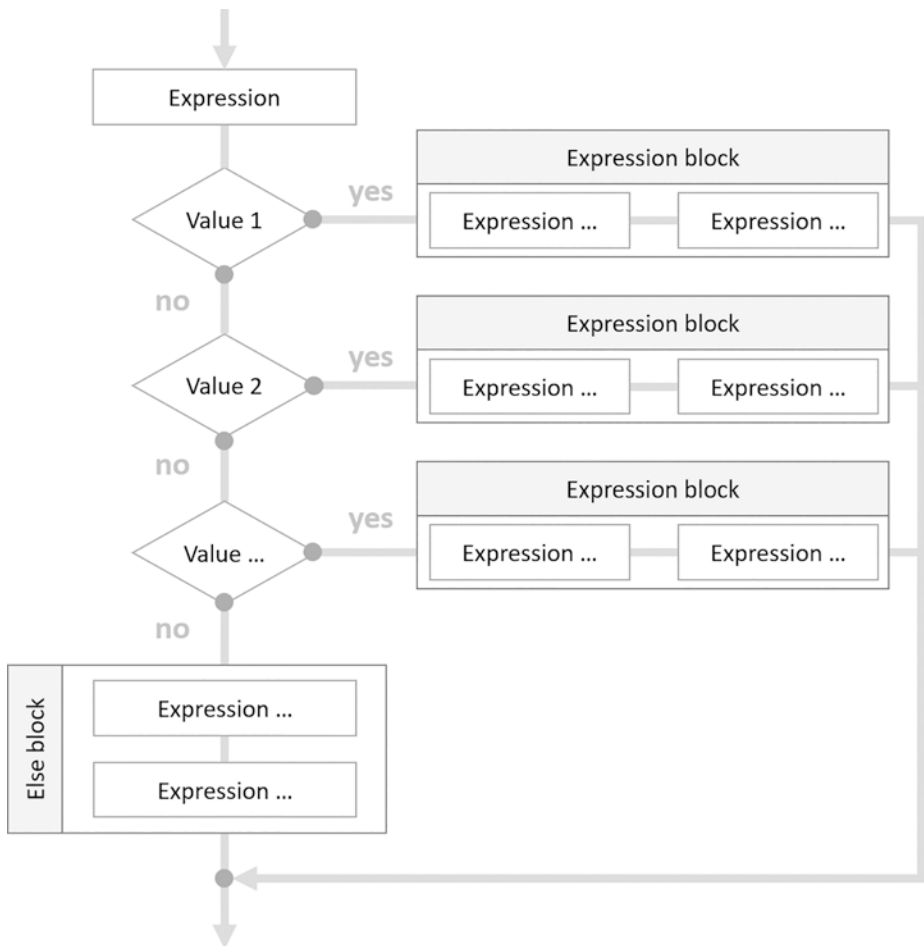
```
Switch received_3months()
Begin
    Case < 1000: category="D"
    Case < 2000: category="C"
```

```

    Case < 4000: category="B"
  Else: category="A"
End

```

This notation is much clearer and therefore both more readable and easier to program. Behind the keyword **Switch** is first the variable that is subject to the check. In our case it is simply the return value of our function **received_3months()**. **Case** is used to introduce a condition to be checked, for example **<1000**. The colon is then followed by what is to be done in this case, in our example the setting of the customer category. The special keyword **Else** can be used to catch all other cases that have not been explicitly checked with specific conditions. This statement is only triggered if none of the other conditions apply. If, however, one of the other conditions does apply, the statements assigned to this case (in our example only one, but there could be several) are executed. After that, the entire construct is



■ Fig. 14.2 Flow chart of a Switch-Case construct

exited, that is, none of the other conditions are checked. Instead, the execution of the program continues after **End**.

The schematic representation of the sequence of a Switch-Case construct is shown in ■ Fig. 14.2.

Many programming languages allow for this elegant approach of checking multiple similar conditions. Typically, such constructs are called *Switch-Case* or *Select-Case constructs*, after the two central keywords **switch** or **select** and **case**, which are commonly used in most languages that have such a construct. Confusingly, some languages use the keyword **case** instead of **switch** or **select**, but we won't concern ourselves with that here.

For example, in C, a language that supports Switch-Case constructs, the above check of the client category would then read like this:

```
switch(received_3months())
{
case < 1000:
    category="D";
    break;
case < 2000:
    category="C";
    break;
case < 4000:
    category="B";
    break;
default:
    category="A";
    break;
}
```

14.7 Events

In addition to classical if-conditions (including their special version as Switch-Case constructs), there is yet another important way to branch to different parts of the program while it is running, and that is through what are called *events*.

Classical conditions are executed when the program has reached the corresponding position where the branch condition is located. The program therefore runs completely *sequentially*. Sometimes, however, you do not know in advance exactly when in the program flow you want to branch. In this case, events can help.

Imagine that in our online banking example there are three buttons on the banking website, “New transfer”, “Export transaction as text file” and “Logout”, which the user can click at any time. Behind each of these buttons are different program instructions, but we do not know in advance when – and if at all – the user will trigger the corresponding functions.

There are basically two ways to deal with this problem:

- Either the program runs in a virtually infinite loop and actively observes the user's behavior at all times, that is, it checks whether the user has clicked the button or not. We will deal with loops a little later in this part of the book, but

the basic idea of a practically infinitely repeating loop that always jumps back to its beginning and runs again should be understandable. Now, as soon as the user clicks on one of the buttons, the program executes the program code behind the button and then returns to the infinite loop in which it constantly monitors the user's actions.

- The second possibility is that the program does not wait in an infinite loop for the user but does whatever it is supposed to do. However, when it learns from the outside that the user has clicked one of the buttons, it immediately jumps into a function that then executes the instructions behind the button.

The difference between the two approaches is that in the first case, waiting for the user and monitoring their activities practically blocks the program. It is always in an *active monitoring state* and constantly checks whether one of the buttons has just been clicked. In the second case, the program learns from *the outside* that a certain *event* has occurred. As soon as this happens, a function associated with this event is executed. Such functions are also called *event handlers* because they describe how the event is handled.

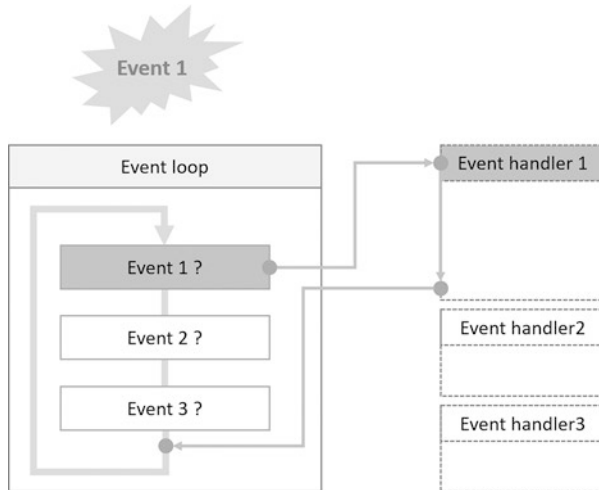
This approach saves computing power, because the active observing that the first approach requires is like active listening. It means signals must be constantly processed, whereas with event control, something else, such as the operating system or interpreter, simply lets you know when an event has occurred. This allows the program to do other things in the meantime, and still react to an event when it occurs.

The way it works is similar to that of an oven which you have to preheat to a certain temperature in order to bake a pizza, for example. Only when the preheating temperature is reached can the pizza be put into the oven. You could go to the oven every few minutes to see if the preheat temperature has already been reached. Or you could have an oven that independently gives an acoustical signal when it is ready. In the first case, we are in the infinite loop and check again and again whether the oven has already reached the preheating temperature. In the second case, someone else (in this case, the oven itself) informs us that the “preheat temperature has been reached” event has occurred. As long as this has not happened, we can simply do something else, because we know that the oven will report when the time comes.

Many programming languages support the use of events. Because it changes the way the program is structured, we speak of the *event-oriented programming paradigm* in this context. Languages that support the processing of events are languages such as JavaScript, which are used to develop graphical user interfaces where you do not have a completely sequential program flow, and do not know in advance exactly which program part must be executed next, due to the unpredictable behavior of the user. In general, graphical user interfaces, as we learned about in ► Sect. 12.2.1, are the prototypical application of an event-oriented approach.

The schematic representation of an event-controlled program is shown in ■ Fig. 14.3.

By the way, not only the user can trigger events. The operating system or connected devices can also trigger events to which the programs can then react. For example, the operating system can announce that it now wants to shut down; that



■ Fig. 14.3 Flow chart of an event-controlled program

gives programs that have an event handler the opportunity to react, and save the current state of the program before the program is closed. Similarly, devices can trigger events, such as a printer or hard disk drive.

So far we have dealt with the basic functionality of event handlers, but what do such functions look like now? Here is an excerpt from the example outlined above (as always, in our pseudo-code language):

```

Function event_transfer(e)
Begin
    create_transfer()
End

Function event_logout(e)
Begin
    logout()
End

```

Here we define two event handlers, one for the event where the user clicked on the “New transfer” button, and one for the event where the user wants to log out. As you can see here, the event handlers are just normal functions. In some languages, event handlers are passed a special object as a function argument (above, the argument **e** of type **event**) that describes the event in more detail. For example, for an event that fires whenever the user moves the mouse, the current “coordinates” of

the mouse pointer could be passed as properties of the object. The event handler function can then retrieve these coordinates from the object and react accordingly.

The only difference between these functions and the ones we learned about in ► Chap. 13 is that they are not called by us, the programmers, but “from outside”. The interpreter or the operating system notifies us when our event has occurred, and they do this by calling our event handler, giving it the event object **e** as additional information. So, in the end, event handlers are simply functions that we develop but don’t call ourselves; we just provide them for whoever signals us that the event has occurred.

The above examples would look like this in JavaScript:

```
function event_transfer() {  
    create_transfer();  
}  
  
function event_logout() {  
    logout();  
}
```

In order for the interpreter to know which event handler to call for each event, the button definitions in the HTML source code of the website would need to be as follows:

```
<button onclick="event_transfer()">New transfer</button>  
<button onclick="event_logout()">Log out</button>
```

This way, two buttons are created and their (standard) **onclick** events, which are triggered whenever someone clicks on the button, is linked to our respective event handlers.

In Delphi, our event handlers would look like this:

```
procedure BankingForm.LogoutButtonClick(Sender: TObject);  
begin  
    create_transfer();  
end;  
  
procedure BankingForm.NewTransferButtonClick(Sender: TObject);  
begin  
    logout();  
end;
```

Here we would have defined a window (a “form”) called **BankingForm**, on which we would place the buttons with the names **NewTransferButton** and **LogoutButton**. The event handlers not only carry the name of the respective button, but also, with “Click”, the name of the event they cover. This way Delphi understands which event it is and which element of the user interface it refers to. The event handlers are also given the respective button that was clicked with the argument **Sender**, which becomes interesting if you use the same event handler for several user interface elements, and you have to distinguish between the different elements that may have triggered the event within the event handler function.

? 14.5 [3 min]

How does a program that follows the event-oriented programming paradigm differ from completely linear programs?

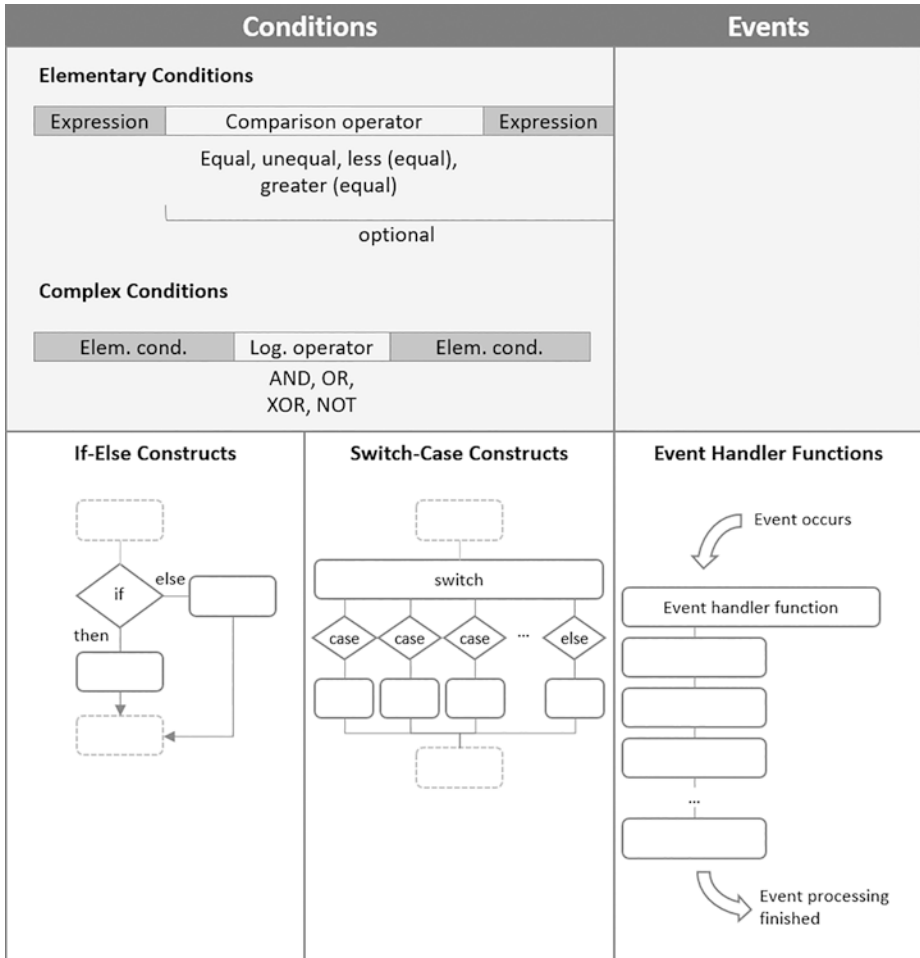
? 14.6 [3 min]

Why is event control particularly well suited for graphical user interfaces?

14.8 Your Roadmap to Learning a New Programming Language

If you are learning a new programming language....
you’ll discover:

- how If-Else constructs are formulated in the language (key words used for this in many languages are **if**, **then**, **else**),
- how to write the comparison operators used to formulate conditions,
- how to write the logical operators that are used to combine several elementary conditions into one overall condition,
- whether the language has a Switch-Case construct and, if so, how it is formulated (keywords used for this in many languages are **switch**, **select**, **case**),
- how code blocks are delimited (i.e., opened and closed) in the language, and whether these delimiters can be omitted if the code block contains only a single statement,
- whether the language supports events, and if so, how to define event handlers (including which arguments to pass to event handlers, if any), and how to link event handlers to the events on whose occurrence they are to be called.



■ Fig. 14.4 Conditions and events for sequence control

An overview of the considered constructs for sequence control can be seen in ■ Fig. 14.4.

14.9 Solutions to the Exercises

■ Exercise 14.1

- Output: **Result C**
- Output: **Result B**
- Output: **Result A, Result B**
- No output (none of the conditions covers the case where x is negative)

■ Exercise 14.2

```
x = input("Please enter a number: ")
If x > 10 Then
    If x > 20 Then show("Result A")
    Else show("Result B")
Else
    If x > 0 Then show("Result B")
End
```

Within the code block **If x > 10**, we first check for **x > 20**. If this condition does not apply, we know that **x** must be greater than 10 (otherwise the program would not have entered this code block at all), but less than or equal to 20. Therefore, it is sufficient to place the keyword **Else** in front of the output of **Result B**.

■ Exercise 14.3

- Five times. Each condition is checked, regardless of the previous result of the conditions.
- By clever nesting of If-Else constructs, the number of checks of conditions can be reduced. If **x** is less than 0, only one condition is checked. The remaining conditions are then not checked at all since they are in the **else** branch to the condition **x < 0**. Only if **x** is greater than or equal to 0 and at the same time less than or equal to 10, the program passes through all conditions.

The trick here is to arrange the conditions in such a way that each condition can at least theoretically be achieved. For example, if we had used **x > 10** as the top/outmost condition, conditions such as **x > 50** would no longer be checked at all, because **x > 10** has already been evaluated as true, and everything that lies in the **else** branch to **x > 10** condition is no longer passed through.

Of course, a formulation with a Switch-Case construct (► Sect. 14.6) would be a much more elegant and readable solution.

14

```
x = input("Please enter a number: ")
If x < 0 Then show("x less than 0!")
Else
    If x > 100 Then show("x greater than 100!")
    Else
        If x > 50 Then show("x greater than 50!")
        Else
            If x > 10 Then show("x greater than 10!")
            Else
                If x >= 0 AND x <= 10 Then
                    show("x between 0 and 10!")
                End
            End
        End
    End
End
End
End
```

■ Exercise 14.4

The problem with this nested If-Else construct is that the condition that checks x in the inner If-Else construct to see if it has a value greater than or equal to 110 can never come to a positive result. If x is indeed greater than 110, the outer condition $x > 100$ is already true, and its associated code block (here consisting of only a single statement) is then executed. The **else**-block, on the other hand, is only executed if x is less than or equal to 100, but then the condition of the inner If-Then-construct ($x \geq 110$) can never be true. The output statement for x greater than 110 is isolated, so to speak.

■ Exercise 14.5

Unlike a linear program, a program that follows the event-oriented programming paradigm can react to events (for example, the user's click on a button) by jumping to a place in the code that was developed precisely for this type of event. If there is no event to be processed at the moment, the program observes its environment attentively and waits to become active again. While a linear program is a long sequence of instructions that are executed step by step from beginning to end, an event-oriented program consists of a collection of event handlers, i.e., functions that are activated whenever the event that the respective function processes has occurred. The event handlers themselves are sequences of instructions, but instead of executing a linear flow control, the program jumps back and forth between the event handlers, depending on which event it is currently processing.

■ Exercise 14.6

Graphical user interfaces usually give the user a lot of freedom to decide which functions to invoke. The user is often not tightly guided but chooses from a palette of options that are regularly reflected in menus, toolbars, buttons, tabs, and other controls. This is where event-based control comes in handy, simply calling the event handler associated with that control that the user has triggered/activated.



Loops: How Do I Repeat Program Instructions Efficiently?

Contents

- 15.1 Loops and Their Variants – 178
- 15.2 Counting Loops – 179
- 15.3 Conditional Loops – 185
- 15.4 Your Roadmap to Learning a New Programming Language – 189
- 15.5 Solutions to the Exercises – 189

Overview

Practically all programming languages use loops to execute similar parts of the code several times in succession. This not only saves a lot of manual and error-prone work, but also allows you to write code that you don't know in advance how often it will be executed. With the help of loops you can elegantly implement the repetition of program statements.

In this chapter you will learn the following:

- What loops are, and how they differ from functions, which can also be used to execute program code repeatedly
- What the difference is between counting and conditional loop constructs
- How to develop a counting loop, and what role the “run variable” plays in it
- How to program a conditional loop and what the difference is between the condition check at the beginning (head control) and at the end of the loop (foot control)
- How to end loops prematurely or continue with the next loop pass.

15.1 Loops and Their Variants

■ Loops versus Functions

Essentially, there are two ways to repeat program code that you have written once: You can swap it out into functions/procedures that can be called again. We have already looked at how this works in ► Chap. 12. Or you can wrap the code to be repeated in so-called *loops*. This is exactly what this chapter is about.

The difference between the two approaches is that with loops, the same program code is run through and executed *directly several times in succession*. For example, you could go through a product price list line by line, product by product, and increase the price of all products that belong to a certain category by 10%. The use case for functions, though, is different; here, it is not a matter of repeating a certain part of the program *several times in immediate succession*, but of making the program code independent to such an extent that it can be called from *different places in the program* at any time.

15

What both approaches to the repetition have in common, is that in this way you avoid writing the same program code several times in the program. This increases the maintainability of the program because a change to the part to be repeated only must be made once. The alternative would be to duplicate the program code wherever you want to use it (instead of calling a function) or duplicate it several times in succession (instead of using a loop), and as a result you would have to make changes to this code in every place it occurs in the program as well. This is tedious and error prone. Another problem in the case of loops is that at the time you write the program, you usually don't even know how many times exactly you want the code to repeat. Think of the product list you want to go through to

increase the price of all products in a particular category by 10%. While you are writing the program, you may not know how many products the list will contain! What happens if a 79th product is suddenly added to the existing 78 because the assortment is expanded? The loops provide a remedy here.

■ Types of Loops

Loops basically come in two variants: either as *counting loops* or as *conditional loops*. With counting loops, it is clear from the outset – at least in principle – how often they are run through. Such loops lend themselves to our problem with the product list. We should be able to easily determine how many products are on this list. But then we also know the number of loop passes that are necessary to run through the product list once completely.

For conditional loops, imagine the following scenario: We have a program that converts temperature values entered by the user in Kelvin to degrees Celsius. Temperatures in Kelvin are always greater than or equal to 0. Zero Kelvin is absolute zero, with no heat at all, it can never get colder; this absolute zero corresponds to -273.15 °C on the Celsius scale. The user is to enter temperatures in Kelvin one after the other. Each Kelvin temperature is converted to degrees Celsius. If the user enters a negative Kelvin temperature at any time, the program should abort. A loop is a good idea here as the conversion process from Kelvin to Celsius is always the same. In this case, however, we do not know in advance how many loop passes there will be. That depends on the user input. The loop of choice here is a *conditional* loop, a loop that runs as long as a certain condition is met; in our example here, the condition would be that the temperature entered by the user is greater than or equal to 0. As soon as the user enters a negative temperature, the condition would no longer be satisfied, and the loop would stop.

15.2 Counting Loops

■ Loops with a Numeric Run Variable

Counting loops are successive repetitions of program code, used where the number of repetitions can be determined in advance. But are the repetitions of the program code really absolutely identical? Let's think back to the example of the product price list that we want to go through to make all products of a certain category 10% more expensive. Now, if we found a product in the category and one of the previous products already belonged to the same category, is the code that is executed in both cases exactly the same? No, apparently not; for the price increase must be made once for one product and once for the other. So, the program code, similar as it is in both cases, must “refer” to a different product each time. Put another way: We need to know where exactly in our loop we currently are, i.e., which product is currently being processed.

This is exactly what the *run variable* allows us to do. It is incremented by a certain value (usually one) with each loop pass. So, you always know which loop pass you are currently in. And we can use this variable in the program code that is

repeated through our loop, for example to address the individual elements of a field that represents our product price list.

Let's take a closer look at the whole thing as pseudo-code:

```
For p From 1 To length(products)
Begin
    If products[p].category = "Garden Furniture" Then
        products[p].price = products[p].price * 1.1
End
```

In this example, we assume that **products** is a field of instances (i.e., objects) of the **Product** class that has the attributes **category** and **price** (scroll back to ► Sect. 11.7 if you have forgotten or are vague about the concepts of “fields”, “classes” and “instances/objects” from object-oriented programming). The run variable in this example is the (integer) variable **p**. It runs, as the line **For p From 1 To length(products)** indicates, from the value 1 to the function value **length(products)**, which we want to assume represents the length of the field **products**, i.e. the number of products contained. With each loop pass, **p** is automatically increased by one. The program code to be repeated is located between the **Begin** and **End** keywords in a code block. We have already seen code blocks in connection with functions (► Sect. 13.1) and If-Then constructs (► Sect. 14.3). The code, which is between **Begin** and **End**, is executed as long as the run variable **p** is less than or equal to the specified maximum value, in our case **length(products)**.

We now use the run variable inside the loop at **products[p]** as an index to pick the respective field element that is to be processed in the current loop pass from the field **products**. Here you can see that although we always repeat the same code, the code does something different each time, because each loop pass works with a different product object, namely the respective element **products[p]**, so the first loop pass works with **products[1]**, the second with **products[2]** and the last with **products[length(products)]**.

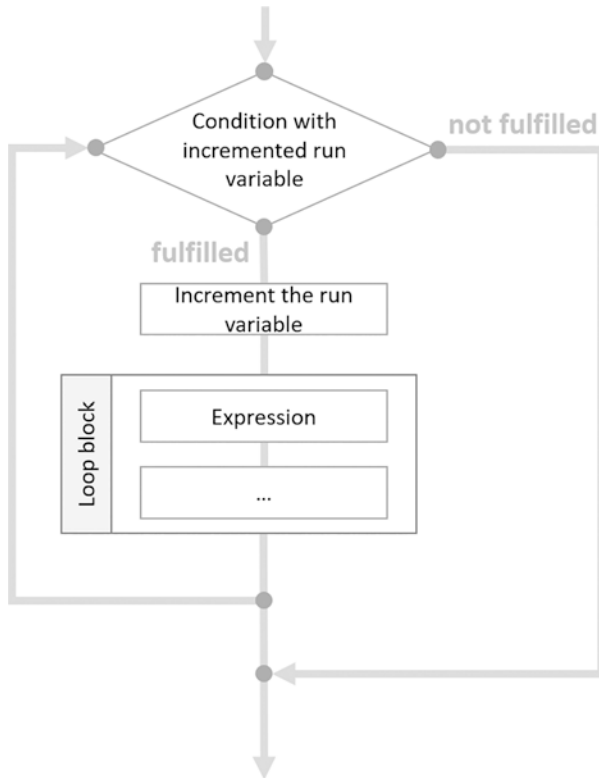
The passage through a counting loop is shown schematically in ■ Fig. 15.1.

■ Loops with Object Run Variable

Some programming languages offer a special variant of the counting loop, which makes tasks such as our example above even easier. Here, instead of counting up a run variable, which we then use as an index for an array of objects, the loop directly traverses the array of objects and works with each individual object in the array in turn.

It might look like this:

```
For Each p In products
Begin
    If p.category= "Garden Furniture" Then
        p.price = p.price * 1.1
End
```



■ Fig. 15.1 Flow chart of a counting loop

This loop, which is introduced in our pseudo-code with the keywords **For Each**, simply runs through all elements of the field **products** in order. The current element of the field in each case is stored in the run variable **p**. Note the difference to the counting loop in the example above: Here, the run variable **p** is not a number that indicates how many times the loop has been running so far, but the current element of the field **products** that is currently in the focus of the loop.

Therefore, in the body of the loop, i.e., in the code block repeated by the loop, this current object **p** can be worked with in the same way as a product object, for example, its attributes can be adjusted. It is important to understand at this point that in most programming languages our variable **p** is not simply *a copy of* the respective element of the field **products**, but ultimately *the element itself*. Sounds abstract, but it makes a crucial difference: After all, if **p** were simply a copy of the particular element of **products** that is currently up, then any changes we make to **p** would of course have no effect on the real element in **products**; after all, we are only working with a copy, and the original would remain unaffected by our changes. But this is not the case. In fact, **p** is the respective element of **products**. Changes made to **p** therefore directly change the respective product in our field **products**.

As you can see, this loop is a bit more elegant than the loop above. The only disadvantage here is: Without counting with the help of an extra variable, which we manually increase with each loop run, we now do not know how many runs of the loop have taken place so far. Our run variable is simply no longer a counter. However, if it is simply a matter of passing through the products in our **products** field in sequence, this need not be a disadvantage.

■ Examples in Different Programming Languages

Let's look at it in two concrete programming languages, PHP, which is specifically made for server-side programs and is used on most websites, and the Microsoft Office macro language VBA.

First, here are the two loop variants in PHP:

```
for ($p=0; $p <= count($products)-1; $p++) {
    if ($products[$p]->category == "Garden Furniture")
        $products[$p]->price = $products[$p]->price * 1.1
}

foreach ($products as &$p) {
    if ($p->category == "Garden Furniture")
        $p->price = $p->price * 1.1
}
```

The counting loop with numeric run variable is introduced here by the keyword **for**, the loop iterating through the field with **foreach**. These are the keywords used in most programming languages for these loop types.

The **for** loop contains three specifications in parentheses:

1. What the run variable is called (variable names are always prefixed with a dollar sign in PHP) and at what value it should start (fields start at index 0 by default in PHP, so the first element would be as **products[0]**).
2. How long it should run; in our case as long as its value is less than or equal to the number of field elements (**count(\$products)**) minus 1; “minus 1” because indexing starts at 0. If the first field element has the index 0, then the last field element has the index **count(\$products)-1** (i.e., with 10 field elements, the index of the last one is 9).
3. How it should be incremented; so far, we have assumed that the run variable is incremented by 1 with each loop pass. However, this does not necessarily have to be the case. For example, we could also just look at every second product; then the last part of the **for** statement would be **\$p = \$p + 2** (**\$p++** is a shorthand for **\$p = \$p + 1**).

In PHP, the attributes of an object are accessed with the help of the arrow operator (->). We had always used the dot for this in our pseudo-code so far.

Another specific feature is found in the second loop variant, the **foreach** loop: Here, the element **\$p** currently processed by the loop is named. However, this is preceded by an ampersand. This ensures that the variable **\$p**, as we discussed above, actually represents the corresponding product object and not just a copy of the currently processed product object in the **products** field. If we were to forgo the ampersand, the assignment **\$p->price = \$p->price * 1.1** would merely process a copy of the object and not the object itself, which is part of our **products** field.

Now the same in VBA:

```
For p = 1 To length(products) Step 1
    If products(p).category = "Garden Furniture" Then
        products(p).price = products(p).price * 1.1
Next

Rem ATTENTION: This loop does not lead to the same result!
For Each p In products
    If p.category = " Garden Furniture" Then
        p.price = p.price * 1.1
Next
```

As you can see, the syntax of the **For** loop (and also the field and object accesses as well as the if-then conditions) is structured a little differently in VBA than in PHP, but the basic concepts are completely identical. However, there is one important difference in the **For Each** loop: Unlike in PHP, the run variable **p** in VBA is always a *copy* of the respective element from our **products** field. There is no way to create the run variable so that changes to it are reflected in the original element of our **products** field. If we want to change the **products** field itself, we have to use the first variant of the counting loop with a numeric run variable.

■ Nested Loops

Loops can also be nested within each other. Imagine you have a two-dimensional array whose rows represent the different variants of one and the same product and whose columns represent the different warehouses of a company. The values in the array represent the number of pieces in each case. For example, if **stock[7,3] = 65**, this means that 65 items of the 7th product variant are currently available in the 3rd warehouse. If you now want to count how many copies of the product (regardless of which variant) are available in total, i.e., across all warehouses, a nested, counting loop is suitable for this:

```

total = 0

For p From 1 To number_variants
Begin
    For w From 1 To number_of_warehouses
    Begin
        total = total + stock[p,w]
    End
End
End

```

The outer loop goes through the rows of the two-dimensional field, that is, the product variants, while the inner loop goes through the warehouses. This means that the loop works its way forward by starting a new row, then going through all the columns for this row (inner loop) and then switching to the next row (outer loop). In this way, the entire field is traversed once. Its individual elements, **stock[p,w]**, i.e., the number of product variants in the respective warehouse, are added up with the help of the variable **total** by adding the field **total[p,w]** just processed by the loops to the current state of this variable; the assignment **total = total + total[p,w]** does nothing else: It adds the respective field contents to the current value of **total** and again assigns the result to the variable **total**. After the nested loops have run through, the variable **total** contains the total number of all product variants across all warehouses.

■ Exit Loops Prematurely

Most programming languages provide a way to exit a counting loop early. This can be useful if, for example, you are writing a piece of code to check if there is a product of a special category, say “garden furniture”, among the products. One way to check this is to go through the entire field and check if the current field element is a product of that category. Once such a product is found, the question of whether there are *any* products in the “Garden Furniture” category is answered. The rest of the field doesn’t really need to be run through, it only costs unnecessary computing time and of course doesn’t change anything in the result. So, it would be appropriate to stop the loop at this point. In our pseudo code it could look like this:

```

found = FALSE

For Each p In products
Begin
    If p.category = "Garden Furniture" Then
    Begin
        found = True
        Break
    End
End
End

```

Using the variable **found**, we can determine after the loop whether a product of the category “garden furniture” has been found. If such a product is found during a loop pass, the variable **found**, which we initially initialized with the value **FALSE**, is set to **TRUE** and the loop is exited immediately.

Most programming languages have a statement that causes the current loop pass to end and the loop to simply continue with the next pass.

15.3 Conditional Loops

■ Functionality and Types of Conditional Loops

Conditional loops differ from counting loops in that it is not known in advance how many times the loop will be run. Instead, the execution depends on whether a certain condition, the *run condition*, is met. As long as this is the case, the loop runs. If the run condition is no longer fulfilled at some point, the loop terminates, and program execution continues behind the loop.

Let’s take the example from the introduction to this chapter: Here, the user is prompted for a temperature in Kelvin and then this temperature is converted to degrees Celsius until he enters a negative Kelvin temperature. Then the program knows to abort, because negative temperatures in Kelvin are physically impossible according to the definition of the Kelvin scale. So, the running condition for our conditional loop is that the user enters a positive Kelvin temperature. When he does, the program converts the temperature to degrees Celsius and asks him for the next Kelvin temperature.

In our pseudo-code, this loop could then look like this:

```
kelvin = input("Please enter temperature in Kelvin: ")

While kelvin >= 0
Begin
    show(kelvin, " Kelvin are: ", kelvin - 273.15,
        " degrees Celsius")
    kelvin = input("Please enter temperature in Kelvin: ")
End
```

So, we start by reading in a temperature in Kelvin before the actual loop. This is necessary so that the run condition of the loop can be checked in a meaningful way. The run condition is defined behind the keyword **While**. This is followed by the code block that is executed when the run condition, **kelvin >= 0**, is met. Specifically, the entered temperature is converted to Kelvin and the result is displayed. Then the user enters a new Kelvin temperature. Since the loop then encounters the keyword **End**, it jumps back to the beginning, which causes it to check whether the run condition is still fulfilled when the user enters a new Kelvin temperature.

In our example, the condition is checked at the beginning of the loop. However, the check can also be made at the end. Then we would have a so-called *foot-controlled* loop, in contrast to the above *head-controlled* loop.

It would look like this:

```
kelvin = input("Please enter temperature in Kelvin: ")

Do
Begin
    If kelvin >= 0 Then
        show(kelvin, " Kelvin are: ", kelvin - 273.15, " degrees
            Celsius")
        kelvin = input("Please enter temperature in Kelvin: ")
    End
End
While kelvin >=0
```

Since the condition is only checked at the end here, the loop is run through once in any case until it reaches the condition check. This construction is somewhat complex in our example, because we still have to check within the loop whether the user's input is a valid Kelvin temperature (i.e. greater than or equal to 0), or indicates that the user wants to abort the loop. The conversion to degrees Celsius only happens in our foot-controlled loop if the Kelvin temperature is greater than or equal to 0. Otherwise, nothing at all happens inside the loop after the input. The loop then checks the run condition, determines that it is no longer satisfied, and does not go into another loop pass, but continues the program after the loop.

■ Figures 15.2 and 15.3 each show a schematic representation of the sequence of a head-controlled loop and a foot-controlled loop.

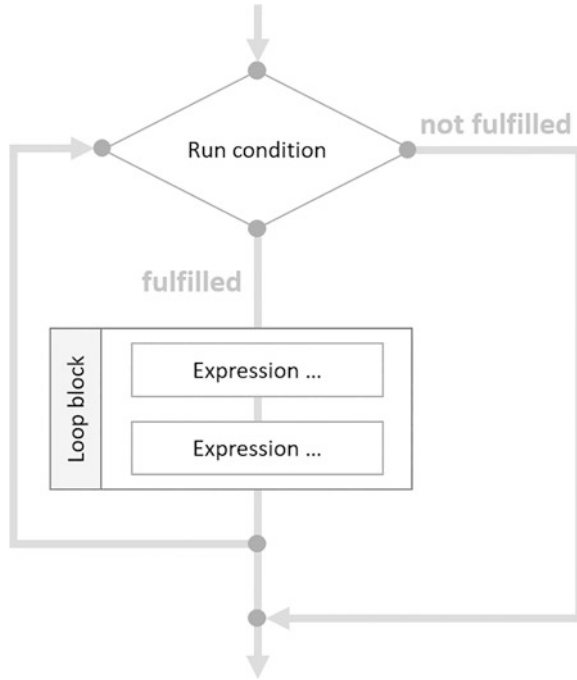
The loop in our example looks altogether a little bit naïve and contrived. Normally you would work with a head-controlled loop in a case like this.

■ Examples in Different Programming Languages

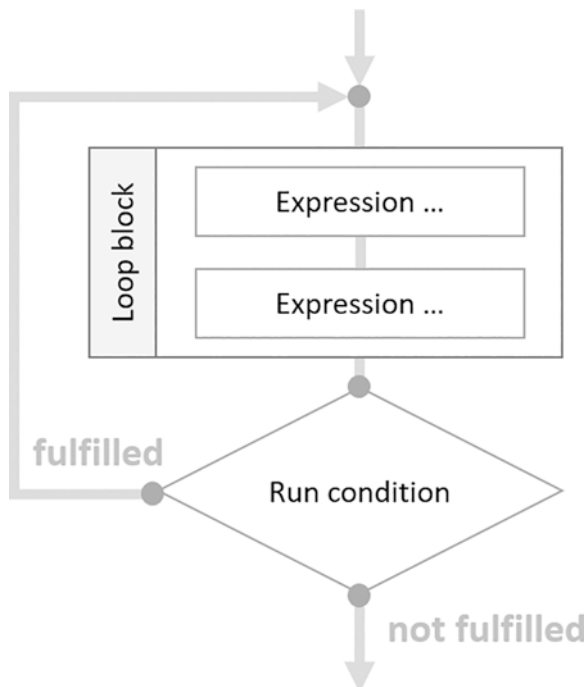
Here are two implementations of our Kelvin-Celsius conversion program in real programming languages, one in Pascal, one in VBA.

First the Pascal version:

```
program temperatureconversion;
var
    kelvin: real;
begin
    kelvin = readln('Please enter temperature in Kelvin: ');
    while kelvin >= 0 do
        begin
            writeln(kelvin, ' Kelvin are: ', kelvin - 273.15,
                ' degrees Celsius');
            kelvin = readln('Please enter temperature in Kelvin: ');
        end
    end.
end.
```



■ Fig. 15.2 Flow chart of a head-controlled conditional loop



■ Fig. 15.3 Flow chart of a foot-controlled conditional loop

Now the whole thing in VBA:

```
Dim kelvin As Double;

kelvin = InputBox("Please enter temperature in Kelvin: ")

While kelvin >=0
    MsgBox (Str(kelvin) & " Kelvin are: " &
        Str(kelvin - 273.15) & " degrees Celsius")
    kelvin = InputBox("Please enter temperature in Kelvin: ")
Wend
```

As you can see, the keywords for conditional loops in Pascal and VBA are **while...do** and **While**, as in most other programming languages. The block of code to be repeated is enclosed in Pascal with **begin** and **end**, in VBA it starts directly after the run condition and ends with the keyword **Wend**.

Just as in the case of counting loops, most programming languages also have special statements to exit a conditional loop completely (often **break**), or to abort the current run and continue with the next run (often **continue**).

■ The Relationship of Counting and Conditional Loops

Counting loops can also be written as conditional loops, but not vice versa. All we need to do for this is to formulate a condition that is satisfied for as long as the counting loop would run. In the example of the previous section, where we want to increase the price of all products in the category “garden furniture” by 10%, it would look like this in pseudo-code:

```
p = 1

While p <= length(products)
Begin
    If products[p].category = "Garden Furniture" Then
        products[p].price = products[p].price * 1.1
    p = p + 1
End
```

15

Again, similar to the counting loop, we work with a run variable. Only this time, we ourselves have to take care of incrementing this variable with each loop pass. We do this with the statement **p = p + 1**. The running condition of the loop is now that the value of this run variable is at most as large as the length of the field **products**. We also have to initialize the run variable ourselves before the first loop pass, which the counting loop also did for us. You can see, however, that it is quite possible to “convert” counting loops into conditional loops, because in the end the counting repetition is also based on a condition. The other way round, of course, is not possible, because in the case of a conditional loop we do not know how often it will be repeated, which is precisely the prerequisite for a counting loop.

15.5 Solutions to the Exercises

? 15.1 [3 min]

What is the problem with the following loop? $x = 1$

```
While x <> 100
Begin
    show("Loop pass no. ", (x+1)/2)
    x = x + 2
End
```

? 15.2 [3 min]

What types of loops are there and how do they differ?

? 15.3 [3 min]

Why can you use a conditional loop to replicate a counting loop, but not the other way around?

? ! 15.4 [5 min]

Consider how a loop could be used to process user input on a graphical user interface.

15.4 Your Roadmap to Learning a New Programming Language

When you learn a new programming language ...
you'll discover:

- which forms of counting loops there are and what role the run variable has in them ("count variable" or current element of a set of objects that is being run through)
- what forms of conditional loops there are, in particular whether there are only head-controlled loops or also foot-controlled loops,
- what possibilities there are to exit loops prematurely or at least to abort the current loop pass and continue the loop with the next pass.

15.5 Solutions to the Exercises

■ Exercise 15.1

The problem with this loop is that it will never end, so it is an infinite loop. The variable x , whose value is checked before each loop pass, is first initialized with the value 1 before the loop and then increased by 2 with each loop pass. Its value is therefore always odd. Therefore, the variable will never take the value 100, which would cause the loops to terminate. The loop continues to run indefinitely (or until the memory is no longer sufficient to hold the high values of the variable x).

■ Exercise 15.2

There are counting and conditional loops. In the case of counting loops, the number of times the loop will be run is, in principle, known before the first loop run. These loops work with a run variable. This run variable is a numerical value which, starting from a start value, is changed according to a fixed rule (for example, increased by one) until it reaches a fixed end value. Alternatively, the loop passes through a set of objects and the run variable represents a different object from this set for each pass. In this way, certain definable sets of objects (such as customers, products, sales transactions) can be run through in a simple manner. Within the loop, you can then work with the respective object that is currently in focus and that is represented by the run variable.

Unlike counting loops, the traversal of a conditional loop depends on whether a condition, which is checked before (head-controlled loops) or after each pass (foot-controlled loops), is satisfied. This condition can also depend on values that arise during the loop passes, such as calculated values or user input. Therefore, with conditional loops, it is not necessarily possible to say before the first loop pass how many times the loop will be run in total.

■ Exercise 15.3

A counting loop is ultimately a special case of the conditional loop. The condition is that the value of the run variable moves within a certain value range (between the start value and the end value) or – if the set of objects is run through and the run variable represents the “current” object – that there are still objects left in the set of objects being run through that have not yet “had their turn”. Because a run condition also comes into play here, a counting loop with its run condition can also be formulated as a conditional loop. However, this does not work the other way round, because in the case of conditional loops it does not have to be clear in advance how often the loop is run, which is a prerequisite for a counting loop. Think of a conditional loop whose execution depends on user input. Since it is not foreseeable in which loop pass the user will make the crucial input that will cause the loop to terminate, we cannot tell how many times the loops will be run.

Of course, there are cases in which the number of runs can be known in advance, even for conditional loops, namely when nothing can change during the loop runs on those variables that decide in the run condition whether the loop will be run a second time or not. Then, even with a conditional loop, it is clear from the beginning how often it will run, and it could also be written as a counting loop.

■ Exercise 15.4

A graphical user interface is normally characterized by the fact that the user can choose from different actions. Each of these actions triggers an event to which the program can react with a corresponding event handling routine. However, this process can also be represented as a loop. To do this, we would read in an action from the user again and again in an infinite loop and then process this action within the loop. This could look stylized like this:

15.5 Solutions to the Exercises

```
abort = FALSE

While abort = FALSE
Begin
  action = read()
  Switch action
  Begin
    Case ...: ...
    Case ...: ...
    // Treatment of all possible cases
    Case "Exit": abort = TRUE
  End
End
```

If you are not completely familiar with the Switch-Case construct, simply go back a few pages to ► Sect. [14.6](#).



How Do I Search For and Fix Errors in a Structured Way?

Contents

- 16.1 Errors at Development Time – 194
- 16.2 Errors at Runtime – 195
- 16.3 Testing – 196
- 16.4 Debugging Methods – 197
- 16.5 Your Roadmap to Learning a New Programming Language – 199

Overview

Errors are the programmer's worst enemy. No matter how carefully you work, errors always creep in. Testing the program, to detect and eliminate errors, is therefore an essential part of programming. Errors manifest themselves either by the program not being able to run at all, by it crashing during execution, or by the fact that even if it runs to the end, it still doesn't do what it's supposed to.

A distinction can be made between errors that occur *at development time*, i.e., while the program is being written, and those that occur *at runtime*, i.e. when the program is being executed.

In this chapter you will learn:

- What types of errors there are and what causes them
- How to catch runtime errors by handling exceptions and make them harmless
- How to proceed skilfully when testing programs
- What a debugger is
- What the debugger features, such as “breakpoints”, “single step mode” and “variable observation/watches” are, and how to use them to understand the location and cause of errors
- How to use temporary, additional output to diagnose errors in the program even without a debugger.

16.1 Errors at Development Time

Errors that already happen at development time are either errors in syntax, where the “grammar” of the program is not correct, and one or more statements do not comply with the “sentence structure rules”, or they are algorithmic errors; errors that are due to the fact that the program code, while formally correct, is simply not an accurate implementation of what the programmer wants to do or achieve.

In the case of *syntax errors*, the compiler or interpreter regularly helps by aborting the compilation or execution of the program with an error message. This error message usually gives information about what kind of error has occurred and where in the code (often via the line number or the name of the program part) the cause probably lies. Such an error could be that a variable has not been declared, and if you then use the variable in the program, the compiler or interpreter will fail at this point. Sometimes, the error messages are somewhat cryptic, and only programmers who are familiar with the internal structure of the programming language will understand what the compiler/interpreter is trying to say. In such cases, a search on the Internet usually helps, because other programmers will also have wondered about this error message. You will find solutions for errors in many different languages on *StackOverflow*.

Algorithmic errors are often more difficult to detect and understand. To do this, use the debugging methods described in ► Sect. 16.4.

16.2 Errors at Runtime

■ Unforeseen Circumstances During Program Execution

The difference between errors that occur during the development of the program and errors that occur at runtime is that in the latter case the program works in principle, but during execution circumstances occur that the developer did not foresee and for which he did not take any precautions.

This could, for example, be that the program tries to access a file that does not exist. This problem never occurred during development, or during testing, because the files that the program wanted to access already existed. So, it never occurred to the developer to make a provision for this eventuality. A similar example, very common in practice, is a situation where the user makes an input that the program can't handle. For example, the user enters a text where a number is expected. The program, which otherwise works perfectly, now tries to calculate with the text, and crashes. During testing, the programmer always entered a number when the program asked him to. Accordingly, he never got into the situation where the program crashed because of non-numeric input, so never took the precaution of testing the user's input and rejecting it if it is not numeric.

Errors that only occur at runtime are regularly caused by a situation that can occur but was not foreseen during development. Under ideal conditions, the program runs without any problems.

The trick is to catch as many sources of errors as possible during development. For example, the program should test whether a user input that is for a calculation is a number, and otherwise reject it. Likewise, it should check that numeric variables to be used as divisors will never be 0, which of course immediately causes an error. The existence of files that the program wants to access can also be checked and the program flow adjusted if the file does not exist.

This anticipation of possible bugs is time-consuming but important, especially when writing programs for others to use. They may not be able to look at the program code, or may lack technical understanding, and may have a low tolerance of bugs, especially if they paid for the program.

■ Catching Errors with Try-Catch Constructs

Many programming languages support error handling in the program by special language constructs. The basic idea is that errors are reflected in *exceptions*. Exceptions are nothing more than events that are triggered when an error of a certain type occurs. In this context, programmers also speak of the exceptions being “thrown”.

Following the analogy of “throwing”, these exceptions can then be “caught” and handled. For example, an exception could occur when opening a file, if the file to be read from does not exist. In this case, the program does not terminate, but continues in a controlled manner with a part of the program that is intended for precisely this situation.

In our pseudo-code, such a construct would look like this:

```

Try
    // Open file and read data
Catch
    // Display error message
End

```

So, you “try” to execute a certain part of the program; if that works, it continues after **End**. But if an exception occurs, the code after the **Catch** keyword is executed first. Some programming languages allow you to catch different exceptions within such a construction and react differently to each of them. Often, this involves providing a special exception object whose properties can be used to provide more details about the exception that occurred, so that you can tailor your response to it.

Now, what is the difference between such a Try-Catch construct and an approach that gets along without this construct by simply checking whether the file exists before trying to open it? In the latter approach, the possible source of the error is checked beforehand, and the action that could lead to an error is not performed at all. With the Try-Catch construct no check takes place before the action, instead you run straight into the error. The only difference is that the error does not cause the program to crash or behave in an uncontrolled manner. Instead, the error is processed in a completely controlled way *after* it has occurred (after the exception has been thrown). The advantage of this approach is that you don’t have to know in advance exactly what could go wrong. Perhaps the file to be read will still not be readable, even though it exists, for example because the user has insufficient access rights. This error would also be caught with our Try-Catch construct, without us having to explicitly provide for it. The danger of this seemingly practical approach is, of course, that one thinks less about the possible sources of errors and the optimal response.

A better approach is therefore to catch as many potential problems and exceptions as possible separately beforehand, to react to them individually, and then finally to include a general Try-Catch constructs in the program for all exceptions that are not handled separately.

16.3 Testing

16

Given the many possible sources of error, *testing* is of course the key to success. Testing, testing and testing again. Testing is almost a science in itself, there are numerous different approaches and types of tests. In fact, tester is even a job description in software development. In some companies, developers and testers work in pairs. In line with the importance of testing, there are special tools to develop tests, sometimes even to perform them automatically, to document the results and to provide project management support for the elimination of errors.

Of course, you don’t have to go to that much trouble. Nevertheless, testing is important. In the box you will find some tips for better testing.

Tips

- Think about how a user would use your program and try out these “use cases”.
- Think about everything that could go wrong, that is, how the user could possibly use the program improperly.
- Expose the program to “extreme” conditions. During development, you tend to always use the same simple examples with which the program works flawlessly, because that’s naturally what you want: few problems during development. Work against this and be your own *devil’s advocate*.
- Test section by section *and* in context. Testing code passages (for example, a function) individually makes debugging easier, because in the case of an error you already know approximately where the cause must lie. Nevertheless, sometimes unforeseen interactions occur when you run the program completely and let the different code passages intertwine. Therefore, you should always follow both approaches, testing in sections and testing in the full program context.

16.4 Debugging Methods

If you notice that something about your program is not working, you start looking for the error. The activities around finding and fixing bugs is called *debugging*.

There are basically two approaches here; either you work with “on-board tools”, in particular with suitable auxiliary outputs in the program flow, or you use special tools, called *debuggers*. The latter are often part of the Integrated Development Environment of the programming language and convenient to use. However, there are also command line debuggers.

Basically, debugging is about two problems:

- To determine *where* the error occurs
- Determine *why* the error occurs.

It is not always the case that you can first find where the error occurs and then answer the question of why. Sometimes you first realize why the error occurs and then must search for the code that causes it. To answer both questions, you essentially have the following tools at your disposal:

■ Outputs

You can temporarily *add extra output* to your program that will be removed later, when the debugging is complete. These outputs can help you understand what parts of the program have already been run. For example, if you are faced with the problem of a program crash, and you don’t know exactly where your program is stopping, you might want to temporarily insert code that creates some extra output. Each time such an output appears, you know that the program has at least run through to this point without crashing. Sometimes it is also not quite clear whether, for example, certain conditions (such as if-then constructs) or loops are run through at all. Here, too, it helps to include a corresponding output to determine the posi-

tion, which is only run through if the program code in the If-Then construct or the loop is executed.

However, you can use temporary, additional output not only to determine what part of your program code is currently executing, but also to check what value certain variables have in the program flow. Understanding the contents of variables often helps diagnose the problem.

■ Breakpoints

If you are working with a debugger, you can set *breakpoints*. With breakpoints, when you start the program, it will only run up to this breakpoint. After that, you can decide whether you want to let it continue running or terminate it at this point. Again, if you follow up a program crash and the program gets to the breakpoint without breaking, then the problem is after the breakpoint.

■ Single Step Mode

Another option that is regularly available to you when working with a debugger is to execute programs in single-step mode. This means that only one program instruction is executed at a time. Only when you press a certain key or key combination, the next program statement is executed. The IDE shows you graphically in the program code which statement is the last executed. This also makes it easy to determine at which point a program terminates. However, if you have loops in your program that are repeated often, it is quite tedious to run through them in single-step mode. In this case, you should set a breakpoint after the loop, let the program run through to there, and then continue in single-step mode from there on. At least, if the error is not in the loop itself, this will save you a lot of keystrokes. However, if the problem is in the loop, you will have to single-step through the loop and check the values of any variables the loop might use.

■ Variable Watches

In conjunction with breakpoints or running your program in single-step mode, there is another feature that many debuggers support, *variable watches*. This feature allows you to view, sometimes even change, the current contents of a variable. So, if your program has reached a breakpoint, you can look at what value certain variables currently have. In single-step mode, this allows you to follow how the values of variables change from one program statement to the next. ■ Figure 16.1 shows such a variable observation for our simple Celsius-Kelvin conversion example. A breakpoint was established at line 60 of the program. The program would be executed up to this line. The next instruction that would be executed is the one in line 60, where you set the breakpoint (marked by a red dot in front of the line).

16.5 Your Roadmap to Learning a New Programming Language

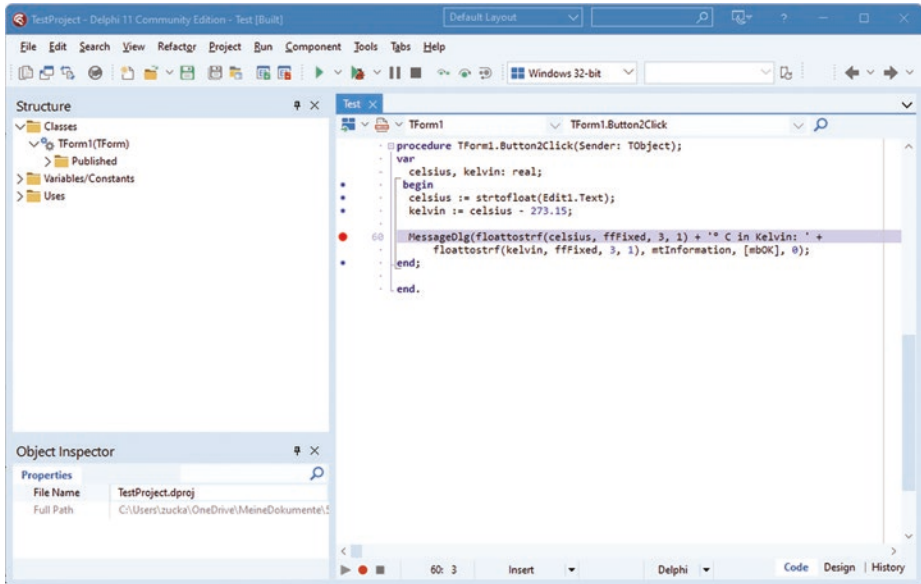


Fig. 16.1 Example of breakpoint with variable observation in Delphi

16.5 Your Roadmap to Learning a New Programming Language

When you learn a new programming language ...
you'll discover:

- what debugging tools are available to you and how you can access them (command line tools, integration in IDE),
- Whether, and if so how, you can set and remove breakpoints,
- whether, and if so how, you can execute your program in single-step mode,
- whether, and if so how, you can monitor the contents of variables.

Python



Introduction

Overview

Now we will start learning Python, our first programming language. As we learn, we will be guided by the 9 questions around the basic concepts of programming that we learned in the previous part. However, we will start with an overview of the origins and development of Python, its popularity and use.

Python was developed by Dutch mathematician and computer scientist *Guido van Rossum* starting in late 1989, apparently, out of boredom over the Christmas holidays (what do *you* do at Christmas?) The name is based on Van Rossum's beloved British comedy troupe *Monty Python*, so it has nothing to do with the snake whose stylized image can be seen in the Python logo.

The hobby project of Guido van Rossum has long since become one of the most popular programming languages around. Two of many indicators of this are Python's positioning in the *TIOBE index* (May 2023: 1st place, followed by C and Java) and the over 2 million question posts on *Stack Overflow*. For practically all internet and tech giants like Google or Amazon, Python is now one of the languages in constant use.

Python is a general-purpose language and is accordingly used for a wide variety of purposes, including—not least due to frameworks such as *Django*—applications on the web. Python has become extremely popular thanks to extension libraries such as *NumPy* for Data Science, and seems to be the de facto standard language for applications in the field of artificial intelligence, for which some significant Python extensions are available with libraries such as *TensorFlow* and *Keras*.

The popularity of Python is due in part to the fact that the language is relatively easy to learn. Python has a simple and straightforward syntax and encourages the development of easily understandable program code. Because of this, Python is considered by many to be the best beginner programming language. Numerous people, and you seem to be no exception, are taking their first steps towards programming today with the programming language originally developed as Van Rossum's hobby project.

Of course, Python has not been a hobby project for a long time. Since 2001, the *Python Software Foundation* has been the non-profit organization that takes care of the maintenance and further development of Python. Until a few years ago, Guido van Rossum played the leading role in this organization, which is why he was occasionally referred to as the “benevolent dictator for life”. By now, however, he has largely withdrawn from the activities.

The goal in this part of the book is not for you to write Python programs that are as efficient as possible, which every Python expert admires, or to make your programs look as “pythonic” as possible, but rather for you to learn how to write *functioning* Python programs that you *understand yourself* and that *fulfill their purpose*. After all, that's what programming is ultimately about: solving a given problem. The solution must work; if it also happens to be particularly elegant, very nice, but if not, that's fine too.

One tip before we really start: Try things out! Try not only to follow the examples presented in the text and the exercises, but also try things out for yourself, off the beaten path that the book gives you. You won't learn as much from anything else as by simply asking yourself the questions: What actually happens if I do this and that slightly differently? Curiosity and the willingness to try new things yourself will make learning any new programming language immensely easier. And Python is perfect for trying things out on your own.

But now enough of the noble words. Let us begin.



Tools & Help: What Do I Need for Programming?

Contents

- 18.1 Installing the Python Interpreter – 208
- 18.2 Installing the *PyCharm* IDE – 210
- 18.3 Getting Help with Python – 211
- 18.4 Summary – 212

Overview

First, we'll look at what you need to program with Python. Fortunately, this is not much at all, so we can quickly get started with the actual work.

In this chapter you will learn:

- How to install the Python interpreter
- how to install the PyCharm development environment
- how to use the Help function of Python
- how to use non-official sources of help for Python.

18.1 Installing the Python Interpreter

You can conveniently download the Python installer from ► <http://www.python.org>. The website is operated by the *Python Software Foundation*, the organization that coordinates the further development of Python. On the website, under “Downloads,” simply choose the latest Python version for your operating system. We're working with Python for Windows here in the book, but everything we'll discuss here applies to the macOS and Linux versions in much the same way. The Python that you then install is called *CPython*, the “official” Python version, if you will, in whose development Python inventor *Guido van Rossum* is involved. In addition, there are numerous other Python implementations. There are even Python interpreters that are written in Python themselves! *CPython*, on the other hand, was developed in the C programming language. And it is exactly this *CPython* that you are downloading now, even though it is not listed under this name on the ► python.org website.

When the download is complete, run the installer and follow the instructions on the screen. Make a note of the path where Python will be installed. You do not need to change anything in the default settings that the setup program provides. The installation is usually complete within a few minutes.

By the way, *CPython* only exists in English, which means that all messages, warnings, and other output are only generated in English; the Python programming language also uses only English terms. In this respect, you didn't miss the dialog where you can select the language during installation, because there isn't one! Apart from that, it is recommended to install the English version of all tools anyway, because this makes it much easier to search for help on the Internet if you don't understand an error message or want to find out how to perform a certain operation with the tool.

After the installation you are ready to go. If you change to the directory where Python has been installed, you can invoke the **Python** program there. A window will open that looks like the one in ■ Fig. 18.1.



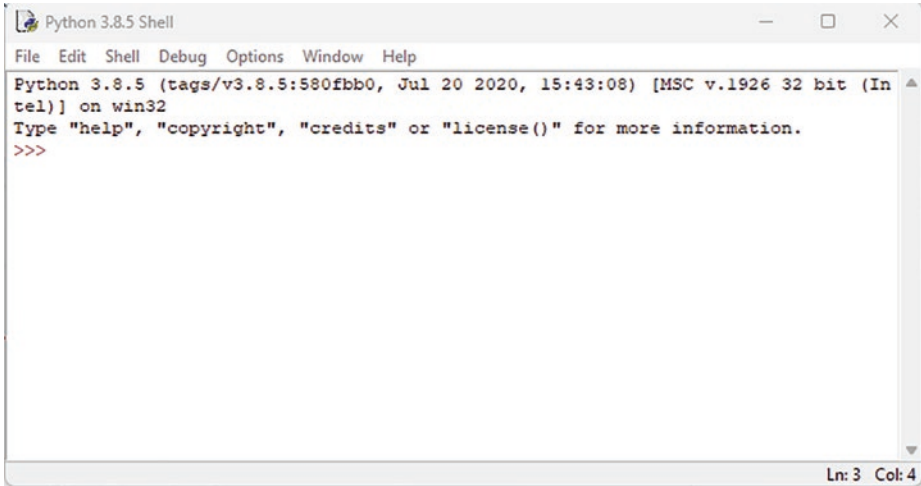
■ Fig. 18.1 The Python console after the first startup

This is the *Python console*. It allows you to run Python in *interactive mode*, that is, to type in a statement and execute it directly (scroll back to ► Chap. 9 if you no longer remember the difference between interactive and script mode). One such command that the console processes is **quit()**; it exits the Python console and closes the window.

Besides this interactive mode, Python of course also offers a script mode that allows us to write and run longer programs. Python code files usually have the extension **.py**. So if you have a program **myprogram.py** and want to run it, you go to the Python directory in the terminal/console of your operating system and type there:

```
python myprogram.py
```

Of course, it is more convenient to use an integrated development environment than to use the Python interpreter directly. Python comes with such an environment in the form of *IDLE*. However, it is rather rudimentary, even if it supports features such as syntax highlighting. You can get an impression of *IDLE* in ■ Fig. 18.2. In the following section, we will work with *PyCharm*, a much more powerful development environment, which we will install next.



■ Fig. 18.2 The Python development environment IDLE

18.2 Installing the *PyCharm* IDE

PyCharm is a popular integrated development environment (IDE) for Python, developed by the Czech software company *JetBrains*, headquartered in Prague. JetBrains specializes in development tools and offers some well-known IDEs such as *IntelliJ IDEA* (especially for Java), *WebStorm* (for web development, especially JavaScript), *PHPStorm* (for PHP) or *RubyMine* (for Ruby). They have also developed a programming language called *Kotlin*, which is mainly used in mobile app development.

You can download the free community version of *PyCharm* from the ► <https://www.jetbrains.com/PyCharm/> website. This is limited in functionality, but already offers considerably more features than we will ever be able to use in this book. Click the “Download” button and then choose your operating system; Windows, MacOS and Linux are supported. After downloading the setup program, start the installation process. You can leave the default settings here as well; however, it is recommended to at least associate **.py** files with *PyCharm* in the “Create Associations” section of the installer, so that they are always opened automatically with the IDE from now on. The installation usually does not take long but requires over 1.6 GB of free hard disk space. The times when you could install MS-DOS development environments from three or four 3.5-inch floppy disks with 1.44 MB capacity each are definitely over!

When you start *PyCharm* for the first time, you can configure in the “Customize” section whether you want to use the dark or the light color scheme. All illustrations in this book were created in the light color scheme for better printability; however, the author otherwise works in dark mode, which is a little less straining on the eyes. It’s best to exit *PyCharm* at this point first by closing the dialog. In the next chapter, we will start our first Python project. Then we will continue at this point.

18.3 Getting Help with Python

■ On the Internet

Python is one of the most popular programming languages around. So, it's no wonder that you can find countless resources around Python on the internet. One of these resources is the aforementioned platform *Stack Overflow*, where you'll find a ton of Python questions and answers that leave little to be desired. As of this writing, there are more than 2.1 million questions tagged **Python** on *Stack Overflow*. Today alone, 1,039 have been added. The probability that your question is also among the 2 million should be high. Therefore, *Stack Overflow* is a good place to start for most questions. The *Stack Overflow* hits are also regularly listed among the top search results on Google. You can, of course, add the **site** keyword to the Google search (**site:**► [StackOverflow.com](https://stackoverflow.com)) to narrow the search to *Stack Overflow*. But even beyond *Stack Overflow*, the internet is full of posts, blogs, tutorials, videos, and every other content format imaginable related to Python.

■ "Official" Python Help

Python itself also comes with official help. If you want to know how a particular element of the programming language works, you can refer to the documentation of classes and functions that is automatically installed with Python. You do this by calling the **help(element)** function in the Python console and giving it as an argument the name of the module, package, class, or function for which you want to see information. For example, calling **help(print)** returns the following help information:

```
>>> 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.
```

Start the console again as in ► Sect. 18.1 and try it out! Do not include the **>>>** as this is the prompt, indicating that the console is ready to accept your input. By the way, if you type **help()** without an argument, the console switches to the Python help system. The modified prompt **help>** indicates that you can now enter terms, which will then display matching help entries. You can leave the help system and return to the Python console by typing **quit** (without brackets).

The output of our **help()** call gives us some information about what the arguments of the **print()** function are, and most importantly, what **print()** does in the

first place. However, this help is rather rudimentary. If you want more details, it is best to search the Internet yourself.

The *Python Software Foundation* also offers a lot of information about the Python standard library (including the `print()` function) on its website ► [python.org](https://docs.python.org/3/library/functions.html#print), under the heading “Library Reference”: ► <https://docs.python.org/3/library/functions.html#print>). (Note, this is the same website from which we downloaded the Python interpreter.) The “Language Reference” explains the structure and syntax of the language, but is rather technical. The “Tutorial” section contains an introduction to Python, which is probably difficult to digest without prior knowledge of other programming languages.

18.4 Summary

In this chapter, we have looked at how to install Python as well as the *PyCharm IDE*. We have also learned about the most important ways to get help with Python.

Be sure to take the following points from this chapter:

- The “official” Python implementation, *CPython*, is further developed by the *Python Software Foundation* and is the variant of Python that you install by default when you download Python from ► python.org.
- Python comes with the Python interpreter program called **python**, which can be used either as an interactive Python console or to interpret entire Python scripts.
- With *IDLE*, Python also supplies a rudimentary IDE.
- *PyCharm* by *JetBrains* is a very powerful Python development environment, of which a limited community version is available for free.
- Help for Python is provided by Python itself with the **help(element)** function. It provides help texts for Python modules, classes, or functions, but they are often not very detailed.
- Further “official” help is available mainly in the form of the Language Reference and the Python Standard Library Reference at ► python.org.
- In addition, given the high popularity of Python, there are many rich sources of information available on the Internet; one of the most important—as for most other programming languages—is the developer forum *Stack Overflow*.



Program Execution & Hello World: What Do I Have to Do to Get a Program Running?

Contents

- 19.1 **Developing and Executing Programs in Python – 214**
- 19.2 **The Python console: Python in interactive mode – 221**
- 19.3 **Get to Know PyCharm – 222**
- 19.4 **Some Alternatives to PyCharm for Python Code Execution – 223**
- 19.5 **Summary – 224**

Overview

Now we finally start programming. We write our first small "Hello World" program in Python and at the same time familiarize ourselves with *PyCharm*, the powerful Python IDE we installed in the last chapter.

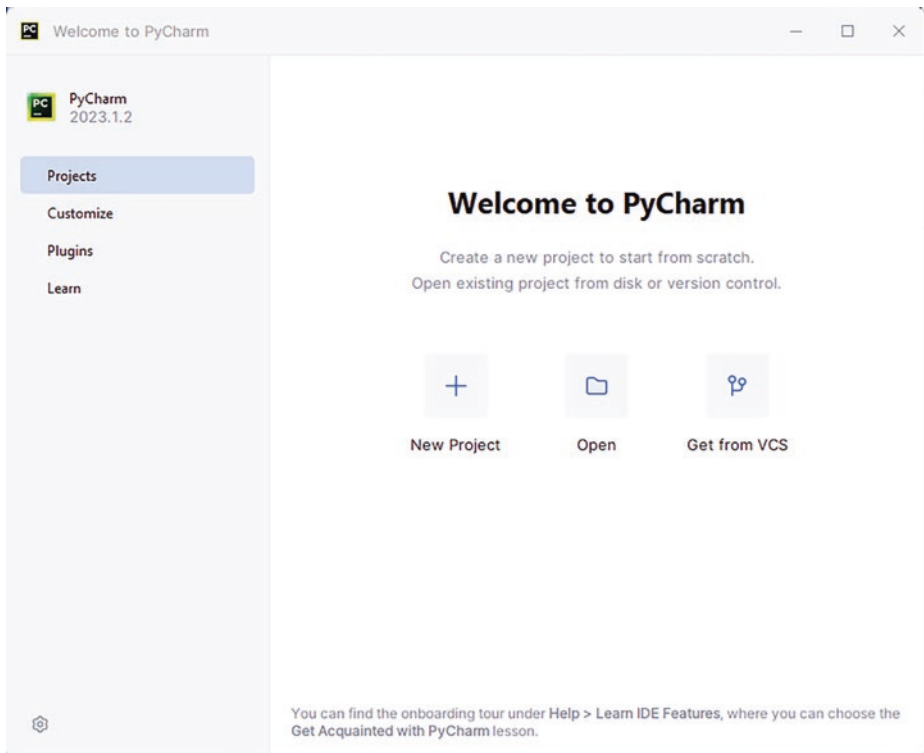
In this chapter you will learn:

- How the user interface of the *PyCharm* IDE is structured
- how to write and run programs with *PyCharm*
- how to use Python with *PyCharm* in interactive mode
- what a simple “Hello World” program in Python looks like.

19.1 Developing and Executing Programs in Python

■ Setting the Interpreter

After starting *PyCharm* for the first time, you will see a dialog that looks something like ■ Fig. 19.1. Since we want to write a new program, click the “Create New Project” button. *PyCharm* will now ask you for the Python interpreter you want to



■ Fig. 19.1 Start dialog of *PyCharm*

use. *PyCharm* can work with different Python interpreters. This is important if you want to maintain programs that only run on an older Python version. Rewriting these programs for a newer Python version might be very costly. Therefore, with *PyCharm*, you can simply use an older interpreter, and continue to run your aging programs without any problems. The ability to define the environment in which your program will run goes even further. Not only can you use a different interpreter than the most recent one, but Python also lets you choose which Python libraries (called modules and packages) your program will access, and in particular which version of those libraries it will use. This way, you can build a custom environment for your program to run in, called a *virtual environment*. We will discuss modules, packages, and virtual environments in more detail in ► Sect. 23.3. The main thing here is to determine the interpreter we want to use. For reasons of compatibility with old programs, in some cases it may make sense to work with an older interpreter.

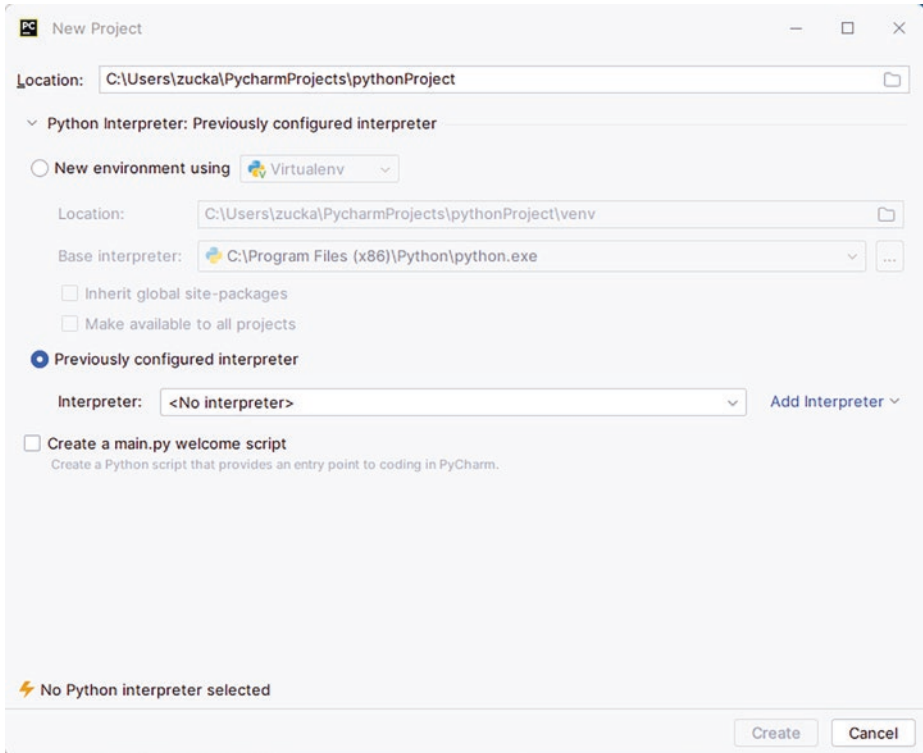
But don't worry. Normally, version jumps in Python don't lead to such major changes that previously written programs are suddenly no longer executable. However, with the transition from Python version 2.X to 3.X, there have indeed been major changes that in some cases have had this unfortunate effect. This is where *PyCharm*'s ability to handle multiple Python interpreters is a handy functionality.

Even if we want to work with the latest Python interpreter, we need to tell *PyCharm* where to find it. To do this, click “Previously Configured Interpreter” from the two central options. With the other option, you could create a virtual environment for your project. But we don't need to do that here. In the “Interpreter” selection box, you may currently have “<No interpreter>” (■ Fig. 19.2). If this is the case, click on the button with the three dots, select “Add Local Interpreter” and select the option “System Interpreter” from the dialog that opens (■ Fig. 19.3). There you should already find the path to the executable file of the Python interpreter. If this is not the case, you can always search for the executable file, which is conveniently named **python**, on your hard drive and select it manually in the dialog here. Now, after you have selected the interpreter for your project, you can specify under “Location” (at the very top of the “New Project” dialog, ■ Fig. 19.2) where *PyCharm* should save your Python project. Now you're ready to get started with Python, so click on “Create” and off you go!

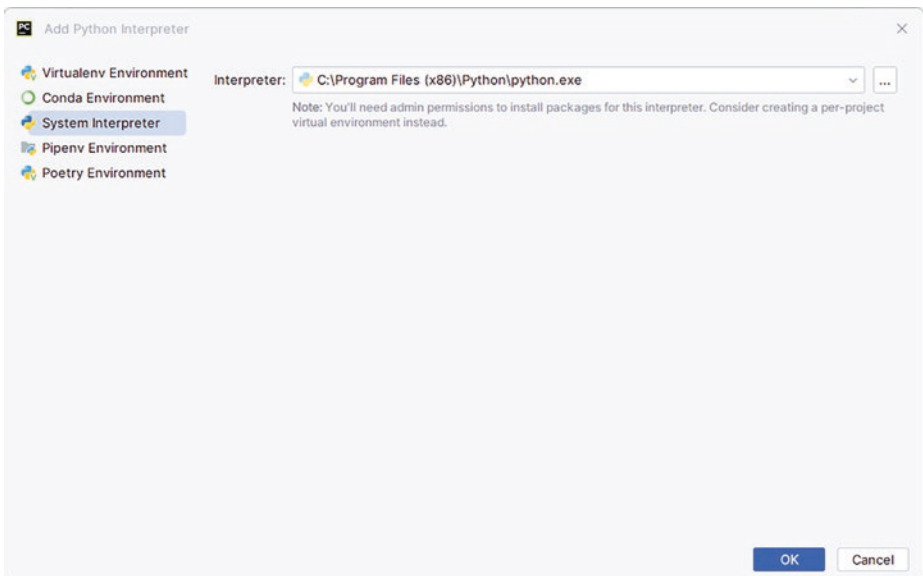
■ Create Python Programs with PyCharm

The *PyCharm* interface opens. In newer version of *PyCharm*, the menu is hidden behind the icon with the four bars on the left side of the program's title bar.

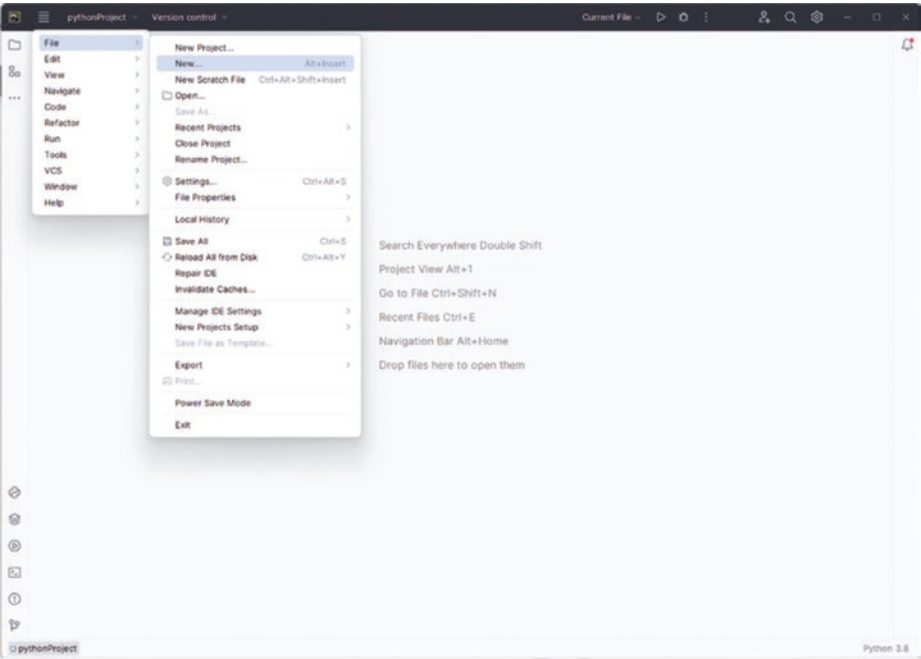
First, we create a new Python script file. To do this, click the “New...” option from the “File” menu and then select the “Python File” (■ Fig. 19.4). If that doesn't work, click the little folder icon on the top of the sidebar on the left, select your project and try again (■ Fig. 19.5). Now you need to give the file a name, for example **helloworld.py** (Python files generally have a **.py** extension). Your new Python script will now open in the right pane of the window. So, let's start programming!



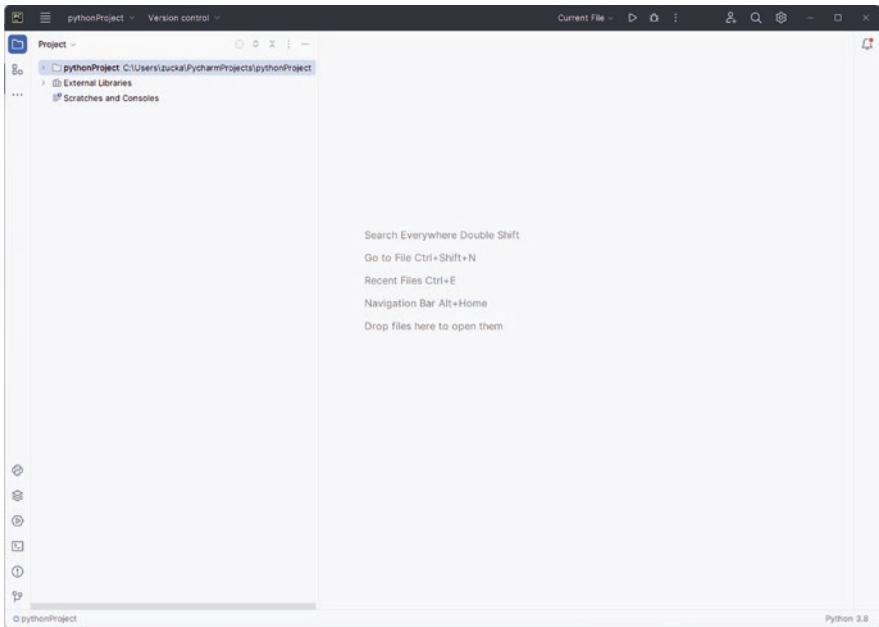
■ Fig. 19.2 Setting up the folder location of your new project in *PyCharm*



■ Fig. 19.3 Setting up the interpreter for your new project in *PyCharm*



■ Fig. 19.4 Creating a new Python file in *PyCharm*



■ Fig. 19.5 Creating a new Python file from the Project view in *PyCharm*

Enter the following lines of code into the script:

```
print("Hello World")
print("This is my first Python program")
```

You will notice that *PyCharm* supports you in typing, for example by immediately creating a closing parenthesis when you type an opening parenthesis. Also, pairs of parentheses that belong together are highlighted when you place the text cursor in front of or behind one of them. This is especially helpful if you have a lot of nested parentheses and want to know which opening and closing parentheses belong together. Every now and then, the IDE also displays little light bulb icons. If you click on them, you usually get hints on how to better format your code, and you can usually implement the formatting directly with a single click.

■ Running Python Programs: In *PyCharm* and on the Command Line

Now we want to start our program. To do this, right-click in the free space in the editor area and select the “Run” option from the context menu (alternatively, you can press <CTRL><SHIFT><F10>).

Now something is happening in *PyCharm*! At the bottom of the window, an area that resembles a tabbed page, titled “Run”, opens. This is the Run console, where your Python programs run. And as you can see, Python has run the little program you typed into the editor and has output two lines of text to the Run console. The output that appears here (using *PyCharm* on a Windows system as an example):

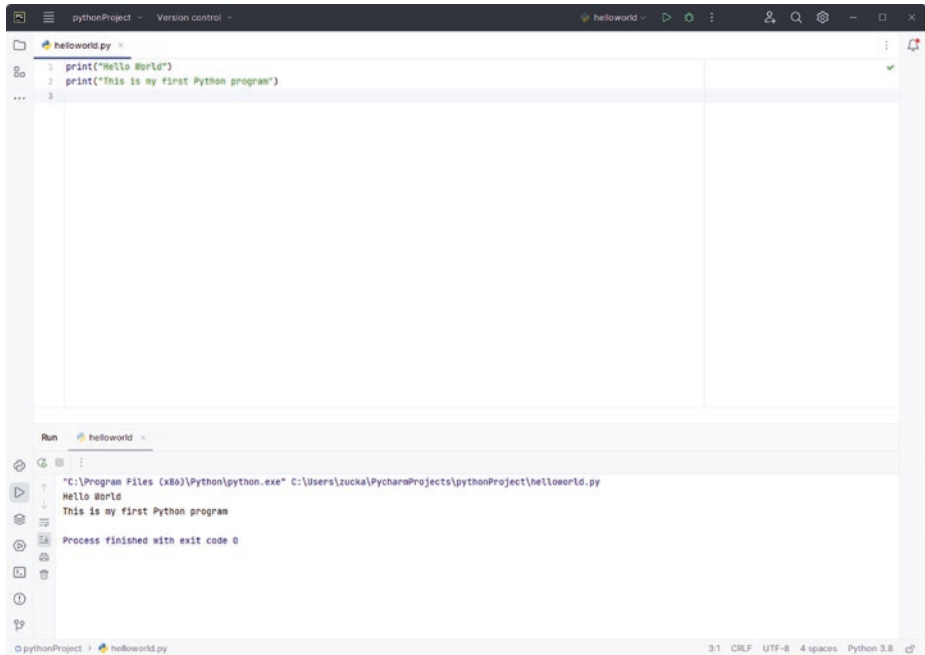
```
C:\[path to your Python interpreter]\python.exe C:/[path to
your project]/helloworld.py

Hello world
This is my first Python program

Process finished with exit code 0
```

You can see what this looks like in the *PyCharm* interface in ■ Fig. 19.6.

Instead of running your program through *PyCharm*, you could also have run it through your operating system’s terminal/console with the command **python helloworld.py** (you may need to add path information to the Python interpreter or your Python script, or both, depending on where they are in the directory structure). You can also try this out easily, because three icons below the play button-like icon (which represents the Run console) in the sidebar, there is a direct access to the command line level of your operating system with the “Terminal” tab. But don’t worry if you don’t have any knowledge of the commands used in the operat-

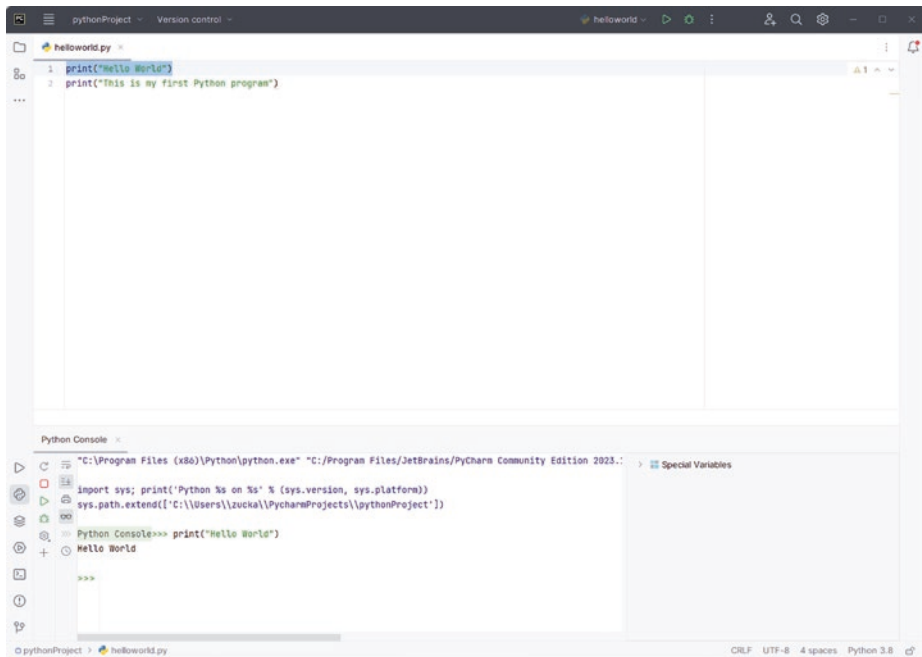


■ Fig. 19.6 Program execution in the Run console

ing system’s terminal, thanks to *PyCharm* you can get along without this at first sight archaic operation of the operating system: After all, you can simply run your program in the Run console (which, of course, does nothing but call the Python interpreter with your program script and display the output on the *PyCharm* interface). By the way, you will see the command that *PyCharm* uses for this as the first line in the run console. You could copy this line into the terminal/console of the operating system (or the “Terminal” tab in *PyCharm*) and execute it immediately.

In addition to this command and the output of our program, you will also see the message **Process finished with exit code 0** in the Run console. It means that your program has run through without errors. This is the optimal case, which we always want to have. However, as you continue with programming, you will notice that you are getting errors – and thus more program terminations – than you would like. Searching for and fixing errors is as much a part of everyday programming as sawdust is to the carpenter’s workshop – not nice, but unavoidable. This applies to beginners and professionals alike.

Therefore, let us now “artificially” cause an error! We currently have an executable program in the editor area. Just delete one of the closing brackets in one of the **print()** calls. Then run the program again. You can do this again with the Run command from the context menu (right mouse button), or – now that we had already executed the program – also with the green Play arrow in the Run console’s sidebar, right below its title.



■ **Fig. 19.7** Output in the Python console in *PyCharm*

If you have removed one of the closing parentheses, *PyCharm*, which automatically checks the syntax of your program in the background, indicates a possible problem with a red “underline” at the position of the missing parentheses. In the sidebar, you will see an exclamation mark icon with a red circle indicating there is a problem. If you click this button in the sidebar, the *Problem* tab will open you will get more information about the nature of the problem (■ Fig. 19.7). However, we will now deliberately ignore all these warnings and stubbornly run our (faulty) program. You will then get a red highlighted error message in the run console like the following:

```
File "C:/[path to your project]/helloworld.py", line 3
    ^
Unexpected EOF while parsing
Process finished with exit code 1
```

The error messages output by the Python interpreter are often of limited help in debugging, as you can clearly see in this example. The syntax check provided by *PyCharm* in the *Problem* tab is often much more useful. By the way, in the output you can also see the exit code with value 1, which signals that the program was terminated prematurely with an error.

Although Python is an interpreted language, and as such is normally slower than compiled programming languages, the programs that we will develop in the

next few chapters will usually run briskly. However, especially when working with looping constructs, which we will do in ► Sect. 24.4, it is quite possible for a program to take longer to run, especially if you have accidentally built your loop in such a way that it would never (at least not without external intervention or a lack of system resources) come to an end. That's when it's handy to be able to abort a program while it's running. This is exactly what can be done in *PyCharm* with the "Stop" button in the sidebar of the Run console. While your script is running, this button is colored red and can be clicked.

19.2 The Python console: Python in interactive mode

One of the tabs at the bottom of the *PyCharm* interface we haven't looked at yet is the *Python console*. It allows us to run Python in *interactive mode*. This means we can type in a Python statement and execute it immediately. So, it's not usually a whole sequence of statements that is entered and then executed en bloc (although that would be possible too), but just a single statement. The instruction is followed by Python's response (however precise), and you can enter new input. Because of this interplay between the input of an instruction and the processing of the instruction by Python, it is called interactive mode, and sometimes *REPL* (*read-eval-print* loop). The input in this is done at the *prompt*, which is nothing more than a request for input. In the Python console, the prompt is marked by three greater-than signs (`>>>`).

At the bottom of the *PyCharm* sidebar, click on the "Python Console" icon (the one with the Python logo on it, right below the "Play" icon symbolizing the Run console). Now, at the prompt, type one of the `print()` statements that are part of our script above and confirm by pressing <ENTER>:

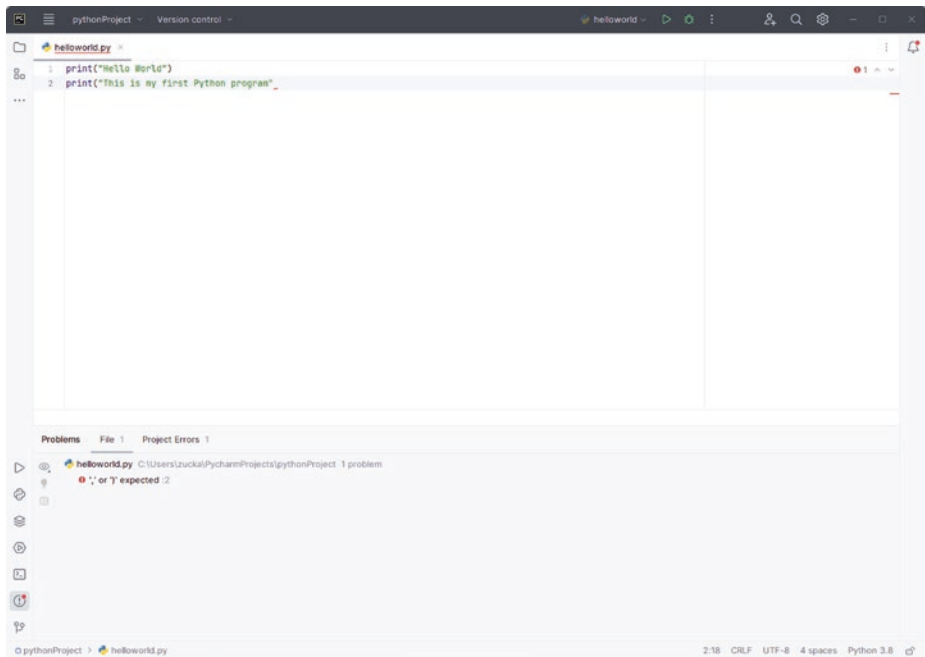
```
>>> print("Hello World")
Hello world
```

You can see that our instruction is executed directly. After that, a new prompt appears directly, where you could enter further instructions. ■ Figure 19.8 shows the Python console in *PyCharm* after executing our statement.

In the preceding code section, as generally in the rest of this part of the book, the prompt is symbolized by the string `>>>`. Please note that you must not enter these prompt characters! Anything not preceded by a prompt is output by Python.

You can use the console to quickly try out Python command instructions. Also, as we saw in the previous chapter, you can call `help` from here (try it out and type `help(print)` to get help information about the `print()` function.) Note that the Python console and the Python script editor are two completely different and neatly separated worlds. In particular, you cannot access variables that you use in your script from the Python console. But more about this in ► Chap. 21.

Of course, you can also start the Python console directly from the command line (console/terminal) of your operating system. To do this, simply run the `python`



■ **Fig. 19.8** Execution of statements in the Python console in *PyCharm*

program without any additional parameters. The Python console will then open in your operating system terminal. You can exit interactive mode again by entering the `quit()` statement, which terminates **python** and returns you to the operating system level.

19.3 Get to Know *PyCharm*

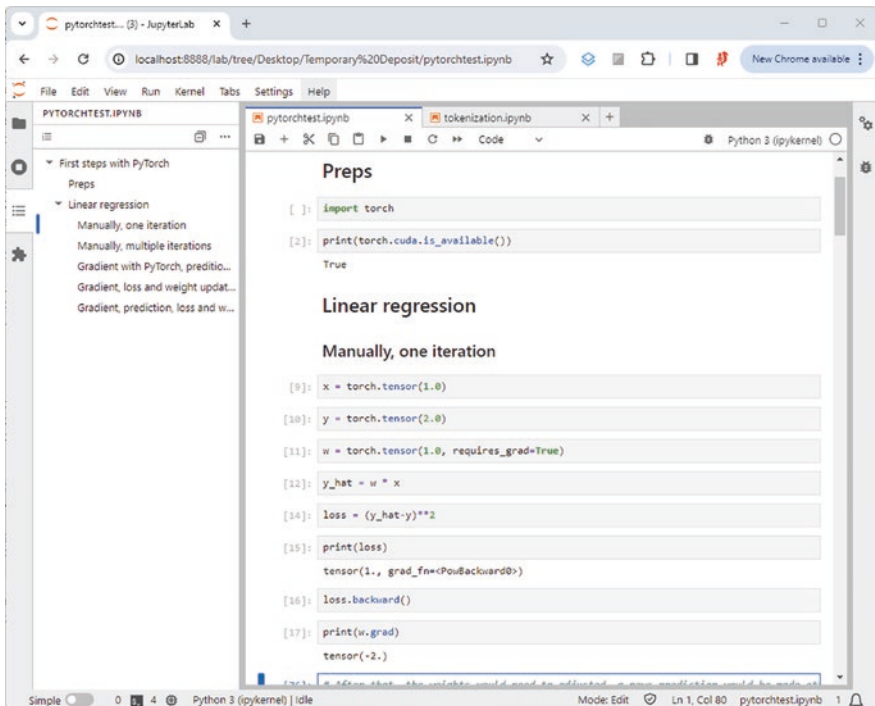
You have already learned about some of the features of *PyCharm*'s interface. Of course, *PyCharm* can do much more; however, we will not exhaust the full range of *PyCharm*'s features here. Some features are only relevant if you want to develop software on a professional level. And as with any traditional office application, even the most professional users don't take advantage of all the features the software offers them.

The *PyCharm* interface is, like that of many integrated development environments, quite complex with its different panes and tabs, some of which are nested within each other. Even though we already get by with a relatively modest set of features for our purposes, it's still a good idea to just play around with *PyCharm* a bit and get to know the interface better. Be curious and try things out. Fortunately, nothing can break during your explorations. In ► Sect. 23.3.3, where we talk about working with installed modules, and in ► Sect. 25.5, where we talk about debugging, we will return to the *PyCharm* interface and learn about some more features.

19.4 Some Alternatives to PyCharm for Python Code Execution

If using PyCharm is too complicated for you, you can of course also use another Integrated Development Environment. There is no shortage of alternatives on the market. The simplest is probably IDLE, the Integrated Development Environment that comes with the Python installation. It allows Python code to be executed in script/batch mode or interactively in the Python console. IDLE supports syntax highlighting, but is otherwise not very convenient. Nevertheless, IDLE serves its purpose well. So before you throw in the towel because you feel overwhelmed with the “battleship” PyCharm, try IDLE! Everything we do in this book can also be done with IDLE instead of PyCharm.

Jupyter is another popular way of executing Python code. Jupyter runs in a web browser and, similar to the Python Console, allows you to enter Python commands in interactive mode and view the results. However, not only text results are displayed, but also graphics, which can be very useful when working with statistical data. In addition to the Python code and the outputs, you can also place headings, explanatory texts and other elements. All of this together results in a so-called Jupyter notebook, which can be saved, changed at any time and executed again and again in whole or in part. The saved notebook can easily be shared with team members, colleagues or other interested parties. In this way, Jupyter is very useful for exploring data and documenting your work steps, for example. In addition to Python, Jupyter also supports a whole range of other languages such as R or Scala. Fig. 19.9 shows



■ Fig. 19.9 A notebook in Jupyter

an example of a Jupyter notebook with some code and output as well as some headings that are not part of the actual code, but are only used to structure the notebook.

If you are on the road and don't have a Python interpreter at hand, you can also run Python code with ChatGPT. ChatGPT has an environment in which Python code can be executed (formerly known as Code Interpreter). For example, ask ChatGPT: "Execute the following Python code and show the results". You can either write the code directly into the prompt or upload it as a file.

19.5 Summary

In this chapter, we looked at how you can use *PyCharm* to develop and run Python programs.

The following are the key points from this chapter:

- *PyCharm* is a powerful integrated development environment (IDE) with a wide range of features, only a small part of which we use in "normal operation".
- *PyCharm* offers practical features through syntax highlighting and live syntax checks that support the development of Python programs.
- *PyCharm* can handle different interpreters. It is therefore necessary that, before you start working, you select an interpreter that you want to use. You must do this even if you have only one interpreter available (namely the one you installed with the latest Python version).
- Python programs run in the *PyCharm* run console.
- In addition, there is the Python console, which allows Python to run in interactive mode; Python commands that are entered here are executed immediately.
- Python programs can also be run directly with the Python interpreter **python** without using *PyCharm*; if the Python interpreter is called without a Python script file as an argument, it starts in interactive mode. This can be exited at any time by typing **quit()**.
- You can use the **print()** function to produce output on the screen.



Syntax, Comments, Code Style & Documentation: How Do I Make Sure That I (And Others) Still Understand My Program Later?

Contents

- 20.1 Design of the Program Code and Naming Conventions – 226**
 - 20.1.1 Indentation and General Code Formatting – 226
 - 20.1.2 End of Statement Without Semicolon, Statements Over Several Lines – 228
 - 20.1.3 Case Sensitivity and Choice of Identifiers – 229
- 20.2 Comments – 230**
- 20.3 Documentation with Docstrings – 232**
- 20.4 Summary – 234**

20

Overview

Before we really get into Python programming, we will first look at what Python code and its most important basic elements look like, and which fundamental conventions must be observed when writing code. Compared to many other programming languages, Python has a special feature in code design that, at first glance, seems to take away some design freedom from us as programmers, but at the same time makes our lives easier.

In addition, we will look at how Python code is *commented* and *documented*. Commenting is important so that we as developers can still understand our own code later, especially if we want to develop it further. Documentation is used to provide information so that other developers who want to use our code in their programs understand exactly how to do so.

In this chapter you will learn:

- What meaning line indentations have in Python
- how to terminate statements in Python and how to extend statements over multiple lines
- how identifiers for variables, functions, methods, and modules are usually chosen in Python and what restrictions there are in doing so
- how to formulate comments in Python
- what docstrings are and how you can use them to document the program code.

20.1 Design of the Program Code and Naming Conventions

20.1.1 Indentation and General Code Formatting

We saw in ► Sect. 10.2 that many programming languages give the developer a lot of leeway in how the source code should be formatted. By choosing suitable indentations, for example, one can make the program code clearer. This freedom of design is ultimately due to the fact that many programming languages completely ignore the formatting of the program code. In other words, the formatting has no significance for the content of the program.

This is different in Python. Where other programming languages have special symbols to mark the beginning and end of code blocks (for example, **begin** and **end**, or open and closed curly braces), Python uses indentations. Statements that are indented the same distance belong to the same code block.

Here's an example of a simple function that you can pass a text and a number of repetitions, and it will simply output the text to the screen the appropriate number of times (in uppercase) and announce the current repetition number. Don't worry that you may not yet understand all the code, you will by the end of this part of the book. Here it's just about the indentations for now:

```
def repeat_text(text: str, reps: int):  
    text = text.upper()  
    for x in range(1, reps+1):  
        output = 'Run no.' + str(x) + ': ' + text  
        print(output)  
  
repeat_text('Hello world', 3)
```

After the function has been created with **def**, it is then called in the last line. The program generates the following output with this function call:

```
Run No.1: HELLO WORLD  
Run No.2: HELLO WORLD  
Run No.3: HELLO WORLD
```

You can see that the code block belonging to the **repeat_text()** function is indented. The same applies to the code block that is executed in the **for** loop and causes the output. This block is even indented twice, it belongs to the function **repeat_text()** and within that function to the **for** loop **for x in range(1, reps+1)**: After the end of the function definition, **repeat_text('Hello world', 3)** continues on the far left again. This statement does not belong to the code block of the **for** loop, not even to the code block of the function definition.

So, code blocks in Python are delimited by indentation (and introduced by a colon, as you can see in both the function definition using the **def** keyword and the **for** loop). Indentations are usually made either with the tab key or by entering four spaces. The second type is generally preferred. For the sake of consistent code, however, you should especially not mix both types of indentation in one program. Due to the enforced indentations, Python code is quite readable, so the loss of design freedom is tolerable. This way of delimiting code blocks eliminates the need to use parentheses or other keywords, the forgetting of which—especially at the end of a code block—is a common source of errors in other programming languages. Apart from the indentations, you are completely free regarding the design of the code.

In the form of style guidelines, however, there is a whole series of suggestions and recommendations on how to write Python code. Adhering to them helps to make the code more readable and understandable. But many of the rules are very detailed and are certainly not always followed, even by Python professionals. The official style guide can be found as a *Python Enhancement Proposal* (PEP) under the serial number PEP 8 on the ► python.org website. It doesn't hurt to take a look there.

If you are working with *PyCharm*, the handy “Reformat Code” feature from the “Code” menu allows you to have your code automatically formatted according to PEP 8 rules. Also, *PyCharm* alerts you to “violations” of the PEP 8 rules with wavy, gray underlines and small tooltip overlays, as you see in ■ Fig. 20.1.



■ Fig. 20.1 Tool tip insertion for code formatting

20.1.2 End of Statement Without Semicolon, Statements Over Several Lines

Python generally does not require a delimiter such as a semicolon at the end of a statement. Each statement simply ends automatically at the end of the line. Even if it is not advisable, due to the reduced clarity, you can also write several statements on one line in Python. In this case, however, you must separate the individual statements with a semicolon.

Conversely, there are also possibilities to extend an instruction over several lines. This makes sense if the statement is very long. Of course, you could also simply write the statement on one long line. However, you might have to scroll horizontally to see the end of the statement. This is uncomfortable and should be avoided. According to the code styling recommendations of PEP 8, lines of code should not be longer than 79 characters. Now, however, there are statements that are simply too long. So, what to do?

Statements can be easily wrapped within round, square or curly brackets:

```

repeat_text('Hello world', 3)
repeat_text('Hello World',
3)

```

We could also display the function call from our example above in two lines. However, it is not allowed to break within the string **'Hello world'**! Indentations are not regulated here, so you can indent as you like, which can greatly improve the readability of the program code.

But you can also break statements outside of round, square and curly brackets by placing a backslash (\) at the end of the line:

```
x = \
'Hello World'

print(x)
```

However, this method is frowned upon and should only be used when absolutely necessary. Mostly, the line breaks are necessary for function calls with many function arguments, and there you are inside round brackets anyway.

A final type of wrapped statement is the *docstrings* or, more generally, strings enclosed in triple quotes (we will deal with docstrings in more detail in ► Sect. 20.3). These can run over several lines:

```
x = '''This is a very long text which does not fit in one line
and therefore must be spread over several lines.'''
```

A “normal” string cannot do that. If we let a string run past the end of the line, the Python interpreter assumes we forgot to terminate the string, and this leads to an error message:

```
SyntaxError: EOL while scanning string literal
```

20.1.3 Case Sensitivity and Choice of Identifiers

Python is case-sensitive, which means that case is generally distinguished. Thus, a variable named **age** is completely different from the variable **AGE**. Of course, case-sensitivity applies not only to variables, but also to the identifiers of functions and methods, classes, modules, and packages.

Identifiers of all these elements can consist of letters, digits and the *underscore* (`_`). However, they must never begin with a digit. **customer_age** is therefore a permissible variable name, **take2** and **_2times4** as well, but not **11friends**. Identifiers that start and/or end with an underscore (or two underscores) have a special meaning in Python, which we will discuss in more detail later. It is recommended to generally avoid identifiers that start or end with underscores, and to use the underscore only *within* identifiers, unless you intend one of the effects associated with leading or trailing underscores.

Apart from these few rules, you are free to choose your identifiers. However, there are some conventions that many Python programmers adhere to, although violating these rules does not lead to syntax errors in the program code. Adherence to these rules makes your code look more “pythonic”. For example, class names or their components usually start with uppercase letters (so, for example, **MyClass**), module names with lowercase letters (for example, **myModule**), variables and func-

tions/methods usually with lowercase letters, with different components regularly separated with underscores (for example, **my_variable**, **my_function**).

You could even use special characters like umlauts in your identifiers, by the way. Python supports the extensive UTF-8 character set, which allows this. However, such identifiers are not recommended. At the least, code exchange with developers whose native language (and keyboards!) don't know these characters is made much more difficult. It is best to stick to identifiers that can easily be typed on a standard English keyboard.

Of course, you don't have to follow all these non-binding conventions. But if you do, your code will look more “Python” and other developers will get along better with your code, not least because they can concentrate on the meaning and functionality of the code and not regularly get “stuck” on the style of the identifiers they are not used to.

20.2 Comments

Comments, as you already know, are parts of the program code that are simply ignored by the interpreter and that you can use to leave notes and explanations for yourself or for others reading your program code. Such comments are introduced with the hash sign (**#**). Everything to the right of it is considered a comment. The comment symbol does not necessarily have to be at the very beginning of a line. In the following example you see both variants, once at the beginning of the line, once further to the right:

```
# Our first little Python program
print('Hello world')# Output on the screen
```

The comments that start somewhere in the middle of the line are also called *inline comments*. They are not welcome in Python, nor in most, if not all programming languages, because they make the code a bit more difficult to read. If you use them, you should make sure that there are some (officially recommended: two) spaces between the end of the actual code line and the comment symbol.

Of course, you can also use comments to temporarily “turn off” program code by removing it from the access of the interpreter, which ignores everything to the right of the comment character. It makes sense to speak of *commenting out* in this context. This is exactly what happened with the second print statement in this example:

```
# Our first little Python program
print('Hello World')
# print('Hello, world')
```

Comments always extend to the end of the line; Python does not know multiline comments in principle. So, if you want to write over several lines—and you will from time to time, because your comments should be fully readable on a normal screen even without scrolling horizontally (officially recommended are at most 79 characters per line)—you have to preface each line with an extra comment symbol. Many development environments do the work for you here, though. If you're working with *PyCharm*, you can highlight the lines you want to comment out, then click the "Comment With Line Comment" option in the Code menu (or alternatively press the <CTRL> and </> key combination) and *PyCharm* will add a comment symbol to the beginning of each of the selected code lines. You can also use the same option to remove the comment symbols from commented lines. Apart from that, there is also a trick you can use to create something like a true multi-line comment. More about this in the next section.

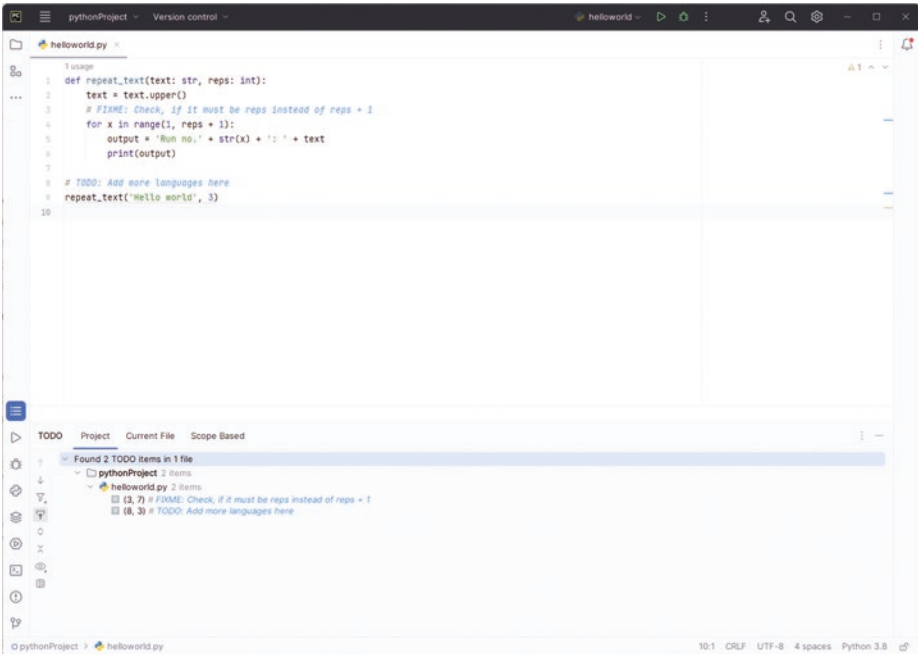
The sense and nonsense of comments can be philosophized and argued about, especially regarding the question about when comments are helpful and when they just make the code more extensive and thus less clear. Go back a few pages to ► Sect. 10.3, where we learned about a number of best practices for commenting program code. As a rule of thumb we can state: There is usually not enough commenting, so feel free to write comments; your “future self” who tries to understand your old code will thank you!

Sometimes you also use comments to document open tasks directly at the appropriate place in the program code. If you work with *PyCharm*, you can preface such comments with **# TODO**:

```
# TODO: Add more languages here
print('Hello World')
print('Hallo Welt')
```

Similarly, you can mark and explain small errors that still need to be fixed at the appropriate code location with **# FIXME**. Comments that start like this are specially highlighted as part of syntax highlighting and are treated specially by *PyCharm*. *PyCharm* displays these kinds of comments in a special sidebar pane called “TODO” as seen in ■ Fig. 20.2. All **TODO** and **FIXME** comments are displayed there; double-clicking on an entry in the TODO area will take you directly to the corresponding place in the code. If you can't see the TODO pane, click the three dots in the sidebar to see areas that are currently hidden.

If you want, you can also define your own comment types analogous to **# TODO** and **# FIXME** via *PyCharm*'s settings. In the vast majority of cases, however, the two special comment types mentioned above should suffice.



■ Fig. 20.2 The TODO pane in PyCharm

20.3 Documentation with Docstrings

In addition to the “real” comments, which are always introduced with the comment symbol `#`, there is a second possibility to store information in the code that is not executed by the interpreter, namely with the help of so-called *docstrings*. Docstrings are special strings written between triple quotes. Consider the following program as an example:

```
'''
This is a docstring for our hello world program, namely
one that even spans several lines.
'''

print('Hello world!')
```

If you run this program, you will get the output **Hello world!** The docstring, on the other hand, is not displayed.

Let’s try something else: This time we don’t use a docstring with three quotes, but a whole ordinary string, similar to **'Hello world!'**, which we output below. Unlike docstrings, ordinary strings in Python can’t span multiple lines in program code, so we’ve put our ‘comment’ on one line. Now what happens when you run this program? Nothing changes! Again, only **'Hello world!'** is output. The reason is

quite simple: whenever you write text into your Python program code, as we did first with the docstring and then with a normal string, this does not result in any output on the screen. Instead, the text is simply ignored (not technically true, but at least there is no visible effect). The same goes for variables, as you'll see in the next chapter. If we simply enter the name of a variable into our program code without any further instruction, nothing happens at all. To display the contents of the variable, or just our text from the example above, we must explicitly tell Python to output it. We do this by calling the **print()** function. It is different in the console. If you enter the name of a variable there and press <ENTER>, the contents are displayed. If you enter a string in the console, the string is immediately displayed again in the console.

So, like all other strings, docstrings, if they are in the program code without any other instruction, will not be displayed on the screen. If this is so, can't we just use normal strings to explain our code? Yes, that would work. However, docstrings have two special properties that make them particularly suitable for *documenting* program code, as their name suggests: First, they can span several lines: We've already seen that. Second, Python treats these docstrings in a special way. If they are at the beginning of a function, class definition, or module, they are used as the contents of the help for that function, class, or module. If you call the **help()** function (for example, for **print()**: **help(print)**), you will see the docstring that is stored at the beginning of the respective program code.

There are numerous Python tools that process these docstrings and output them in the form of documentation. Python's own help uses a tool called *pydoc*, which extracts the docstring from the code and displays it when **help()** is called. In addition, there are several other utilities that work with docstrings, for example *autodoc*, *doxygen*, and *pydoctor*. Some of these programs are specifically designed for Python, others allow the automatic generation of code documentation for different programming languages. The output of this documentation does not have to be just text in the Python console. Quite a few tools also support the generation of HTML, PDF or even LaTeX documents. To structure the documentation cleanly, some tools require a special structure of the docstrings.

Of course, clever minds have long since considered the question of what docstrings in general should look like. The official answer to this question can be found in the *Python Enhancement Proposals* (PEP) as PEP 257 (*Docstring Conventions*).

Unlike comments, docstrings are used more to document code *for other users*, that is, to explain how to *use* the code for your own purposes, and less to make notes about how the code *works*. That's what comments are usually used for. We will come back to docstrings in some later places and use them in our programs.

We will also take a closer look at another type of documentation later. These are the *function annotations*, which allow you to document the expected data types of function arguments, as well as the data types of the return values of functions in the program code. The annotations are also used by *pydoc* for help and can also be processed by other documentation tools.

Docstrings and function annotations are often aimed at an audience other than yourself, namely the *users* of your code. When writing Python code to be used by other developers, it is important to document what, for example, functions do,

what their arguments mean, and what return values the developer using your code can expect. He doesn't want to have to read your code to find out this information. That's why it's important to provide documentation that allows third parties (if that's provided) to use your program code "blindly" and view its inner workings like a black box. Relevant for the users of your code is only the *interface*, i.e., how to use it, not how it works in detail. Docstrings and function annotations are suitable tools for this.

20.4 Summary

In this chapter we have looked at the special features of Python when it comes to formatting the program code, how identifiers (for example, the variables and functions/methods) may/must be structured, and how you can comment and document your program code.

You should definitely take away the following points from this chapter:

- In Python, indentations mark a coherent block of code and can therefore not be used arbitrarily; Python already "enforces" an easily readable formatting of the program code, so to speak.
- Statements usually end at the end of a line in Python, no special delimiter is needed.
- Statements can go over several lines if the line break is inside round, square, or curly brackets, or is specially marked by the backslash (`\`), although the latter type of line break is rather rare.
- Several statements can be separated by semicolons on a single line.
- Python is case-sensitive when defining variables or identifiers.
- Identifiers are allowed to consist of letters, numbers, and the underscore.
- Unlike digits, letters and underscores may also appear at the beginning of an identifier.
- Since single and double underscores at the beginning (and also at the end) of identifiers have a special meaning, identifiers beginning with underscores should generally be avoided.
- Identifiers of variables and functions/methods are usually written in lowercase in Python, and their components, if they are compound terms, are separated from each other with underscores.
- Comments explaining the code are always single line in Python and are introduced with the comment symbol `#`; anything to the right of this is not interpreted as program code but is ignored by the interpreter.
- Docstrings, which are enclosed in triple quotes, are used in Python to provide documentation for users of the code; they are the essential part of Python help, which can be called via `help()`, and are further processed into various formats of documentation by numerous developer tools.
- Docstrings can span multiple lines.



Variables & Objects: How Do I Store Data to Work With?

Contents

- 21.1 Creating and Assigning Variables – 236**
- 21.2 Deleting Variables – 238**
- 21.3 Basic Types of Variables – 238**
 - 21.3.1 Numbers (int, float) – 239
 - 21.3.2 Strings (str) – 239
 - 21.3.3 Truth Values (bool) – 241
 - 21.3.4 None – 242
 - 21.3.5 Other Data Types – 243
- 21.4 Variables as Objects – 244**
 - 21.4.1 Attributes and Methods of Variables – 244
 - 21.4.2 Creating Variables with the Constructor Method – 248
- 21.5 Converting Variables – 249**
- 21.6 Complex Data Types – 251**
 - 21.6.1 Lists – 251
 - 21.6.2 Tuples – 258
 - 21.6.3 Dictionaries – 259
 - 21.6.4 Sets – 262
- 21.7 Self-Defined Classes – 264**
 - 21.7.1 Defining and Using Classes – 264
 - 21.7.2 Deriving Classes from Other Classes – 266
 - 21.7.3 Avoid Ambiguity: Name Mangling – 268
- 21.8 Summary – 268**
- 21.9 Solutions to the Exercises – 271**

Overview

In this chapter, we'll shift our focus to handling variables in Python. Additionally, we'll explore how to aggregate various variables into list-like structures (arrays) and map key-value pairs in hashes (dictionaries). Given Python's object-oriented nature, where all variables are objects possessing attributes and methods, our discussion will extend beyond merely creating, assigning, and converting variables across different data types. We'll delve into Python's implementation of object orientation and demonstrate how you can leverage object-oriented programming in Python to benefit your projects.

In this chapter you will learn:

- How to create variables in Python and assign values to them
- what basic types of variables there are, and how to use them
- what the object character of variables expresses and what this means for your practical work with variables
- how to convert variables from one data type to another, and where Python does the conversion for you automatically
- which more complex data types (for example, lists and dictionaries) exist and how you use them
- how class definitions work in Python, and how to define and work with object classes yourself

21.1 Creating and Assigning Variables

Unlike some other languages, creating variables in Python is quite simple. This is because variables do not need to be declared in Python; they are simply created automatically the first time they are used. Thus, the assignment

```
>>> x = 5
```

creates an (integer) variable and sets its value to 5 (the `>>>` is the prompt character that prompts you for input, so you don't enter this!)

We have named our variable `x` here for simplicity. In ► Chap. 11, we said that variable names should ideally be meaningful and allow the reader of the code to guess what kind of content the variable will have. Even though we're making it very simple here in these examples, Python gives you every opportunity to use meaningful variable names. As you already know from ► Sect. 20.1.3, names in Python can consist of upper and lower case letters, digits, and the underscore character. Numbers are not allowed at the beginning of the variable name but can be used everywhere else in the name. In addition, the underscore at the beginning (and sometimes also at the end) of names has a special meaning in Python, which we will discuss later. It is therefore advisable not to start or end variable names with an underscore. Otherwise, however, you are completely free to name your variables as you wish.

Python decides for itself what type the variable should have. Over the lifetime of the variable, the type may well change, for example, by assigning a different type of data to the variable. With

```
>>> x = 'A string (str) variable'
```

not only the value but also the data type of the variable is changed, it is now a string variable. In the next section, when we look at the object character of variables, you will learn about a second way to create variables in Python beyond assigning values.

If you are working with the Python console, you can display the value of a variable at any time by typing its name.

```
>>> x  
'A string (str) variable'
```

If, on the other hand, you write your code in a Python script, you must use the **print()** function, which you learned about in the previous chapter, to print the contents of the variable:

```
print(x)
```

If you write only the variable name in your Python program, there is no output.

By the way, you cannot access variables that you create in your program in the console. The namespace of the console and that of your program are separate. If you use a variable in the console that you have created in the program, you will receive an error message (unless, of course, you have already created a variable with the same name via the console—but then you would also work with this variable and not with the one that you use in your program).

Sometimes you will accidentally access a variable that does not exist, for example because you mistyped the identifier. Then you will get an error message like the following:

```
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
```

The crucial thing here is the last line. It tells us that a variable called `y`, which we have tried to access here, does not exist at all.

21

21.2 Deleting Variables

Once created, variables can be deleted again with the help of the **del** command. This makes sense, especially if the variable occupies a lot of memory (for example, if you have completely read in a large file) and you want to release the memory again after you no longer need the data.

If you delete a variable in the Python console and then try to access it, you will get an error message:

```
>>> del x
>>> x
Traceback (most recent call last):
  File "<input>", line 1, in <module>
NameError: name 'x' is not defined
```

The error message talks about the *name* `x` not being defined. Like many other programming languages, Python makes a clear distinction between the value of the variable and its name. The name is just a reference to the value, which is located in a certain area of memory. In principle, the name and value exist independently of each other. Now, it could be that several names refer to exactly the same value, that is, to the same location in memory. If the value stored in the memory in question changes, the values of all these variables also change accordingly. In such a situation with multiple names all pointing to the same memory location, if you delete one name, the value and the other names (and therefore variables) are preserved. You can then simply no longer address the value under the deleted name, but only under the remaining names.

Python counts the names that point to a value (the result is called the *reference counter*). If there is no more name pointing to a certain value, Python deletes the value itself. This process is called *garbage collection*. However, since there will usually be only one name that has a *binding* to the value of your variable, when you call the **del** command, the value itself is usually also deleted and the memory in question is freed.

21.3 Basic Types of Variables

In this section we deal with the most important types of variables. We will first concentrate on variables that contain only one value. In the following section, we will look at more complex data types that can hold multiple values at the same time.

21.3.1 Numbers (int, float)

Numbers in Python come in the form of integers (**int**) or floating-point numbers (**float**). Unlike many other programming languages, **int** variables in Python do not have a fixed range of values beyond which another data type that takes up more memory must be chosen. Python simply reserves as much memory for the variable as is needed to hold the value it contains.

You can see this very nicely by looking at the size of the variables using the **sys.getsizeof(object)** function:

```
>>> x = 5
>>> sys.getsizeof(x)
14

>>> x = 1000000000000
18
```

As you can see, the memory requirement has increased from the original 14 bytes to 18 bytes after we assigned a significantly larger value, one trillion, to the variable instead of 5.

You may wonder why even a small value like 5 still requires 14 bytes in memory. In many other programming languages such a variable would have the size of only 2 bytes (= 16 bits). This can represent numbers between 0 and $2^{16} = 65,536$. So why is Python such a “memory hog”? The answer has to do with the way Python stores variables and will be discussed in the following section.

The decimal separator for floating point numbers is the period, as is customary in English. The biggest problem with this in Python is that if you habitually use the comma instead of the period (as it is common in many European and South American countries), you don’t get an error message:

```
>>> pi = 3.1415926535
>>> pi
(3, 1415926535)
```

Python misinterpreted our input and created a variable of a completely different type, namely a *tuple*. We will deal with these tuples in more detail a little later in this chapter.

21.3.2 Strings (str)

Strings, variables of type **str**, can be enclosed in Python in either single or double quotes:

```
>>> x = "A text in double quotes."
>>> x
'A text in double quotes.'

>>> y = 'A text in single quotes.'
>>> y
'A text in single quotes.'
```

The advantage of being able to use both types of quotation marks is that in Python you have no difficulty in displaying quotation marks within a text, because the two different variants of quotation marks mean that there is no danger of confusion between the quotation marks that are part of the text and those that delimit the string at the front and back:

```
>>> quote = 'Hamlet said, "To be or not to be. That is the
question here!'"
>>> quote
'Hamlet said, "To be or not to be. That is the question here!'"
```

As you will have noticed, Python automatically encloses the output of the variable's contents in (single) quotes to make it clear that this is a string. The following example shows that this is a practical feature:

```
>>> x = '5'
>>> x
'5'

>>> x = 5
>>> x
5
```

In the first assignment, the content of the variable is a string, in the second case it is a number, which could now be used in a calculation.

In Python, it is very easy to create strings that go over multiple lines. To do this, simply enclose the text in triple quotes:

```
z = """The text starts on the first line
and continues on the second line."""

print(z)
```

This program results in the following output:

```
The text begins on the first line
and continues on the second line.
```

In fact, the line break is preserved in the output. You can use this feature not only when you are working in script mode, i.e., writing a program to execute it afterwards. Even in interactive mode, Python recognizes after pressing the <ENTER> key that you have started a multiline string here, and therefore waits with the execution of the statement (which is, after all, normally triggered with <ENTER>) and allows you to continue writing on the next line instead.

You have already become acquainted with this type of string as *docstrings* in the previous chapter. Docstrings are placed in your program code as documentation, but not with the intention of being further processed or output to the screen for the end user of the program.

Sometimes you want to wrap strings in the program code without this wrapping being visible when the string is output; the point is simply to make the program code clearer (remember the recommended limit of 79 characters per line from ► Sect. 20.1.1!). In this situation you can use backslash (\):

```
message = 'Hello ' \
'world'
print(message)
```

This code generates the output:

```
Hello world
```

This means that there are not two strings on two different lines, but only one string, which is distributed over two lines in the code for practical reasons only.

21.3.3 Truth Values (bool)

Truth values, or the logical values true and false, are represented in Python with the data type **bool**, a reverence to the English mathematician and logician George Boole, already mentioned in the first part of the book, who made a significant contribution to the development of formal logic in the nineteenth century.

Unlike the other variable types, variables of type **bool** can only take on two values: **True** and **False**. Pay attention to upper and lower case! The constants **True** and **False** must each be written with a capital letter. If we were to write **false** instead,

Python would assume that we wanted to use a variable called **false**, which of course does not exist:

21

```
>>> x = false
Traceback (most recent call last):
  File "<input>", line 1, in <module>
NameError: name 'false' is not defined

>>> x = False
>>> x
False
```

Also, do not use quotes because they would make the variable a **str** variable:

```
>>> x = 'False'
>>> x
'False'

>>> type(x)
<class 'str'>

>>> x = False
>>> x
False

>>> type(x)
<class 'bool'>
```

Python stores the values **True** and **False** internally as 0 and 1, so you can calculate with them just like with normal numbers:

```
>>> x = 5 * True
>>> x
5
```

21.3.4 None

A special data type is **NoneType**. You cannot create your own variables of this type. Instead, Python has already created an object of type **NoneType** for you, namely **None** (how creative!). This allows you to assign the value **None** to a variable, which means that this variable currently has no real, meaningful value:

```
>>> x = None
>>> x
None

>>> type(x)
<class 'NoneType'>
```

But isn't that a bit awkward? Couldn't we just assign the variable the value **0** if the variable is a number, or "", i.e., an empty string, if it is a string? Of course, we could indeed do that, but only if the values **0** or "" have no meaning in terms of content. However, when you measure temperatures, for example, or survey a person's attitude to a topic on a scale from -5 to $+5$, the value **0** can well have a real meaning of its own. In this case, it makes a difference whether the respondent has given the value **0** and thus signaled a neutral attitude to the topic or has not answered the question at all (**None**). In order to make this distinction visible, it makes sense to have a special indicator for "no real value present", and this is exactly what the **None** value is.

You cannot calculate with **None**, by the way:

```
>>> None + 1
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'NoneType' and
'int'.
```

If **None** is evaluated as a logical expression, it is treated as **False**. So, **None** is really not good for anything other than indicating that a variable does not contain a real value.

21.3.5 Other Data Types

In addition to the data types discussed so far, Python inherently knows several other data types, such as **complex**, a data type used to represent the complex numbers known from mathematics, which consist of a real and an imaginary part.

Various packages (program libraries) that are not part of the standard language also provide their own data types. One example is the *NumPy* package, a library for efficient work with vectors and matrices that is an important foundation for work with statistical and machine learning methods, fields in which Python has found considerable use.

Not only are some new complex data types provided by *NumPy*, but also several basic data types are available with *NumPy*. For the already known data types **int** and **float**, for example, *NumPy* has its own alternatives, which are characterized by the fact that they do not adapt in their memory requirements to the variable

content, as the standard data types in Python do, but always occupy a fixed number of bytes in memory. This allows for very fast computation with such variables, especially with large fields of such variables, and efficient computation is an essential skill, especially in processing large amounts of data, such as in machine learning.

However, since we can get by for most use cases with the data types discussed here, we'll leave it at that and now take a closer look at the character of variables in Python in the next section.

21.4 Variables as Objects

21.4.1 Attributes and Methods of Variables

Python is strongly influenced by the object-oriented programming paradigm. Unlike some other programming languages that have also adopted this programming approach, such as C++, in Python even the simplest variables are objects.

A variable in Python is ultimately always the concrete instance of a *class*, for example the class **float** for floating point numbers. The class **float** has a whole series of methods and attributes that you can access via the concrete instance of the class, i.e., variable object. If you are no longer familiar with the terms class and instance/object, simply turn back a few pages to ► Sect. 11.7 and refresh your knowledge before continuing!

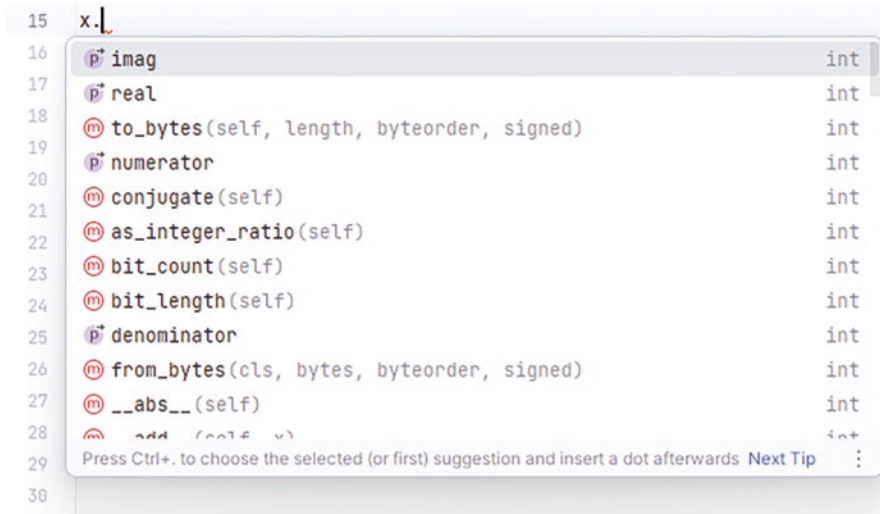
If you work with *PyCharm* and enter the name of a variable you already use in your program in the code editor, followed by a dot, a small code completion menu pops up where you can see the methods and properties that belong to the class of your variable. You can do the same thing in the console by typing a variable that you have already assigned a value to via the Python console, again followed by a period.

The dot is the operator in Python that allows you to access the methods and attributes of an object. Therefore, when you type a dot after the variable name, *PyCharm* expects you to use a method or attribute of that variable and displays the corresponding list. You can see an example of this in ■ Fig. 21.1.

The menu entries marked with a small “m” are the methods that the class **float** makes available to the object, the entries marked with a small “p” are the attributes (“p” for *property*). You can also see where the object got its methods or attributes from in the code completion menu on the right. In our case, all methods and attributes come directly from the class **int**, but some others will also come from the more general class **object**, from which the class **int** is derived. In a sense, the class **object** is the parent class for the class **int** and inherits methods and attributes from it.

Now create a **float** variable in the console and call the method **is_integer()** for this variable object:

```
>>> x = 5.3
```



■ Fig. 21.1 Code completion menu for an integer variable, called from the code editor

```
>>> x.is_integer()
False
```

The method checks whether the floating-point number is also an integer, which is of course not the case in our example here. The class method `is_integer()` does not need any function arguments, because it automatically refers to the object for which we call it, i.e. `x`. Although no arguments need to be passed to the function, it must still always be called with the (empty) round brackets that identify it as a function.

However, the class `float` has not only methods, but also some attributes. One of these attributes is `__class__`. It represents the class of the object:

```
>>> x.__class__
<class 'float'>
```

Alternatively, you can also determine the object type with the function `type(object)`, to which the object is passed as an argument:

```
>>> type(x)
<class 'float'>
```

With the help of the function `isinstance(object, class)`, which like `type(object)` is part of the Python standard library, you can determine whether a variable is of a certain type; with our object-oriented terminology more precisely formulated, you

can determine whether a variable is an *instance* of a certain *class*. To do this, the variable and the class to be checked for are passed to the function:

21

```
>>> isinstance(x, str)
False

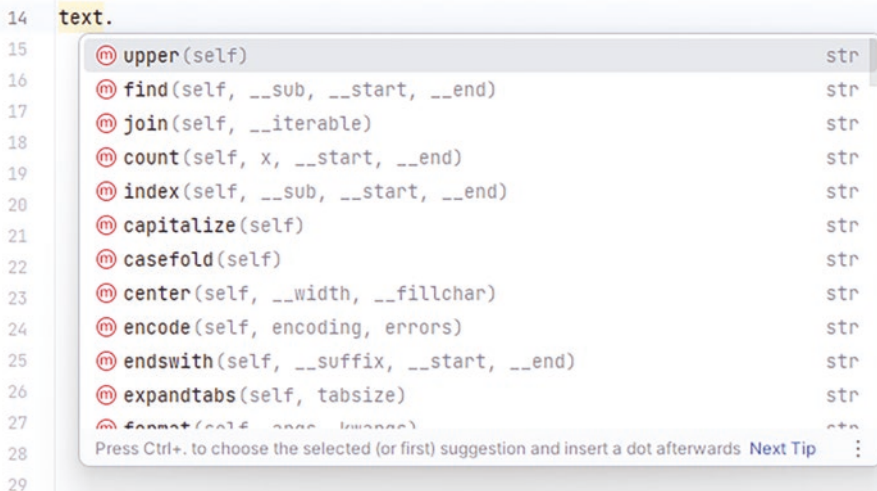
>>> isinstance(x, float)
True
```

Let's take a closer look at the methods of the objects, i.e., the functions that the class makes available to the object. A little earlier on, we already got to know **is_integer()**, a method of the class **float**, which checks the current object, the object *whose* method we are calling, to see whether it is an integer. As you remember, we don't have to pass the variable we want to check to this special function, because the method is already part of the object and therefore knows which object it should work with.

Now let's look at string variables:

```
text = 'Strings in Python have many interesting methods.'
```

If you are working with *PyCharm* and in the script editor, or in the Python console you can now type **text.** (with the dot operator) in the script editor or in the Python console, and the familiar context menu opens with the attributes and methods that the string class **str** makes available to your **text** object. You will immediately see that a rich selection of different methods is available here (■ Fig. 21.2).



■ Fig. 21.2 Methods of a **str** object

Let's try some of them out:

- **lower()** and **upper()** transform the string into lower and upper case letters respectively:

```
>>> text = 'An ordinary text, with a few words and punctuation marks.'
>>> text.lower()
'an ordinary text, with a few words and punctuation marks.'

>>> text.upper()
'AN ORDINARY TEXT, WITH A FEW WORDS AND PUNCTUATION MARKS.'
```

- **isnumeric()** checks if the string could be turned into a number:

```
>>> text.isnumeric()
False
```

- **count(substring)** counts the occurrences of the (partial) string **substring** passed as an argument to the method; it is case-sensitive, as you can see in the following example (only the “an” in “and” is counted):

```
>>> text.count('an')
1
```

- **replace(old, new, occurrences)** replaces the specified number of occurrences of the old string with the new string; specifying the number of occurrences to replace is optional, so you can omit it, which will simply replace *all* occurrences:

```
>>> text.replace('words', 'terms', 1)
'An ordinary text, with a few terms and punctuation marks.'
```

- **__len__()** determines the length of the string:

```
>>> text.__len__()
58
```

As you can see from the two leading and trailing underscores, this is a special core Python method.

? 21.1 [20 min]

Objects of type **str**, i.e., string variables, have several interesting methods other than the ones presented here as an example. Find out what other methods are available for **str** objects and try them out in the Python console. If you don't understand a method or can't use it successfully, don't dwell on it and move on to the next method.

You can read about the details of the methods in the help. To do this, call the help with **help()** in the Python console and put **str**. in front of the methods when calling them, so that Python knows exactly which class you want to look at for the method (there could be several classes that have a method of that name), for example **help(str.isnumeric)**.

You may have noticed during the exercise that methods such as **upper()**, **lower()**, and **replace()** do not modify the object for which they are called, but merely *return a modified copy of the object*. If you want to change the original object, you must assign the changed version, i.e., the return value of the method, to it. Let's take a closer look at this using **upper()** as an example:

```
>>> text = 'An ordinary text, with a few words and punctuation
marks.'
>>> text.upper()
'AN ORDINARY TEXT, WITH A FEW WORDS AND PUNCTUATION MARKS.'

>>> text
'An ordinary text, with a few words and punctuation marks.'

>>> text = text.upper()
>>> text
'AN ORDINARY TEXT, WITH A FEW WORDS AND PUNCTUATION MARKS.'
```

As you can see, calling the **text.upper()** method leaves the **text** variable completely unchanged. Only assigning the return value of the method to our original variable changes the object **text**.

21.4.2 Creating Variables with the Constructor Method

In the previous section, we saw that variables can be created simply by assigning a value to them for the first time. But there is another way to create new variables. As you know by now, variables are objects, that is, instances of a class. Like all classes, these objects have a constructor method, which is a special method that creates an object of that type. We can use these constructors to create variables. Let's look at the following example:

```
>>> x = int(3)
>>> x
3

>>> type(x)
<class 'int'>
```

So, the constructor method returns an **int** object with the value passed to it as argument. This alone is perhaps not all that interesting; after all, we could have achieved the same effect more easily with the simple assignment `x = 3`. What is interesting is that we can also pass a floating-point number or a string to the constructor and it will create an **int** object from it. In the case of the floating-point number, the decimal part is simply ignored. If a string is passed to the constructor, the text must of course be convertible into a number, otherwise we get an error message:

```
>>> 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 Converting Variables

It is not uncommon to need to change the data type of variables. To do this, consider the following example, in which we add an integer variable and a floating-point number variable:

```
>>> x = 2
>>> type(x)
<class 'int'>

>>> x = x + 3.7
>>> x
5.7

>>> type(x)
<class 'float'>
```


When we created the variable, Python automatically chose **int** as the data type because we had assigned an integer to the variable. But when we then added 3.7, a floating-point number, Python changed the type to **float** to accommodate the new value. So, Python *implicitly* converts without us having to intervene.

Now let's try something else:

```
>>> 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'.
```

If we add a string to our variable **x**, Python no longer implicitly converts. Maybe it works the other way around, where we define **x** as a string and add a number to it?

```
>>> 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
```

This doesn't work either. So, Python does not convert between numbers and strings implicitly. Nevertheless, we sometimes need to convert strings to numbers to compute with them.

Let's look at this with an example. Do you remember the conversion from Kelvin to Celsius from ► Sect. 12.2.2? There we discussed a simple program that takes a temperature in Kelvin as user input and converts it to degrees Celsius. We will now develop this program in Python. To do this, we make use of the **input(prompt)** function for input, which prompts the user for input, and returns that input in the form of a string. With this knowledge, it would be obvious to write code like the following:

```
temp_kelvin = input('Please enter a temperature in Kelvin: ')
temp_celsius = temp_kelvin + 273.15
print(temp_kelvin, 'Kelvin are', temp_celsius,
      'degrees Celsius.')
```

If we run this program and enter a temperature in Kelvin, we get output like the following:

```
Traceback (most recent call last):
File "C:/Users/MyUser/Python/var_examples.py", line 54, in
<module>.
temp_celsius = temp_kelvin + 273.15
TypeError: can only concatenate str (not "float") to str
```

By now you understand the problem that occurs here: The **input()** function returns a string, but it can't be computed with because Python doesn't convert to a number implicitly. We need to convert *explicitly*. That's exactly what we do in this modified version of the example:

```
temp_kelvin = input('Please enter a temperature in Kelvin: ')
temp_celsius = float(temp_kelvin) + 273.15
print(temp_kelvin, 'Kelvin are', temp_celsius,
      'degrees Celsius.')
```

This time we explicitly convert the string variable **temp_kelvin** into a floating point number, using the **float()** function. You will immediately recognize the reference to the last section: **float()** is, of course, the constructor method of the class **float** here as well. So, when we convert the string variable **temp_kelvin** to a floating-point number, we are doing nothing more than simply creating a new **float** object by calling the constructor of this class. We immediately initialize the new **float** object with a value; this value may also be a string, the **float** constructor then creates a floating point number from it. The *explicit* conversion in Python thus runs via the class constructors. They can receive as arguments not only a value of the type that the class constructor creates (**float** in our case), but also various other types (**str** in our example). In general, the explicit conversion thus has the form: ***datatype(value)***.

? 21.2 [10 min]

Write a program that takes the age of the user in years and outputs the number of minutes the user has lived (at least).

21.6 Complex Data Types

21.6.1 Lists

21.6.1.1 Creating and Displaying Lists

Unlike many other programming languages, Python does not know the concept of arrays as such. Instead, however, Python has a more general type of variable fields of which the array is ultimately a special case: the list.

Lists are *ordered* summaries of *arbitrary* objects. Therefore, an array can also be understood as a list because it is nothing more than an ordered summary of objects of the *same type*. Lists are created in Python with square brackets, as you can see in the following example:

```
>>> numbers = [1, 2, 3, 4, 5, 6, 7]
>>> first_names = ['Beverly', 'Thomas', 'Marc', 'Jimmy',
                  'Cathy']
```

Entering the list name into the console shows us the contents of the list. Square brackets around the elements of the list remind us that this is indeed a list:

21

```
>>> numbers
[1, 2, 3, 4, 5, 6, 7]

>>> first_names
['Beverly', 'Thomas', 'Marc', 'Jimmy', 'Cathy']
```

If you are not working in the console but in script mode, i.e. writing a whole program, then use the **print()** function for output, which can also process lists:

```
print(numbers)
print(first_names)
```

Writing the identifier of the list alone, for example **numbers** does not lead to an output in the program—unlike in the Python console—as we already saw in ► Sect. 21.1.

21.6.1.2 Selecting Individual Elements of a List

Individual elements of the list can now be accessed by indexing, again using square brackets:

```
>>> numbers[3]
4

>>> firstnames[2]
'Marc'
```

Indexing in Python starts at 0, so the element with index 1 is already the *second* element in the list. You can use *negative* indexes to select from the back. If we wanted to pick the second name from the back of the list, we could write:

```
>>> first_name[-2]
'Jimmy'
```

The last element of the list has the index -1 , not -0 as you might expect.

The colon operator can also be used to specify a range as an index. For example, if we wanted to select the second to fourth first names, we could do it as follows:

```
>>> first_names[2:5]
['Marc', 'Jimmy', 'Cathy']
```

The elements with the indices 2, 3 and 4, i.e., the third, fourth and fifth element of the list, are selected. Please note that the element with index 5, i.e., the sixth element, is not part of the selection. The right border of the index specification is—counter-intuitively and unlike in other languages, such as R—not part of the selected elements.

Just like the original list, the selection itself is again a list because we have selected several elements. If, on the other hand, we select only a single element, the selection is no longer a list, but has the type of the respective element of the list:

```
>>> type(first_names)
<class 'list'>

>>> sel = first_names[2:5]
>>> type(sel)
<class 'list'>

>>> sel = first_names[2]
>>> type(sel)
<class 'str'>
```

When indexing with a range of indexes, one side can also remain open. If the left side remains open, the system selects from the beginning of the list; if the right side remains open, the system selects to the end of the list:

```
>>> first_names = ['Beverly', 'Thomas', 'Marc', 'Jimmy',
                  'Cathy']
>>> first_names[:3]
['Beverly', 'Thomas', 'Marc']

>>> first_names[3:]
['Jimmy', 'Cathy']
```

You could also leave both range boundaries open (**first_names[:]**), then all elements of the list would simply be selected (ultimately creating a copy of the list). If you want to select several, non-contiguous elements of a list in one step, for example the first and third element, this is not quite so easy in Python. This is where the use of list-comprehension expressions comes in handy, which we will deal with later when we look at the implementation of loops in Python.

21.6.1.3 Edit Lists

You can use the indexing we just learned about not only to select items from a list for display or further processing, but also to modify the items in the list directly. In this section, we will look at how you can modify elements of a list, add elements to a list, delete elements from a list, sort a list, and join multiple lists into a new list.

■ Change List Elements

Let's start by changing the value of a list item:

```
>>> first_names = ['Beverly', 'Thomas', 'Marc', 'Jimmy',
                   'Cathy']
>>> first_names[1] = 'Jason'
>>> first_names
['Beverly', 'Jason', 'Marc', 'Jimmy', 'Cathy']
```

Even entire index ranges can be assigned like this:

```
>>> first_names = ['Beverly', 'Thomas', 'Marc', 'Jimmy',
                   'Cathy']
>>> first_names[3:5] = ['Joe', 'Aimee']
>>> first_names
['Beverly', 'Thomas', 'Marc', 'Joe', 'Aimee']
```

In this case, make sure that the value you assign is again a list (that is, it is enclosed in square brackets and is of the same the length of the sublist you want to replace).

? 21.3 [5 min]

What happens if the assigned object is not a list or does not have the length of the substituted sublist? Try it out and see if you can explain the results!

Lists, like all variables in Python, are objects, so they have corresponding attributes and methods.

■ Append Elements to Lists

The **append(object)** method can be used to add an object to the list. This directly changes the list instance for which the **append()** method is called:

```
>>> first_names = ['Beverly', 'Thomas', 'Marc', 'Jimmy',
                   'Cathy']
>>> first_names.append('Sandy')
>>> first_names
['Beverly', 'Thomas', 'Marc', 'Jimmy', 'Cathy', 'Sandy']
```

If you want to add an element *anywhere in* the list, use the method **insert(*insert_before_element_index*, *object*)** and pass it the position you want the new element to have and the element itself:

```
>>> first_names = ['Beverly', 'Thomas', 'Marc', 'Jimmy',
                   'Cathy']
>>> first_names.insert(2, 'Sandy')
>>> first_names
['Beverly', 'Thomas', 'Sandy', 'Marc', 'Jimmy', 'Cathy']
```

■ Delete Elements from Lists

You can delete items from a list with similar ease:

```
>>> first_names = ['Beverly', 'Thomas', 'Marc', 'Jimmy',
                   'Cathy']
>>> first_names.__delitem__(4)
>>> first_names
['Beverly', 'Thomas', 'Marc', 'Jimmy']

>>> del first_names[2]
['Beverly', 'Thomas', 'Jimmy']
```

To do this, you can either use the class method **__delitem__(*elementindex*)**, to which you only need to pass the index of the element to be deleted, or the operator **del**, which, because it is an operator, is used without parentheses! You call it together with the element to be deleted. You can even use the **del** operator to delete several elements at once, for example by calling **del first_names[2:4]**.

As you can see from the example of deletion, some operations can be implemented with both a class method and an operator. This is also true for the selections we looked at in the previous section. Ultimately, the selection **firstname[2]** is nothing more than the call **first_names.__getitem__(3)** of the **__getitem__(*elementindex*)** method, and that's exactly what Python processes internally when you index your list using the square brackets, which are themselves nothing more than an operator.

■ Sort Lists

You can easily sort your list by using the class methods **sort()** and **reverse()**, depending on whether you want to sort in ascending or descending order:

```
>>> first_names = ['Beverly', 'Thomas', 'Marc', 'Jimmy',
                  'Cathy']
>>> first_names.sort()
>>> first_names
['Beverly', 'Cathy', 'Jimmy', 'Marc', 'Thomas']

>>> first_names.reverse()
>>> first_names
['Thomas', 'Marc', 'Jimmy', 'Cathy', 'Beverly']
```

■ Determine the Length of a List

You can easily determine the length of a list using the `__len__()` method.

```
>>> first_name.__len__()
5
```

Unlike the previously discussed methods to manipulate the class **list**, `__len__()` does not modify the list, but only returns the length of the list.

■ Joining Lists

If you want to join two lists together, use the plus operator (+).

```
>>> first_names + numbers
['Beverly', 'Thomas', 'Marc', 'Jimmy', 'Cathy', 1, 2, 3, 4, 5,
6, 7]
```

Here you can see that the result is a list whose elements are partly strings, but also partly numbers. Lists, unlike the more specialized arrays found in many programming languages, can contain different kinds of elements. In particular, the elements of lists can themselves be lists. We will take a closer look at this situation in the following section.

21.6.1.4 Lists as Elements of Lists

Consider the following list:

```
>>> list_with_sublist = [1, 2, 3, ['a', 'b', 'c'], 4]
```

It contains as fourth element (i.e., as element with index 3) again a list. We can also see this very quickly if we select the element and inspect it more closely:

```
>>> list_with_sublist = [1, 2, 3, ['a', 'b', 'c'], 4]
>>> list_with_sublist[3]
['a', 'b', 'c']

>>> type(list_with_sublist[3])
<class 'list'>
```

To access an element of our “sub-list”, we first pick the fourth element with **list_with_sublist[3]**. This is now again a list. So, we should be able to select from this (sub-)list again, just as we did with the full list. For example, if we wanted to pick out the third element of the list, i.e., the **c**, we would index “twice” as follows:

```
>>> list_with_lists[3][2]
'c'
```

In this way, lists can also be used to construct multidimensional variable fields. Suppose we wanted to map a rectangular value scheme that looks like this:

1	2	3
4	5	6
7	8	9

This can be easily mapped with nested lists:

```
>>> three_by_three = [[1,2,3], [4,5,6], [7,8,9]]
>>> three_by_three
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

We can then access the coordinates row 2, column 1 (i.e., the number 4 in our value scheme) by double indexing our list (note that the indexing starts at 0!):

```
>>> three_by_three[1][0]
4
```

The first index is always the row index, the second the column index. Although multidimensional fields can also be represented very well with lists, at least if you are working with very large fields and the speed of the program is an important factor, you will want to use special data structures such as those contained in the *NumPy* module. *NumPy* is an important add-on library for anyone working in the data science area. The library also provides a special array data type that, while not

as flexible as lists (because it only takes elements of the same type), is more compact in memory and faster to access. For our purposes here, however, lists, which are part of the standard Python language, will suffice.

? 21.4 [20 min]

Try out working with lists a little! Create lists, select items from them, add new items, delete items. It is important that you get a feel for how to work with lists, because lists play a big role in practical work with Python.

21.6.1.5 Strings as Lists

A special feature regarding lists are strings. In Python, they have list-like properties. Their individual elements, the characters, can be read-accessed in list notation:

```
>>> message = 'Hello World'
>>> message[1]
'e'
```

Attempting to *edit* a string character in this way, such as with `message[1] = 'x'`, will result in the error message **'str' object does not support item assignment**.

21.6.2 Tuples

Tuples are a data type that is similar to lists in many ways. Just like lists, tuples are ordered collections of multiple objects that do not necessarily have to be of the same type. The main difference with lists is that tuples are *immutable*. Take a look at the following example, in which we create a tuple from three integers:

```
>>> number = (27, 9, 51)
>>> number
(27, 9, 51)

>>> type(number)
<class 'tuple'>

>>> number[1]
9

>>> type(number[1])
<class 'int'>

>>> number[1]=36
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Note that—unlike with lists—the elements with which the tuple is initialized are enclosed in *round* brackets. However, the elements are accessed in exactly the same way as in lists, namely by specifying the index of the element in question in *square* brackets; and here, too, indexing naturally starts at 0—as always in Python—so that **number[1]** queries the *second* element of the tuple. In the last step, we try to assign a value to this second element of the tuple. This fails because tuples are an immutable data type whose elements cannot be changed after initialization. Also, no new elements can be added. The tuple is and remains as it was created when it was created.

By the way, you can also omit the parentheses when creating the tuple. So, we could also create the tuple **number** above this way:

```
>>> number = 27, 9, 51
>>> number
(27, 9, 51)
```

You may have asked yourself why you should use tuples at all, when lists can do everything tuples can do but are mutable on top of that. The main advantage of tuples is that Python can process them faster than lists. They are also useful when you want to make sure that your data is protected from being overwritten. If you accidentally try to do this in your program, you will get an error message, as you saw in the example above. Even if you don't use tuples consciously very often, Python uses tuples in the background. For example, unlike many other languages, Python allows you to place multiple variable assignments in a single statement, like this:

```
>>> a, b = 5, 3
>>> a
5

>>> b
3
```

What happens internally here is that Python first creates a tuple **(5, 3)** and then assigns its elements to the two variables **a** and **b**. We will see later how in Python—and this is again not possible in many other programming languages—you can have a function/method return more than one value. “Under the hood” this work with tuples.

21.6.3 Dictionaries

Another complex data type besides lists and tuples is the dictionary. The term “dictionary” describes the functionality of these data structures quite well. Unlike lists, where we use the index of the relevant element in the list to select a value, a *key* is

used in dictionaries to achieve the same thing. The dictionaries are therefore *associative fields*. If you no longer remember the concept, it is best to turn back a few pages to ► Sect. 11.6.

As an example, let's create a dictionary that stores the age of a person (value) for each first name (key). The various key-value pairs are written in curly brackets, with the key and value each separated by a colon, and the pairs themselves separated by commas:

```
>>> d = {'Thomas': 30, 'Beverly': 19, 'Marc': 28}
>>> d
{'Thomas': 30, 'Beverly': 19, 'Marc': 28}
```

In our example, the keys are strings, the values are numbers. However, this does not have to be the case. Numbers can themselves be keys. Even tuples can be keys, but not lists, because keys must always be immutable (remember that tuples are immutable, but lists can be modified). All possible object types are suitable as values, including lists or dictionaries themselves. In this way, it is also possible to construct a nested dictionary. We will take a closer look at this in an exercise later on.

The individual elements are now accessed with the help of the key, as is usual with associative fields:

```
>>> d['Thomas']
30
```

Note that although the key-value pairs are written in curly brackets when the dictionary is created, square brackets are used to access individual elements of the field.

Unlike lists, dictionaries are *unordered* collections of elements. Accessing individual elements using a numeric index, which specifies the position of the element within the dictionary, is not possible because in an unordered data structure elements have no natural position at which they can be located and queried. Therefore, attempting to access an element using a numeric index will result in an error message:

```
>>> d[1]
Traceback (most recent call last):
File "<input>", line 1, in <module>
KeyError: 1
```

This **KeyError** tells us that a key **1** does not occur in the dictionary. Python interprets the **1** as a key, tries to find the corresponding value, but finds that **1** is not among the keys used in the dictionary. We would receive a similar error message if we used a name as a key that does not occur in the dictionary, for example with **d['Jacob']**.

Elements can easily be added to a dictionary by making an assignment for the new element, associating the new key with a value:

```
>>> d['Cathy'] = 36
>>> d
{'Thomas': 31, 'Beverly': 19, 'Marc': 28, 'Cathy': 36}
```

Of course, we can also use an existing key in such an assignment, as in the following example:

```
>>> d['Beverly'] = 22
>>> d
{'Thomas': 31, 'Beverly': 22, 'Marc': 28, 'Cathy': 36}
```

Since the key is ultimately the identifier we use to access an element of the dictionary, it must be unique. Therefore, we cannot simply add a new element that has a key which already exists in the dictionary. Instead we change the already existing element in this case.

Not only adding, but also deleting elements is very easy with the **del** operator already known from lists:

```
>>> del d['Thomas']
>>> d
{'Beverly': 22, 'Marc': 28, 'Cathy': 36}
```

Alternatively, as with lists, a call to the class method **__delitem__()** would be possible: **d.__delitem__('Thomas')**. Unlike the ordered lists, this method takes the corresponding key as argument instead of a numeric position index.

Sometimes you are interested in checking whether a certain key occurs in the dictionary. This can easily be done with the **in** operator. If you apply the **in** operator to a key and a dictionary, the result is a logical value that indicates whether the key is used in the dictionary or not.

```
>>> 'Cathy' in d
True

>>> 'Amy' in d
False
```

Of course, keys and values can also be extracted from the dictionary. For this purpose, the dictionary class has two special methods, **keys()** and **values()**. The return values of these methods are a bit more complicated, we will deal with this kind of object, the iterable objects, in more detail later. To work with them easily, we con-

vert them into a list using the **list()** method, which is the constructor method of the **list** class:

21

```
>>> list(d.keys())
['Beverly', 'Marc', 'Cathy']

>>> list(d.values())
[22, 28, 36]
```

With these lists, in turn, we can of course do anything that lists allow us to do, for example, access a particular element. Since the list of keys is of course an ordered data structure, we can access its elements with a numerical index. For example, the second element with index 1 (remember: indexing in Python starts at 0 for the first element):

```
>>> list(d.keys())[1]
'Marc'
```

The complete elements of the dictionary can also be extracted into a list:

```
>>> list(d.items())
[('Beverly', 22), ('Marc', 28), ('Cathy', 36)]
```

The elements of the list are now the individual elements of the dictionary. They are themselves tuples of key and value:

```
>>> type(list(d.items())[0])
<class 'tuple'>
```

? 21.5 [10 min]

Create a dictionary whose keys are numeric product numbers and whose values are dictionaries containing, for each product number, the description, manufacturer, and price of the product in question.

21.6.4 Sets

The last data type we want to look at is the *sets*. Sets and dictionaries have in common that both are *unordered* collections of objects. Similar to the keys of a dictionary, which must always be unique, each element in a set can only occur once. Sets support set operations as they are known from mathematical set theory, for example, the determination of the intersection or union of two sets.

To create a set, we use the curly braces—similar to the dictionary; this time, however, the curly braces do not contain key-value pairs, but simply the individual elements of the set:

```
>>> friends_thomas = {'Will', 'Beverly', 'Peter', 'Michael'}
>>> friends_julia = {'Peter', 'Will', 'Helen', 'Michael',
'Caroline'}
```

You can easily verify that the order of the elements does not matter by comparing two sets containing the same elements but in different order:

```
>>> {'Beverly', 'Peter'} == {'Peter', 'Beverly'}
True
```

In doing so, we use the double equal sign for comparison, which, we will see later, is Python's operator for comparisons for equality (a single equal sign would be considered by Python to be an attempt at assignment, which of course would not work here). The result of the comparison, **True**, confirms that the order of the elements in the sets does not matter; the two sets are identical despite different orders of their otherwise equal elements.

With the sets defined above, we could now check, for example, which elements occur in both sets, i.e., which people are friends of both Julia and Thomas:

```
>>> friends_thomas.intersection(friends_julia)
{'Will', 'Peter'}
```

To do this, we use the class method **intersection(*other_set*)** here, which the class **set** brings with it by default. In this case, the result, namely the intersection that is again returned to us as **set** (easily recognizable by the curly brackets), would of course be the same if instead of calling **friends_thomas.intersection(friends_julia)** we had called **friends_julia.intersection(friends_thomas)** the other way around.

Similarly, we could have worked with the intersection operator **&**, which is supported by the class **set**:

```
>>> friends_thomas & friends_julia
{'Will', 'Peter'}
```

The fact that the intersection operator is an ampersand is no coincidence, since we are looking for all elements in the intersection that are contained in one *and* the other set.

Union sets, i.e., the set of all elements contained in one or both initial sets, can be determined with the class method **union(*other_set*)** or the pipe operator **|**:

21

```
>>> friends_thomas.union(friends_julia)
{'Helen', 'Caroline', 'Michael', 'Will', 'Peter', 'Beverly'}

>>> friends_thomas | friends_julia
{'Helen', 'Caroline', 'Michael', 'Will', 'Peter', 'Beverly'}
```

As you can see, the union of our two sets contains the names **Will** and **Peter** only once, although they appear in both **friends_thomas** and **friends_julia**. But this is precisely the nature of sets (as well as sets in mathematics): All elements of a set are different from each other, no element can occur more than once.

Similarly, we can check whether one set is a *subset* of another:

```
>>> friends_thomas.issubset(friends_julia)
False

>>> {'Michael', 'Beverly'}.issubset(friends_thomas)
True
```

Again, a convenient shortcut is available in the form of an operator, namely the less-than-equal operator:

```
>>> {'Michael', 'Beverly'} <= friends_thomas
True
```

? 21.6 [20 min]

Find out (for example, with the **help()** function) what other interesting operations are possible with sets and try them out. Use the example sets **friends_thomas** and **friends_julia** from this section or create your own sets.

21.7 Self-Defined Classes

21.7.1 Defining and Using Classes

All the data types we looked at were classes, whether they were basic types like **int** and **str**, or more complex types like lists or dictionaries. Because Python as a programming language follows the object-oriented paradigm, we can of course define classes ourselves.

Remember the example of the **Product** class from ► Sect. 11.7.2, which captures all the important basic information about a product? These properties were the name, a more detailed description, the part number, the manufacturer's name, and the price. We can build such a class very easily in Python, using the **class** keyword:

```
class Product:
    name = ''
    description = ''
    item_number = ''
    manufacturer = ''
    price = 0.0
```

After the colon, the code block begins (attention, indentation!) with the attributes of the classes, to each of which we assign an initial value. That's about it! Now we can use our class **Product** and create a variable of this type:

```
gardenshovel = Product()
```

Product() is the constructor method of our class, which we use here just as we did for Python's basic data types in ► Sect. 21.4.2. While we haven't defined a constructor of our own at all, our class gets a default constructor from Python that does nothing but create an object of the class. Later, when we look at methods/functions in a bit more detail, we'll look at how we can write our own constructor and use it to allow the user of our class to, for example, set the values of certain attributes to their own specifications right when a new class instance is created. This is exactly what we did in ► Sect. 21.4.2, when we called the constructor of the class **int** with **int(56)** and caused it to create a new **int** object containing the integer value 56.

Now that we have created the object of type **Product**, we can customize its properties as we wish:

```
gardenshovel.name = 'Garden shovel, stainless steel'.
gardenshovel.price = 10.99
```

If you work with *PyCharm* and type **gardenshovel**. (including the dot operator!), the dropdown menu that opens will list, among other things, the attributes we just defined that the object **gardenshovel**, as an instance of the **Product** class, has.

You can easily verify that these assignments worked by displaying the values of the attributes:


```
print(gardenshovel.name)
print(gardenshovel.price)
```

Note that we are no longer in Python's interactive mode here (you can easily tell by the missing input prompt `>>>` before the statements), although we could of course have fed the class definition into the Python console. Therefore, to display the contents of a variable, simply typing its identifier is no longer sufficient (in fact, it has no effect at all). Instead, we must explicitly cause the output by calling the `print()` function.

21.7.2 Deriving Classes from Other Classes

In ► Sect. 11.7.3 we learned about the concept of inheritance, which of course Python also offers as an object-oriented language. There, we defined the class **Book** as a derived class of the class **Product**, which has two special attributes, **author** and **pages**, that the base/parent class **Product** does not have by default. In Python, to derive one class from another, the base class name is placed in parentheses after the derived class name in the class definition:

```
class Book(Product):
    pages = 0
    author = ''
```

Now we can create an instance of the **Book** class by calling its default constructor, which Python kindly provides:

```
grisham1992 = Book()
```

If you now display the attributes of the new object in *PyCharm* by typing **grisham1992**, you will immediately see that the instance of the class **Book** has not only its own attributes, namely **pages** and **author**, but also the properties inherited from **Product**, such as **description** and **price**. *PyCharm* also shows you, as you can see in ■ Fig. 21.3, from which class the respective attribute originates.

We can now work with all properties as we wish:

```
grisham1992.name = 'The Pelican Brief'
grisham1992.price = 8.99
grisham1992.author = 'John Grisham'
grisham1992.pages = 478
```



■ Fig. 21.3 Own and inherited properties of the “Book” class

Classes in Python can be derived not only from one base class, but also from multiple base classes. For example, there could be another class **Copyright**:

```

class Copyright:
    owner = ''
    year = 1900

```

Then we could derive our class **Book** from both base classes, **Product** and **Copyright** at the same time:

```

class Book(Product, Copyright):
    pages = 0
    author = ''

```

By specifying both base classes in the definition of our class **Book**, we create a class that inherits the attributes and methods of both classes. Accordingly, we can now work with the attributes that the **Copyright** class brings:

```

grisham1992 = Book()
grisham1992.owner = 'Double'
grisham1992.year = 1992

```

? 21.7 [10 min]

Define a **Customer** class with useful customer attributes and derive a **BusinessCustomer** class from this class that contains additional attributes that are only relevant for business customers.

21.7.3 Avoid Ambiguity: Name Mangling

But what if not only the class **Product**, but also the class **Copyright** had an attribute called **name**? The derived class **Book** would nevertheless have only *one* attribute **name**. But from which “parent” would this attribute now originate? Is it the name from **Product** or the name from **Copyright**? The answer in this case would be: that of **Product**, because Python proceeds from left to right, i.e. the class from which it is derived “first” is searched first for the attribute name. Python only searches in the other parent classes if the first parent class did not have an attribute of this identifier.

To eliminate name confusion altogether, there is an option to prefix class attributes with a double underscore. Then the class definition of our **Copyright** class would look like this:

```
class Copyright:
    owner = ''
    __name = ''
    year = 1900
```

The effect of the double underscore is that Python automatically makes the property accessible under the identifier `__class__attribute`, which in our example is `__Copyright__name`:

```
grisham1992._Copyright__name = 'Copyright'
print(grisham1992._Copyright__name)
```

This process, also known as *name mangling*, allows us to avoid ambiguities and misunderstandings when accessing attributes whose names may occur more than once in the class hierarchy.

21.8 Summary

In this chapter, we learned about variables in Python and how to work with them. We also learned how to define classes and create objects as instances of classes.

You should take away the following points from this chapter:

- Python knows simple data types, most notably **int** (integers), **float** (floating point numbers), **str** (strings), and **bool** (logical/truth values), as well as more complex data types, most notably **list** (an ordered collection of different

objects), **dictionary** (an unordered associative field), **tuple** (an immutable collection of objects), and **set** (an unordered collection of unique objects).

- All data types are classes, variables of these data types are object instances; they have attributes (or properties) and methods to manipulate and otherwise work with the objects.
- Variable names are case-sensitive, as is the case everywhere in Python; the official recommendation is to use lowercase variables, and to separate multiple terms in variable names with an underscore.
- Variables do not have to be declared.
- Variables can be generated either by assigning an object to a variable name (where Python automatically determines the type) or by using the constructor method of the respective data type class.
- Constructors can often be called with objects of other types as arguments; this allows explicit conversion between data types.
- Python implicitly converts relatively little, but does between **int** and **float**, where necessary.
- Strings can be enclosed in single and double quotes.
- To display the contents of a variable, its name can be typed into the Python console; within a Python program, however, the output must always be explicitly brought up (primarily using the **print()** function).
- The complex data types in Python differ in whether they are mutable or not (mutable: **list**, **dictionary**, **set**; immutable: **tuple**) and whether the elements in them are stored in an ordered or unordered fashion (ordered: **list**, **tuple**; unordered: **dictionary**, **set**).
- The elements of complex data types (and in the case of dictionaries, both keys and values) may be of different types and may themselves even be objects of that or another complex data type; for example, there may be a list that in turn contains lists as elements, or a dictionary whose keys are partly tuples and whose values are lists and other dictionaries.
- The elements of ordered data types (lists, tuples) can be addressed via numeric indices (that is, element numbers); the first element always has the index 0.
- The colon operator can be used to address an index range, where **A:B** means all elements between the range boundaries **A** (inclusive) and **B-1** (inclusive).
- Range boundaries can also be left open, which is equivalent to “from the beginning” (left boundary not specified) or to the end (right boundary not specified).
- Negative indexes mean: Indexing from the back instead of the front.
- Python does not know a special data type for arrays/fields, but the data type **list**, which holds arbitrary elements in ordered form; an array is thus a special case of a list (namely one whose elements are all of the same type).
- Strings also behave like lists in read access, in that their individual characters can be addressed in the notation usual for lists; however, write access to the characters is not possible in this way.
- You can define your own classes using the keyword **class**; classes can be derived from one or more “parent classes”, i.e., inherit their attributes and methods.
- A double underscore of an attribute or method in a class definition means that the attribute/method is also accessible under the name ***_class_attribute*** or

`__class__` `method()` (*name mangling*); in this way, ambiguities in attribute/method identifiers caused by multiple inheritance can be avoided.

21

The following table gives an overview of the most important data types. For the simple data types, you can see the creation by assignment and by calling the constructor method of the respective class. Of course, the complex data types **list**, **tuple**, **dictionary** and **set** can also be created by constructor, but this is not shown here for the sake of clarity.

Overview of the most important data types			
Data type	Saves	Creation of a variable	Access to elements
int	Integers	Assignment: <code>x = 5</code> Constructor: <code>x = int('5')</code>	—
float	Floating point values	Assignment: <code>x = 0.5</code> Constructor: <code>x = float('5')</code>	—
str	Strings	Assignment: <code>x = 'Hello World'</code> Constructor: <code>x = str(0.5)</code>	—
bool- ean	Logical/truth values	Assignment: <code>x = False</code> Constructor: <code>x = bool('True')</code>	—
list	Ordered, changeable compilation of other objects (possibly also of different types)	Elements comma-separated in square brackets: <code>x = ['Marc', 'Helen', 'Toby', 'Thomas']</code>	Index or index range in square brackets: <code>x[1]</code> (gives: 'Helen') <code>x[2:3]</code> (gives: ['Hellen', 'Thomas'])
tuple	Ordered, unchangeable compilation of other objects (possibly of different types)	Elements comma separated in round brackets: <code>x = ('abc', 27.5)</code>	Index in round brackets: <code>x(1)</code> (gives: 27.5)
dic- tion- ary	Unordered set of key-value pairs, keys and values can be objects of different types; keys must be unique	Key-value pairs comma-separated in curly brackets, key and value each separated by colon: <code>x = {'Marc' : 27, 'Helen' : 41}</code>	Key in square brackets: <code>x = ['Marc']</code> (provides: 27)

Data type	Saves	Creation of a variable	Access to elements
set	Unordered collection of objects (possibly of different types) that must be unique (an object can only appear once in the list)	Elements comma-separated in curly brackets: <code>x = {'Marc', 'Helen'}</code>	Since there is no natural order and no key for access, selection of a single element makes no sense (you would already have to know the element to access it)

21.9 Solutions to the Exercises

■ Exercise 21.1

Some examples of other methods of the **str** object are:

- **find(sub)**: Searches for **sub** in the string of the **str** object and returns the index of the first character of the found substring, or **-1** if the substring was not found. As always, this is case sensitive. Usage examples:

```
>>> x = 'Hello world!'
>>> x.find('wo')
6

>>> x.find('WO')
-1

>>> x.upper().find('WO')
6
```

In the last example, the string is first converted to upper case. **x.upper()** returns the string converted to upper case. This is, of course, again a **str** object. This is then searched using its **find()** method. This time it returns a hit.

- Further arguments can be used to restrict the range in which the string is to be searched; the specification **x.find(sub[, start[, end]]) -> int** in the help means that since **start** and **end** are optional arguments of **find()**, they can but do not have to be specified. That's why they are each enclosed in square brackets. Note the subtleties of the bracket order: the square brackets around the **end** argument are contained within the square brackets enclosing the **start** argument. This means that **end** can only be specified if **start** has been specified, but **start** can be used without specifying **end**!
- After the **->** you find the type of the return value of the function, in our case **int**, because the index is returned as a number.

- **capitalize()**: Capitalizes the string (first word only!) and converts all other characters to lowercase. Usage example:

```
>>> x = 'hello world'
>>> x.capitalize()
'Hello world'
```

- **is.lower(), is.upper()**: Check whether the string is all lowercase or all uppercase. Usage example:

```
>>> x.isupper()
False

>>> x = 'HELLO WORLD'
>>> x.isupper()
True
```

- **center(width, [fillchar])**: Creates a string of length **width**, centred by the string of the **str** object. To the left and right of it, it is “padded” with the padding character (by default, if not specified: space). Usage example:

```
>>> x = 'Hello World'
>>> x.center(50, '*')
'*****Hello World*****'
```

■ Exercise 21.2

A program that reads the user’s age in years and outputs it as minutes might look like this: `age = input('Please enter your age in years:')`

```
minutes = int(age) * 365 * 24 * 60
print('With', age, 'you have already experienced at least',
minutes)
```

■ Exercise 21.3

Let’s first assign an object to the list elements with indices 3 and 4. We start with an integer number:

```
>>> first_names = ['Beverly', 'Thomas', 'Marc', 'Jimmy',
'Cathy']
>>> first_names [3:4] = 23
Traceback (most recent call last):
File "<input>", line 1, in <module>
```

```
TypeError: can only assign an iterable
```

We get an error message. Generally, Python cannot assign something which is itself not a list to a list/selection from a list (unless the selection contains only one element). Contrary to what one might expect, Python does not simply replace the elements 3 and 4, i.e. **'Jimmy'** and **'Cathy'** (indexing starts at 0!), with the number 23. The picture changes when we “wrap” the number 23 in a list:

```
>>> first_names = ['Beverly', 'Thomas', 'Marc', 'Jimmy',
                   'Cathy']
>>> first_names[3:4] = [23]
>>> first_names
['Beverly', 'Thomas', 'Marc', 23]
```

Now the two selected elements are actually replaced by the list containing only the number 23. But since this is shorter than the replaced partial list, our list **first_names** is *shortened* accordingly.

Now let's try something else. This time we replace the part list with a string:

```
>>> first_names = ['Beverly', 'Thomas', 'Marc', 'Jimmy',
                   'Cathy']
>>> first_names[3:4] = 'Amy'
>>> first_names
['Beverly', 'Thomas', 'Marc', 'A', 'm', 'y']
```

Unlike the assignment **first_names[3:4] = 23** above, this time we don't get an error message. But something seemingly strange happens: The partial list **first_names[3:4]** is replaced by the letters of the name Amy, with each letter becoming a new list element. The reason is that strings can also be interpreted as lists. Therefore, the assignment **first_names[3:4] = 'Amy'** is ultimately a replacement by a list, namely by the list **['A', 'm', 'y']**.

■ Exercise 21.4

No solution.

■ Exercise 21.5

Here we are now dealing with a nested dictionary. For the sake of clarity, you can see the dictionary definition below with line breaks (you remember from ► Sect. 20.1.2 that a statement can be wrapped inside curly braces):

```
>>> d= { 12345:
...     {
...         'description': 'Plastic Garden Chair "Garden Friend"',
...         'manufacturer': 'Garden Paradise Inc.',
```



```

...     'price': 10.99
... },
... 56789:
... {
...     'description': 'Garden shovel, stainless steel',
...     'manufacturer': 'Big G Gardening Tools Inc.',
...     'price': 49.90
... }
... }

```

Access is then via two keys:

```

>>> d[12345]['price']
10.99

```

The expression **d[12345]** returns a dictionary, and a value is selected from this dictionary using a key that exists in this dictionary.

■ Exercise 21.6

Some examples of other set operations:

- **difference(*other_set*)**: Returns the elements that exist in **other_set** but not in the set whose **difference()** method was called. Usage example:

```

>>> friends_thomas = {'Will', 'Beverly', 'Peter', 'Michael'}
>>> friends_julia = {'Peter', 'Will', 'Helen', 'Michael',
...                 'Caroline'}
>>> friends_julia.difference(friends_thomas)
{'Caroline', 'Helen'}

```

- **remove(*element*)**: Deletes an element from a set. Usage example:

```

>>> friends_julia.remove('Helen')
>>> friends_julia
{'Peter', 'Will', 'Michael', 'Caroline'}

```

- **isdisjoint(*other_set*)**: Checks if the set whose **isdisjoint()** method is called and the other set are disjoint, that is, have no elements in common. Usage example:

```

>>> friends_thomas = {'Will', 'Beverly', 'Peter', 'Michael'}
>>> friends_julia = {'Peter', 'Will', 'Helen', 'Michael'}
>>> friends_thomas.isdisjoint(friends_julia)
False

```

■ Exercise 21.7

The two classes **Customer** and **BusinessCustomer** could look like this:

```
class Client:
    firstname = ''
    lastname = ''
    street = ''
    city = ''
    zip = ''
    email = ''

class BusinessCustomer(Customer):
    company = ''
    payment_terms = 14
    taxid = ''
```



User Interfaces: How Do I Input and Output Data?

Contents

- 22.1 Input and Output in the Console – 278**
- 22.2 Graphical User Interfaces with Tkinter – 280**
 - 22.2.1 Overview – 280
 - 22.2.2 Hello Tkinter! – 281
 - 22.2.3 Graphical Controls (Widgets) – 282
 - 22.2.4 Arrangement of the Controls (Geometry Managers) – 299
 - 22.2.5 Events – 305
 - 22.2.6 Example: Calculator Application – 308
- 22.3 Working with Files – 315**
- 22.4 Exercise: Development of a Simple Text Editor – 317**
- 22.5 Summary – 318**
- 22.6 Solutions to the Exercises – 319**

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1007/978-3-658-42912-6_22.

Overview

We have dealt extensively with the organization of data in the program, namely the variables and objects used. Now is the time to talk about how data can be received from and output back to the user. To this end, in this chapter we will first deal with the simplest way of input and output, namely via the *console*. After that, we'll give our programs a much more appealing look with *graphical user interfaces* (GUIs). Not only will we take a closer look at working with GUIs using a complete application example, but you will also have the opportunity to program your own first GUI application as part of an exercise. Finally, we turn to working with *files*, which is of course extremely important in practice.

In this chapter you will learn:

- How to output information to the console and how to query the user in the console
- how to use the Python library **tkinter** to provide your program with a graphical user interface
- what controls are available to you, how they can be configured, placed and arranged on the interface
- how you can react in your program code to events that the user triggers via the interface (for example, when clicking on a button)
- how to read data from and write data to files.

22.1 Input and Output in the Console

Already in the previous chapters we have seen the two most important functions that are used to input and output data in the Python console, **input()** and **print()**.

■ Input

input(prompt) displays a prompt and lets the user enter input from the keyboard, completing the input by pressing the <RETURN> or <ENTER> key. **input()** then returns the input as a return value, and always as a string. This is important, particularly if you expect the input to be numbers that you can use in calculations later. In this case, you must first explicitly convert the return value of **input()** to a number, as we did in ► Sect. 21.5.

■ Output

The most important tool for outputting information is the **print()** function. It can be used to print one or more objects. If more than one object is to be printed, the optional string argument **sep** determines how the individual objects are separated from each other in the output; by default, this is done with a space character. The optional **end** argument controls what is printed at the end of the output; unless otherwise specified with **end**, a line break is placed at the end of the output. The line break is represented by an escape sequence, **\n** (for *new line*), which we already learned about in ► Sect. 11.2.2 in connection with strings. Of course, we can also

use these escape sequences directly in strings that we want to output: Consider the following small program as an example:

```
user = input('Username: ')
pwd = input('Password: ')
print('Welcome,' user, '!\nYour password is:', pwd)
```

Here we read a username and password from the user and then output a total of four objects:

- The string **'Welcome, '**
- The string variable **user**
- The string **'!\n Your password is:'** (Attention: This string contains a line break after the exclamation mark!)
- The string variable **pwd**.

If you call the program now and enter **peter** and **889X!z5** as user name and password, you get the following output:

```
Welcome, peter !
Your password is: 889X!z5
```

The space between the username **peter** and the exclamation mark is a bit unattractive. It is due to the default value of the separator argument **sep**, which is a space. Because of this, the two objects to be output, the variable **user** and the string beginning with the exclamation mark, are separated by a space. To avoid such problems, it is recommended to control the output of whitespace itself and set the argument **sep** to “empty string”, so that no separator is output by the **print()** function itself. The call to **print()** could then be like this:

```
print('Welcome,' user, '!\nYour password is: ', pwd, sep = '')
```

If you compare this with the call above, you will see that we have inserted a space wherever a space is to be output and have provided an empty string as **sep** argument. Please note that the **sep** argument must always be called with its name, otherwise the **print()** function does not know whether the last string still belongs to the objects to be output or whether it has a special meaning, i.e. it already represents the *next argument* of the function, as in our case. So, we call **sep** as a *keyword argument*. More on this in ► Sect. 23.1.2.

Of course, the objects that are printed with **print()** do not have to be strings only. In fact, you can use **print()** to print practically any object, and even include objects of different types, i.e., different classes, in one and the same **print()** call. This applies to classes that you have defined yourself. But how can this work? How does **print()** know how to display an object of type **Product** from ► Sect. 21.7, for example? The answer is simple: Classes in Python can have a special function —

`str__()`. It returns a string representation of the object and is called by `print()` when an object of that class is to be output. As the developer of the class, you can specify how objects of your class should be represented. All you have to do is define a `__str__()` method. We will look at an example of this in ► Sect. 23.2.

22

? 22.1 [5 min]

Specify three different ways to output the three string expressions 'First line', 'Second line' and 'Third line' in three consecutive lines.

22.2 Graphical User Interfaces with Tkinter

22.2.1 Overview

Programs on the command line or in the Python console are not everyone's cup of tea. Especially if you develop software for non-tech-savvy end users, you simply can't get around graphical user interfaces. Therefore, in this section we will look at how to design graphical user interfaces in Python with manageable effort, and how to back them up with program functionality. There are a number of ways to design graphical user interfaces in Python. With the *Streamlit* library, for example, you can use Python to create dynamic web pages that can also be hosted on web servers. In this chapter, we will concentrate on applications that run locally on a client, e.g. your computer.

Using the example of a graphical calculator, you will see that with only a few lines of code you can write a useful, fully functional program with an attractive interface that does not force the user to sit in front of a black console and stubbornly follow the given program flow. The chapter concludes with an exercise in which you will develop your own simple text editor that allows you to open, edit, and save text files.

There are many different libraries and frameworks for developing graphical interfaces. Many of them are cross-platform, which means that the programs you develop with them run on different computer (and sometimes mobile) operating systems. A commonly used library that we will work with is **tkinter**. Conveniently, it comes standard with Python, so we don't need to install anything extra. **tkinter** is based on **Tk**, a cross-platform library for graphical user interfaces that was originally developed in the early 1990s in a programming language called *Tcl*. *Tcl* has become popular mainly because of the **Tk** library. This is because it is not only available for *Tcl* itself, but also for a large number of other programming languages, including Python.

Python provides a package called **tkinter**, which is ultimately a kind of “connection” to **Tk**. To use the **Tk** library, Python calls a *Tcl* interpreter, which is also part of the standard Python installation. So, in effect, you are indirectly working with another programming language, but you don't need to understand *Tcl* syntax or call the *Tcl* interpreter yourself to do so. Instead, you can work in Python and use the usual Python syntax. Where necessary, Python then “translates” your statements into *Tcl* code and calls the *Tcl* interpreter. The practical thing about **Tk** is that once you understand how this library works, you can quickly develop your own graphical user interfaces in other programming languages that support **Tk**—and there are quite a few of them—without any major conversion difficulties.

In the next section, we will write a simple *Hello World* program that opens a window on the screen. After that, we will take a closer look at the various graphical controls of the user interface, the *widgets*, such as buttons, input fields and checkboxes. After that, we will only be missing two more components. We need to arrange these elements on the interface the way we want them to be in order to get the desired interface appearance. And finally, we must wire the controls to the program code behind them, so that they react appropriately to the user's actions. This gives us everything we need to write Python programs with a proper graphical interface, such as the calculator we will develop at the end of our introduction to **tkinter**. But let's start small first!

22.2.2 Hello Tkinter!

Create a new Python file with the following program code and run the program:

```
from tkinter import *
win = Tk()

win.title('Hello World Program')
win.geometry('900x500')

win.mainloop()
```

A window opens that looks something like  Fig. 22.1.

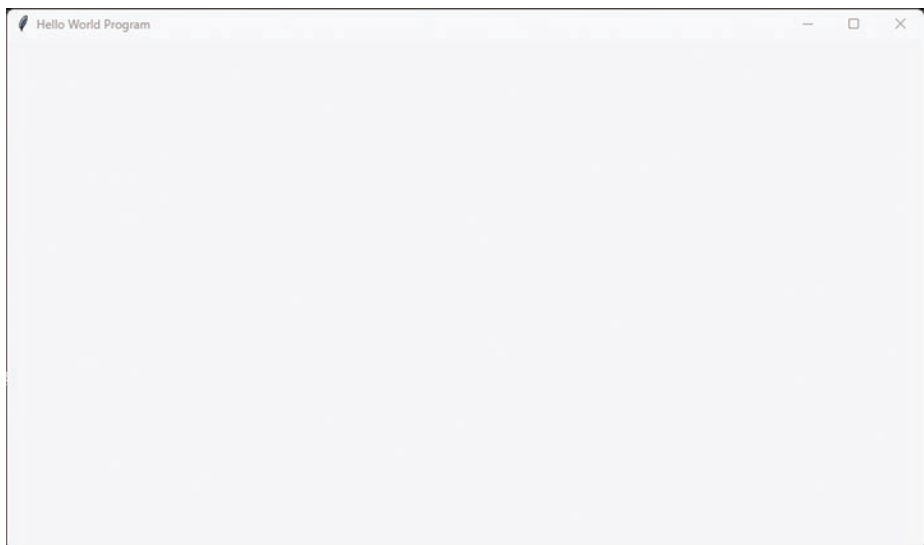


 Fig. 22.1 A *Hello World* program with **tkinter**

The window is still very empty, but that will change quickly in the following sections.

If you take a closer look at the code, you will notice that it starts with an **import** statement. This is necessary to make the module **tkinter** available. You should not worry about the exact structure of the **import** statement at this point, because we will deal with the import from modules in more detail in ► Sect. 23.3. At this point, it is sufficient to know that the **import** statement makes the classes of the module **tkinter**, in particular the class **Tk**, usable for our program.

We create an instance of this class **Tk** in our program, namely the object **win**, the main window of our application. With the methods **title(titletext)** and **geometry(dimensions)** we set two important properties, the title of the window and its size in pixels. Then, using the **mainloop()** method, we display the window on the screen and start event processing; our program can now react to user actions. Our first **tkinter** program is complete! Simple, isn't it? However, the user can still do little with our graphical interface. So, we need control elements that allow the user to make inputs and trigger actions. These controls are called *widgets* in **Tk/tkinter**. We will deal with them in the next section.

22.2.3 Graphical Controls (Widgets)

In this section we will look at a number of important widgets. Using the first widget, the button, as an example, you will see how widgets are created and how their properties are adjusted when they are created (or even later).

22.2.3.1 Buttons (Class Button)

Buttons are built into **tkinter** as the class **Button**. They are used to let the user of the program trigger actions. Like all widgets, they can be easily created using their constructor method. The program from the previous section extended by this constructor call then looks like this:

```
from tkinter import *
win = Tk()

win.title('Hello World Program')
win.geometry('900x500')
mybutton = Button(win, text = 'Press me')
mybutton.pack()

win.mainloop()
```

The **Button()** constructor method receives as function arguments the window in which we want to place the button, in our case **win**. Options for the design of the button can be specified; in our example, we label the button “Press me” with the help of the option **text**. Of course, we can still change the button's properties *after*

we have created the button. However, the properties are not simply editable class attributes of the **Button** class, but are changed with a special method called **config()**. Like the constructor, the **config()** method expects key-value pairs, consisting of the name of the option to be modified as a key and the value to be assigned to it. So, for example, if we wanted to change the width of our button, we would have to adjust the **width** option as follows:

```
mybutton.config(width = 50)
```

The same can be done with the option **text**, i.e., the label of the button.

A second way to modify options is to access them like a dictionary, whose keys are the option names:

```
mybutton['width'] = 50
```

The question now is what setting options are available? To find out, you should take a look at the help of the class **tkinter**. To do this, you must first import the **Button** class (or just all classes, as we do below) from the **tkinter** module into the console (remember: your Python programs and the console do not use the same namespace; the fact that you already have a Python program that imports the module does not mean that you can also use the module in the console. So, before calling the help function, you still have to execute an **import** statement in the console):

```
>>> from tkinter import *
>>> help(Button)
```

Right at the top of the help text that appears is the following 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
```

So, as you can see, on the one hand there are *standard options* common to most widgets, on the other hand there are special options available only for buttons. Unfortunately, the help lacks a description of what setting each option controls, and how these options are used. Also, the “official” documentation of the **tkinter** package (► <https://docs.python.org/3/library/tk.html>) is of rather limited use, especially for beginners. However, there are numerous pages on the Internet where the features are explained in an understandable way, currently for example ► https://www.tutorialspoint.com/python/tk_button.htm.

■ Table 22.1 shows an overview of some of the options that most, if not all, widgets have. This table also explains procedures for encoding colors and working with font formatting that can be used in many places in **tkinter**.

■ Table 22.2 then lists some of the *special* button options. In the following sections on the other widgets, you will find such a table with the most important specific properties for exactly this widget.

We have just seen options that partly behave like a dictionary. Therefore, you can also use the **keys()** method to read the keys and thus the names of the options. If, after creating the widget instance, you include in your program code the statement

```
print(mybuttons.keys())
```

you will see the names of the available options in the (run) console when you run the program. If you call the **config()** method with no arguments, you will get back the entire dictionary with all name-value pairs. You can also have this output:

```
options = switch.config()
print(options)
```

Often, the lowercase string option values such as '**sunken**' for the **relief** option can also be controlled with the help of a predefined, then capitalized constant, in our example **SUNKEN**. In the following, however, we will regularly work with the strings instead of the constants. If you want to work with the constants, look for the file **constants.py** in the **tkinter** directory of your hard disk. The constants are defined there. To use them, add the following import statement to your program:

```
from tkinter.constants import *
```

So far, we haven't even talked about the call to the **pack()** method that we snuck into our program. This statement makes the button visible on the screen in the first place (comment out the line and see what happens when you run the program again!). Making it visible is closely related to the arrangement of the widgets on the graphical interface. We will look at this in more detail in ► Sect. 22.2.4. At this point, all we need to do is call **pack()** to display our control on the interface.

■ Table 22.1 Standard properties of **tkinter** widgets

Option	Type	Meaning
active-back-ground/active-fore-ground	Str	Color of the control (background) or the text on it (foreground) when the control is activated (for in our case, by clicking on the button) <i>Like all colors in tkinter</i> , you can either specify one of the many predefined color constants (for example 'green' or 'purple'); you can quickly find lists of these color constants on the internet), or a red-green-blue (RGB) coded value of the form '#RRGGBB' , where RR , GG , and BB represent the hexadecimal (!) coded red, green, and blue parts of the color, respectively. It is best to use one of the many converters on the internet to convert the decimal values to the hexadecimal system. The value for red (R = 255, G = 0, B = 0) would thus become '#FF0000' , because FF represents the number 255 in the hexadecimal number system
back-ground/fore-ground	Str	Default color of the control (background) or the text on it (foreground)
border	Int	Thickness of the border of the control in pixels (border=0 means no border)
cursor	Str	The shape of the mouse cursor when the mouse pointer is over the control; examples are 'hand2' (hand), 'watch' (hourglass), 'cross' (cross), 'left_ptr' ("normal" mouse pointer with top left tip). There are also lists on the internet that specify the possible pointer characteristics
font	Font	The font formatting of the control; if you want to deviate from the default font, you must add an additional import statement and then create a new Font object using the Font() constructor: from tkinter.font import * font = Font(family = 'Times', size = 36, weight = 'bold', underline = 1) After that, the new Font object font can be assigned to the font option of the control: mybutton['font'] = font The family is a font identifier (e.g. Helvetica , Courier). The weight distinguishes between 'bold' and 'normal' . In addition, you can use the slant option, which is not used in the above example, to set the text to italic or not italic ('normal'). overstrike , which can take the values 1 and 0 (or True and False) just like underline , is used to strike through the text You can make changes to your Font object using the config() method, just as you can with controls: font.config(weight = 'normal') Changes you make in this way automatically affect <i>all</i> controls to whose font option you originally assigned the now changed Font object
padx, pady	int	Indentation of the text (or an image) on the control left/right (padx) or top/bottom (pady)
relief	str	3D representation of the control. Possible values here are: 'raised' (protruding, the default value), 'sunken' (deepened), 'flat' , 'groove' (deepened border) and 'ridge' (simple border, otherwise flat)
text	str	Labeling of the control

 **Table 22.2** Special properties of the button widget

Option	Type	Meaning
com- mand	func- tion	Function that is executed when the user clicks on the button. We will take a closer look at this event handling in ► Sect. 22.2.5
default	int	If default = 1 , then the button is the default button (triggered when the user presses the <ENTER> key)
height/ width	int	Height/width of the button. Specified in letter heights if text is displayed on the button, in pixels if an image is displayed. If not specified at all, width and height are calculated automatically
state	str	Button can be set to either ' normal ' (clickable) or ' disabled ' (grayed out). If the button is currently clicked, state takes the value ' active '

22.2.3.2 Menus (Class Menu)

Menus can be designed very easily with **tkinter**. To do this, we first create a new menu bar for our window, i.e., the **Tk** object **win** in the example above. The menu bar is represented in **tkinter** by the class **Menu**. The window to which the menu bar is to belong is already passed as an argument to the call of the constructor of this class. Conversely, we explicitly assign the new menu bar to the window with the help of the menu option of the **Tk** object, so that it is visible later:

```
menubar_top = Menu(win)
win.config(menu = menubar_top)
```

As you already know, instead of calling the **Tk** object's **config()** method in the last statement, we could have taken advantage of the practical fact that **tkinter** widgets work partly like dictionaries, and their options can therefore be accessed as well, like the key-value pairs of a dictionary. Accordingly, we could have written:

```
win['menu'] = menubar_top
```

Now, your application has a menu bar, but it is not yet displayed when you start the program in this state. First, we have to add the individual pull-down menus, which will then be expandable and collapsible for the user. These pull-down menus are also objects of the **Menu** class, which we can create by calling the class constructor. This time, however, we don't attach the new (pull-down) menus directly to the window **win**, but to our existing menu bar **menubar_top**:

```
file_menu = Menu(menubar_top, tearoff = 0)
```

The option **tearoff=0** causes the menu to be “bolted” to the window. If you omit this option or set it to **tearoff=1**, the first entry in your menu will be a dashed line. If you click on this entry, the menu detaches itself from its anchoring and is displayed in its own window, which can be moved around the screen at will (try it out!); a behavior that you would normally want to disable.

Now we can start adding new command items to the menu with **add_command(label = label)**:

```
file_menu.add_command(label = 'Open...')
file_menu.add_command(label = 'Save')
file_menu.add_separator()
file_menu.add_command(label = 'Close')
```

Using the **add_separator()** function of the **Menu** object we create a horizontal separator in the menu to structure our menu items more clearly. In addition to **add_command()** and **add_separator()**, there are two other types of menu items, **add_checkbutton()** and **add_radiobutton()**. Both allow the user to make a setting, as in the following example. The current setting (whether the menu option is currently selected or not) can be read at any time from the variable passed as the option **variable** to the **add_checkbutton()** function.

```
file_menu.add_checkbutton(label = 'Auto save', variable = auto_
save)
```

So far, our menu—unlike the individual menu items—has no display name at all. We change this by calling the function **add_cascade()** of the menu bar object, which also ensures that the menu is actually displayed as a pull-down menu on our menu bar:

```
menubar_top.add_cascade(label = 'File',
menu = file_menu)
```

If you now start the program like this, a program window will open that has a menu bar with a “File” menu. Analogously, we could of course now place more pull-down menus on the menu bar.

The **add_...()** functions like **add_command()** can be called with numerous options, many of which can be found in ■ Table 22.1. For example:

```
file_menu.add_command(label = 'Save', background='#FF0000')
```

adds a menu item “Save” with a red background, the statement

```
file_menu.add_command(label = 'Save', state='disabled')
```

shows a deactivated (grayed out) menu item.

Of course, you can still modify your menu *after* you have created it. With **`delete(index_from, index_to)`** you can delete menu items by their numeric indices. With **`entryconfig(index, option=value)`** you change an **option** (for example **state** to enable/disable) for the menu item defined by **index**. You can query the options with **`entrycget(index, option)`**. Thus, to display the label (option **label**) of the second menu item, you can use the statement

```
print(file_menu.entrycget(1, 'label'))
```

(the indexes start, as always in Python, at 0!).

If you want to insert a menu entry later, use the functions **`insert_...(index, option)`**, where ... stands for the respective element (e.g. **command**, **separator**, **checkbox**), just as with **`add_...()`**. The entry is then inserted *after* the entry specified by **index**.

Our menu is now quite pretty to look at, but it has no functionality: clicking on a menu command does not trigger any action. This is because in our calls to **`add_command()`** of the pull-down menu object **file_menu**, we omitted the important **command** option, which allows us to specify a function to be called whenever the user clicks on the menu item. In this function we would then place the part of the program code that should be behind the menu item. We will deal with these event handlers in more detail in ► Sect. 22.2.5. At this point here we will concentrate entirely on creating a visually appealing interface.

Dealing with a menu in **tkinter** is, as you can see, not complicated at all, but it requires several different steps. Therefore, here again is the overview what you have to do to provide your program with a menu:

1. Create the menu bar using the constructor **`menubar = Menu(window)`**.
2. Add the menu bar to the window: **`window['menu'] = menubar`**.
3. Create a new pull-down menu that you want to place on the menu bar: **`pd_menu = Menu(menubar)`**.
4. Add menu items, i.e. commands (with **`add_command()`**), preference options (with **`add_checkbox()`** or **`add_radiobutton()`**) or separators (**`add_separator()`**) to the new pull-down menu, for example: **`pd_menu.add_command(label = 'Title of command')`**. Note that later, to add functionality to the menu item, we will include the **`command = event_function`** option at this point.
5. Attach your new pull-down menu to the menu bar and assign a title that will be visible in the menu bar: **`menubar.add_cascade(label = 'Title of menu', menu = pd_menu)`**.

In this section, we've looked at creating an application menu as the main menu in your program. However, the **Menu** object can also be used to open pop-up/context menus anywhere in your window thanks to the **`post(x, y)`** function, but this is beyond the brief introduction to **tkinter** in this section.

■ **Table 22.3** Special properties of the Entry widget

Option	Type	Meaning
justify	str	Alignment of the text in the entry field, ' left ' (left-aligned, the default), ' center ' (centered) or ' right ' (right-aligned)
selectback-ground	str	Background color of text selections
selectfore-ground	str	Font color of text selections
show	str	Character that is displayed (instead of the entered character); can be used for password entries, for example
width	int	Width of the entry field (in characters, not pixels)

22.2.3.3 Input Fields (Classes Entry and ScrolledText)

■ Single-Line Entry: Entry

Input fields are used to accept text input from the user. In **tkinter**, a distinction is made between single-line (widget class **Entry**) and multi-line text input (widget class **ScrolledText**). We will take a closer look at both in this section.

The widget of type **Entry** can be created very easily with the constructor and assigned to the window, in our example still to the **Tk** object **win**.

```
myentry = Entry(win)
myentry.pack()
```

As with the button before, we call the **pack()** method to display the control in our window. In ► Sect. 22.2.4 we will look at how we can influence the arrangement of the controls in the window in more detail.

In ■ Table 22.3 you will find an overview of the most important properties of the widget, which you can access in the usual way either by dictionary indexing **input['option']** or (but then only by writing) by **input.configure(option=value)**.

You can use the **get()** method at any time to retrieve the text that is currently in the Entry field, for example:

```
my_text = input.get()
```

Of course, you can also edit the text. This is done with the help of the **insert(index, string)** method. **index** specifies the text position (starting at 0) where you want to insert, **string** specifies the text to be inserted. So, at the beginning you can start with

```
input.insert(0, 'A first text in the input field.')
```

to preset the input field with a text. Instead of a real position index, you can simply specify **'end'**, **'insert'**, or **'anchor'** in the **insert()** method. Then the text is inserted at the end (**'end'**), at the current cursor position (**'insert'**), or at the beginning of the current selection (if there is currently an selection, otherwise simply at the beginning of the text). In this way, you can use the statement

```
input.insert('end', 'End of text.')
```

for example, to append text to the end. Similarly, you can delete text by using the **delete(from_index, to_index)** method. If you omit the optional argument **to_index**, only the character with the index **from_index** is removed.

With **icursor(index)** you can place the cursor at a certain position in the text, more precisely, *behind* the character specified by **index**. Note, however, that the counting of characters in **tkinter** starts at 1, which is not typical for Python. If you specify the number 0 as **index**, you place the cursor *in front of* the first character. This counting method applies everywhere where character indices are used.

In order for the cursor to actually be visible in the text, you must give the control focus, i.e., make it the currently active control in your window. You can do this, as with all other widgets, with the **focus()** method.

Finally, the method **selection_range(from_index, to_index)** allows you to select text. Again, you can work with the special constants **'end'**, **'insert'** and **'anchor'** known from **insert()**. You can query the currently selected text with **selection_get()**, the method **selection_present()** returns **True** if a text is selected.

Entry also allows you to work with the clipboard. You can add text to the clipboard using the method **clipboard_append(text)**; retrieve the contents of the clipboard using **clipboard_get()**, and clear the clipboard using **clipboard_clear()**.

■ Multiline Input: ScrolledText

If longer texts are to be edited, the widget **ScrolledText** is a good choice. To use it, the class **ScrolledText** must first be imported from the **scrolledtext** module of the **tkinter** package:

```
from tkinter.scrolledtext import ScrolledText
```

In many ways, the **ScrolledText** widget behaves exactly like the **Entry** widget. Most of what you have learned about **Entry** can be applied directly to **ScrolledText** (differences are that the **justify** and **show** options, the **icursor()** method and the insertion position **'anchor'** are not available).

Since **ScrolledText** allows multi-line input, it is only logical that text positions can also be addressed in accordance with a line/column scheme. For example, if we wanted to delete all text starting from the fifth character in the second line, this

could be done with the method call below. If the widget previously created with the constructor method **ScrolledText()** is named **st**:

```
st.delete(2.4, 'end')
```

Notice that where there was a single index at **Entry**, there now appears to be a fractional number: 2.4. However, this number is a coded row/column specification. It says: Second line, fifth character. So, while within a line the characters are counted starting from 0 (and 4 thus refers to the fifth character), as we are already used to from **Entry**, the line count in **tkinter** starts at 1! 2.4 thus refers to the *second* line, but the *fifth* character.

The **ScrolledText** widget is much more powerful than the **Entry** widget. For example, it inherently allows undoing or redoing of editing operations using the methods **edit_undo()** and **edit_redo()** (the **undo** option of the widget must be set to **True** beforehand). **ScrolledText** also allows you to assign names to text areas (called *tags*) and then access and edit the text areas using the *tags*. This way you can, for example, color different text areas differently. So, if you want to develop a text editor with syntax highlighting, the **ScrolledText** widget is not a bad starting point.

22.2.3.4 Text Outputs (Label)

Labels are static texts that are displayed in the application window. They usually do not offer the user any interaction options but are used to display information or to describe the function of controls if these do not have their own labels (such as buttons).

You can create and display a label widget as follows:

```
label = Label(win, text = 'Here is a fixed text')
label.pack()
```

■ Table 22.4 shows some important properties of labels. In addition, there are of course the standard properties provided by most widgets, which are listed in

■ Table 22.4 Special properties of the Label widget

Option	Type	Meaning
anchor	str	Position and orientation of the text. It is described using the English abbreviations of the cardinal points, for example 'ne' for <i>northeast</i> , i.e. top right; possible specifications are therefore: 'n' (top center), 'ne' (top right), 'e' (right center), 'se' (bottom right), 's' (bottom), 'sw' (bottom left), 'w' (left center), 'nw' (top left) and additionally 'center' (horizontally and vertically center)
width	int	Width of the label in characters. If not specified, the label is made wide enough to fit the text to be added
wrap-length	int	Number of characters after which the text on the label is to be wrapped. If not specified, there will be no wrapping

■ Table 22.1 and whose most significant feature for labels is certainly the **text** option, i.e., the text that is displayed on the label. In the example, we have set the text directly when calling the constructor, but of course, like all other properties, it can easily be changed later with **label.configure(text = new_text)** or with **label['text'] = new_text**.

22

22.2.3.5 Check Buttons and Radio Buttons

Check buttons (or checkboxes) and radio buttons are used to give the user a choice of several settings/options. While checkboxes allow the user to select several different settings/options, radio buttons only allow the user to select one setting/option at a time.

Check buttons and radio buttons are represented in **tkinter** by classes with the same names. Let's first create two radio buttons:

```
selection_var = IntVar()

selection_opt1 = Radiobutton(win, text = 'Option One',
                             variable = selection_var, value = 1)
selection_opt2 = Radiobutton(win, text = 'Option Two',
                             variable = selection_var, value = 2)

selection_opt1.pack()
selection_opt2.pack()
```

First, consider the constructor calls of the radio buttons themselves: Here, as usual, we first pass our window object **win** (of type **Tk**), to which the radio button should belong, and a text to be displayed as a selection option. Beyond that, however, there are two more arguments, **variable** and **value**.

To the **variable** argument, we assign a variable **selection_var**, which we created at the beginning of the code section, as an instance of the class **IntVar**. **IntVar** is one of several special variable classes that **tkinter** comes with. It is not simply an ordinary **int** variable. The special thing about this variable, after we assign it to the **variable** argument of the radio button constructor, is that it always reflects the current state of our radio button set, that is, it indicates which radio button option is currently selected. We define the value that **selection_var** takes when a particular radio button is selected via the **value** argument of **Radiobutton()**. So, if the user clicks on the “Option Two” radio button, **selection_var** has a value of 2. If the user clicks on “Option One”, **selection_var** takes a value of 1. The **IntVar selection_var** is automatically updated as soon as the selection situation of the radio button changes.

This is very convenient; however, there is one peculiarity to keep in mind when querying the value: If you execute the

```
print(selection_var)
```

the following is output to the Run console:

```
PY_VAR0
```

But this is not the value of the variable. This is because **selection_var** is not a normal **int** variable. We have to query its value with its **get()** method, even though this takes some getting used to. The following statement leads to success:

```
print(selection_var.get())
```

Analogously, there is a method **set()**, with which the value of the **IntVar** variable can be changed. Since we have linked the variable **selection_var** to the radio button via the argument **variable** of the constructor **Radiobutton()**, this also changes the selection among the radio buttons. With

```
selection_var.set(2)
```

we automatically select the radio button whose assigned value is 2, i.e. the “Left” radio button. The same could be achieved by calling the method **select()** of the respective radio button instance:

```
select_left.select()
```

Because we assigned our **IntVar** variable **selection_var** as **variable** option to *both radio buttons*, **tkinter** knows that these two radio buttons belong together and therefore only *one of them* may be selected at a time. So always make sure to assign the same variable as **variable** option to radio buttons that belong to the same “selection group”!

Check buttons function very similarly to radio buttons, with the difference, of course, that here several checkboxes can be selected at the same time, because the two checkboxes can be selected independently of each other. Whereas with radio buttons it was clear that if button 1 is selected, button 2 cannot also be selected,

■ **Table 22.5** Special properties of the **Checkbutton** and **Radiobutton** widgets

Option	Type	Meaning
indicatoron	Bool	If False , then instead of the circular (for radio buttons) or square (for check buttons) selection element, a real button is displayed that looks pressed down when the radio button/check button is selected
selectcolor	str	Background color of the circular (for radio buttons) or square (for check buttons) selection element

with two check buttons there are no longer just two, but four different states (both selected, none selected, only the first button selected, only the second button selected); the state of one therefore no longer depends on the state of the other button. Accordingly, you also need two different **IntVar** variables as **variable** arguments of the constructor **Checkbutton()**:

```
selection_var1 = IntVar()
selection_var2 = IntVar()

selection_opt1 = Checkbutton(win, text = 'Option One',
                             variable = selection_var1)
selection_opt2 = Checkbutton(win, text = 'Option Two',
                             variable = selection_var2)

selection_opt1.pack()
selection_opt2.pack()
```

You may have noticed that with the **Checkbutton()** constructor, we no longer have a **value** argument here. By default, the status variable **variable** takes the value **True** if the check button is selected, and **False** if it is not. However, with the optional arguments or widget options **onvalue** and **offvalue**, this can be changed if needed.

■ Table 22.5 gives an overview of the most important special properties of widgets of the types **Radiobutton** and **Checkbutton**, beyond the standard options listed in ■ Table 22.1.

22.2.3.6 Selection Lists (Listbox)

Selection lists, represented by the **tkinter** class **Listbox**, allow a (single or multiple) selection from an enumeration of text entries. After creating a list box using the **Listbox()** class constructor and displaying the list box in the window using the **pack()** method:

```
mylistbox = Listbox(win)
mylistbox.pack()
```

■ **Table 22.6** Special properties of the **Listbox** widget

Option	Type	Meaning
active-style	str	Specifies how the active entry, that is, the selected entry that currently has the focus, should be visually highlighted; possible values are ' underline ' (text underlined), ' dotbox ' (outline with dotted line), and ' none ' (no highlighting), where ' underline ' is the default value
height	int	The height of the list box but measured in entries rather than pixels; the default value is 10. If you have more than the number of entries set in height and want to ensure that the entry at the <i>index</i> position is displayed, you can call the see(index) method; it scrolls the list box so that the <i>index</i> entry is visible in any case
select-mode	str	Specifies how many entries can be selected at the same time and how they must be related; possible specifications are: ' single ' (one entry can be selected by clicking on the entry), ' browse ' (one entry can be selected by clicking or moving the mouse with the mouse button pressed), ' multiple ' (several entries can be selected, clicking on an entry sets the selection if it wasn't selected before or removes it if it was already selected) and ' extended ' (several entries can be selected by clicking while holding down the <CTRL> or <SHIFT> key); the default value is ' multiple ', which is somewhat unusual for Windows users

we can start adding items to the list box. To do this, we use the **insert(index, entry,...)** method:

```
mylistbox.insert('end', 'Beverly', 'Thomas', 'Marc',
                'Jimmy', 'Cathy')
```

insert() has some similarities to the **Entry** widget method of the same name: *index* first specifies the numeric index of the element to be inserted *after*; as with **Entry**'s **insert()** method, the value '**end**' can also be specified, which ensures that the entries are added to the end of the list. The list can also be extended by several entries at the same time. Please note that the indexing of the list entries starts at 0, as is typical for Python (unlike the indexing of the characters for the **Entry** widgets, but just like the “column indexing” for the multiline **ScrolledText** widgets).

Also analogous to **Entry**, you can delete list entries from the list box with the method **delete(index_from, index_to)**. In the case of the **Entry** widget, this method had removed individual characters from the contents of the **Entry** field, i.e., a string and thus a “character list”.

By calling the method **selection_set(index_from, index_to)**, you select entries in the list box as if the user had selected them.

In both **delete()** and **selection_set()**, *index_to* is an optional argument, i.e. an argument that can be omitted when calling the method.

Just as you can use **selection_set()** to select one (or more) entries, you can apply **selection_includes(index)** to check whether the element of the list denoted by *index* is currently selected.

Finally, with **selection_clear()** you can clear the current selection in the list box.

You determine the number of list entries using the **length()** method, while **get(index_from, index_to)** allows you to read the text entry of one or more entries. If you use the optional argument **index_to** and thus specify a range, you get a *tuple* with the text entries of the specified list entries as return values.

■ Table 22.6 lists some important options that you can customize in the now familiar ways for the **Listbox** widget.

In addition, you can of course work with the standard options that we looked at in ■ Table 22.1. Many of these properties can also be applied to *individual items* in the list box. To do this, use the **itemconfig(index, option=value)** method. For example, to give the third item a greenish background, you can execute the following statement:

```
mylistbox.itemconfig(2, background='#ED5036')
```

In ■ Table 22.6 you will once again find cases of options which can take one of several values, for example **activestyle**. In view of the rather difficult documentation situation with **tkinter**, the question naturally arises which expressions are permissible at all. You still don't know what the individual values do, but you can quickly find out by trial and error. So, the important thing in the first step would be to understand what options you have in the first place. There is a simple trick for this: You provoke an error. To do this, consider the following code:

```
mylistbox['activestyle'] = 'xxx'
```

Here we are trying to assign the value 'xxx' to the **activestyle** option, which likely is not a valid expression for this option. If we run a program with this statement, we get an error message, but that is exactly what we are trying to achieve in this case. In our example we get:

```
Traceback (most recent call last):
File "C:/Users/MyUser/Desktop/Documents/tkinter_hello_world.
py", line 138, in <module>.
names['activestyle']='xxx'
File "C:/Users/MyUser/AppData/Local/Programs/Python\
Python37-32/lib/tkinter/__init__.py", line 1489, in __
setitem__
self.configure({key: value})
File "C:/Users/MyUser/AppData/Local/Programs/Python\
Python37-32/lib/tkinter/__init__.py", line 1482, in configure
return self._configure('configure', cnf, kw)
File "C:/Users/MyUser/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
```


You can see from this example that although we are using the **showwarning()** function, which by default displays an exclamation mark as an icon, we can override the default behavior by specifying the **icon** option.

The function returns the value as a lowercase string containing the label of the button that was clicked, in our example **'yes'**, **'no'** or **'cancel'**. This function value is important in order to be able to react to the user's input accordingly.

Unlike the controls we looked at in the previous sections, the message box is not a class that we need to create an instance of. Rather, it is simply a function that is contained in the **messagebox** module. It's much the same with the file dialogs, which we'll look at in the next section.

22.2.3.8 File Open/File Save Dialogs

If you want to let the user select a file to open or to save something to, the **tkinter** module **filedialog** allows you to easily use the well known standard dialogs “Open file” and “Save file as” in your own programs.

First, we need to import the module. For the sake of simplicity, we import all classes and functions contained in the module, just as we did with the **messagebox** module in the previous section:

```
from tkinter.filedialog import *
```

After that, the **askopenfilename()** or **asksaveasfilename()** functions can be used to call the dialogs for opening or saving files. In the following example we select a file to open and print the result in the (run-)console:

```
filename = askopenfilename(defaultextension='txt',
    filetype=[('Text files', '*.txt'), ('All files', '*.*')],
    title='Open file...',
    initialdir='C:\\\\Windows')

print(filename)
```

The functions return the name (including the path) of the selected file. If the user cancels the dialog without selecting a file or entering a file name, an empty string is returned. As you can see from the example above, you can control the behavior of the dialog with some options. **filetypes** is a list (note the enclosing square brackets!) of tuples, each consisting of a description and a file extension; the user can then preselect the file types thus defined. The **title** option controls the dialog title; **initial-dir** is the directory whose contents should be displayed by default when the dialog opens; the backslash (\) as part of the pathname must be written twice, a single backslash would be interpreted by Python as an attempt to escape the character behind it; if you are not familiar with escaping, go back to ► Sect. 11.2.2.

By the way: You may have wondered in the last section, with the **messagebox** module, why we don't actually work with classes here, as with the other widgets,

but instead call functions like **showwarning()** or **askopenfilename()**. The answer is: Because it's easiest for us that way! These functions create the necessary class instances in the background. Since we're only interested in the results, in this case the filenames, and don't really want to work with the dialog (class) instances themselves, it's sufficient that an appropriate function provides us with the filename and saves us the trouble of creating instances of the necessary classes ourselves and working with them. The situation is different, of course, with the other widgets, which are permanently present in our application window and with which we also want to work later.

22.2.3.9 Other Widgets

In addition to the widgets we have looked at in the previous sections, there are a number of other controls that you can use when designing your program interface, for example **Canvas** (drawing area), **Spinbox** (input field with up-down arrows to adjust the value), **Treeview** (hierarchical display of elements; import from module **tkinter.ttk** module), **Notebook** (display of controls on tabs/tabs; also in **tkinter.ttk** module), **Progressbar** (in **tkinter.ttk** module), just to name a few. For working with images, which can be added to many controls (e.g., menu items, buttons) or used as background (e.g., of the main application window), there is also the class **PhotoImage**.

As you can see, there is still a lot to discover with **tkinter**! The only downside is the relatively poor documentation, at least for the above-mentioned widgets from the **tkinter.ttk** module. There is more or less sufficient "official" documentation (currently available at ► <https://docs.python.org/3/library/tkinter.ttk.html>). Beyond that, however, your own internet research is often the best option. Fortunately, you can usually find what you are looking for quickly, as many programmers before you have already had similar questions!

22.2.4 Arrangement of the Controls (Geometry Managers)

In the previous sections, we got to know a number of control elements that allow the user to control the program via the interface. To construct an interface that is visually appealing and easy to use, the various control elements must be placed on the interface where they should be.

To arrange the controls on the surface, **tkinter** offers three *Geometry Managers*, which we will deal with in more detail in the following sections. Each of the three follows a different basic principle of how the position of the controls on the surface is defined. Depending on what positioning will be the most suitable for your application, you can select the relevant Geometry Manager.

The three Geometry Managers are:

- **Pack:** **Pack** simply places the controls one next to the other, or one below the other. In the previous sections, we have already worked with this Geometry Manager by calling the widgets' **pack()** method. This caused the widgets to be placed one below the other, centering them in the available space.

- **Grid:** **Grid** mentally divides the available space on the surface into a grid of rows and columns and allows controls to be placed in each “cell” of this grid. **Grid** is well suited for constructing complex surfaces and should suffice for most purposes.
- **Place:** **Place** positions elements according to their *absolute* “coordinates”, which are measured in pixels, or *relative* (measured in proportions of the window width/height) to the upper left corner of the window.

22.2.4.1 Pack

When we took a closer look at the various widgets, we used the **pack()** method to ensure that these widgets were actually displayed on our program interface. However, the widgets were not only displayed, but of course also positioned in the window at the same time, one below the other. And that is exactly what the Geometry Manager **Pack** does: As the name suggests, it “packs” the various controls together, either vertically as a “stack” (which is the default orientation) or also horizontally as a “row”. Exactly where an element appears on the surface then depends primarily on how large its predecessors are in the stack (in the case of vertical alignment) or the row (in the case of horizontal alignment).

As an example, let’s consider a program interface that is used to enter a password:

```
from tkinter import *

win = Tk()

win.title('Demonstration of the Geometry Managers')
win.geometry('500x100')

prompt = Label(win, text = 'Please enter your password:')
pwd = Entry(win)
login = Button(win, text = 'Login')

pwd['show'] = '*'
pwd['width'] = 20
pwd.focus()
```

We create three widgets: a **label** called **prompt** to display a prompt, an entry field **pwd** to hold the password, and a login button to confirm the password entry. We then configure the entry field **pwd** to hide the password input and display only asterisks for the characters entered. We also set its width to 20 characters and assign it focus so that the user can start typing directly. Next, as before, we call the **pack()** method to let the **pack** Geometry Manager place the widgets on the surface. To make sure that the controls are displayed next to each other and not—as would

happen by default—below each other, we call **pack()** with the option **side**, which accepts the constants **'left'**, **'right'** and **'bottom'** as values for the direction of the widget positioning in addition to the default value **'top'**:

```
prompt.pack(side = 'left')
pwd.pack(side = 'left')
login.pack(side = 'left')

win.mainloop()
```

You can see the result in ■ Fig. 22.2. The widgets are now right next to each other. With the option **padx** (and vertically **pady**) we can add some spacing (*padding*) to the left and right of each widget, for example with:

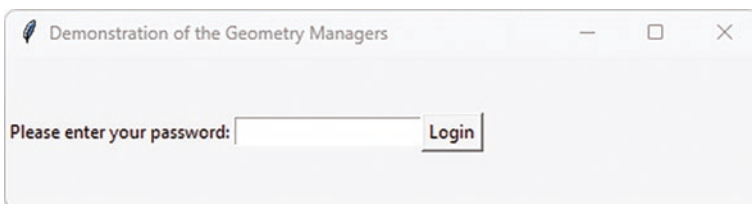
```
pwd.pack(side = 'left', padx = 5)
```

Two other important options can be used to control the behavior of **pack()** in even more detail: **expand**, which can take the values **1** and **0** or **True** and **False**, determines whether the **pack** Geometry Manager should use the full width it has available. In our example, this is the entire window width. If we were to change the default setting for all three widgets and tell **pack()** to use the full width with **expand=1**, i.e.

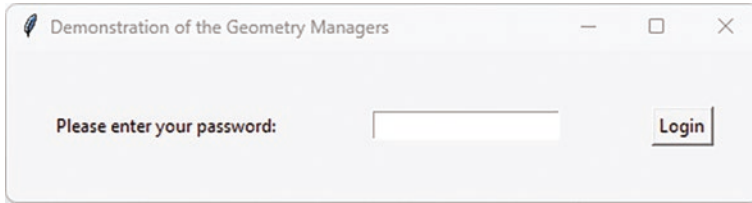
```
prompt.pack(side = 'left', expand = 1)
pwd.pack(side = 'left', expand = 1)
login.pack(side = 'left', expand = 1)
```

the result looks as in ■ Fig. 22.3.

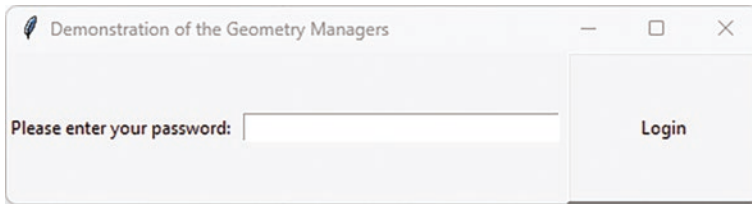
This still “packs” the widgets, but each widget has more space. Within the space available to each widget, the control is centered by default, but this can easily be adjusted with the specifications **'n'**, **'ne'**, **'e'**, **'se'**, **'s'**, **'sw'**, **'w'**, **'nw'**, and **'center'** (default) of the additional **anchor** option. These options are the same as for the **Label** widget (see ■ Table 22.4).



■ Fig. 22.2 Arrangement of the widgets with **pack(side = LEFT)**



■ Fig. 22.3 Arrangement of widgets with **expand** option



■ Fig. 22.4 Arrangement of the widgets with different options

Also, we can tell **pack()** to maximally expand the widgets in the space available to them, which is useful for some purposes, but often looks rather strange. To do this, we set the **fill** option to one of the constants **'x'** (expand horizontally), **'y'** (expand vertically), or **'both'** (expand both horizontally and vertically). This is exactly what happens in the following example with the input field, which we let expand in the horizontal direction, and the button, which we let expand in both directions to take up the available space, resulting in the interface seen in

■ Fig. 22.4:

```
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')
```

As this example shows, you will usually have to play around with the pack options **side**, **fill**, **expand**, **anchor** and **padx/pady** until you have found a good layout. With a little experience, you can implement a surface structure that you have sketched on paper or digitally.

22.2.4.2 Grid

While Pack tries to “pack” the controls side by side, the **grid** Geometry Manager sees the application window as a *grid* of *rows* and *columns*. Controls can be freely positioned within the grid. The individual “cells” in the grid are addressed using the Python-typical indices starting at 0.

Consider the password input example used in the last section:

```
from tkinter import *

win = Tk()

win.title('Demonstration of the Geometry Managers')
win.geometry('500x100')

prompt = Label(win, text = 'Please enter your password:')
pwd = Entry(win)
login = Button(win, text = 'Login')

pwd['show'] = '*'
pwd['width'] = 20
pwd.focus()
```

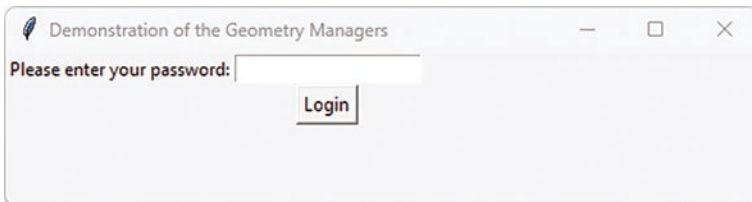
In the next step, we use the **grid(row = row, column = column)** method to position the prompt and the input field next to each other, with the login button below the input field:

```
prompt.grid(row = 0, column = 0)
pwd.grid(row = 0, column = 1)
login.grid(row = 1, column = 1)

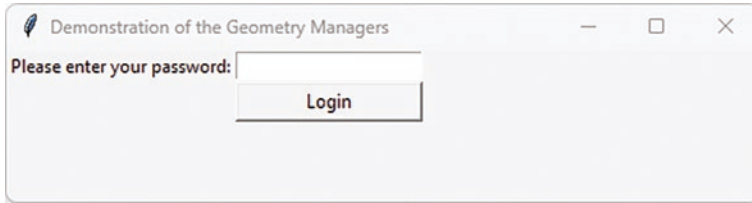
win.mainloop()
```

The result is shown in ■ Fig. 22.5.

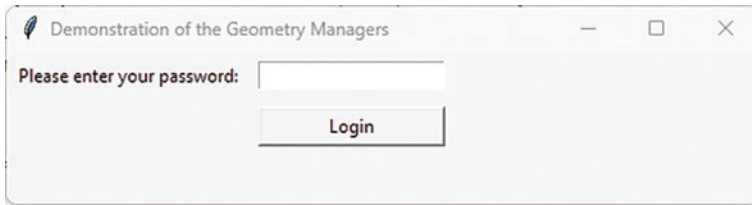
As you can see, the input field and the button are displayed in the same column, namely column 1 (i.e., the second column). The Geometry Manager sizes the cells so that the widgets just fit into them. Within the cells, the controls are centered by default (vertically, and—as you can see in the example of the button—horizontally).



■ Fig. 22.5 Arrangement of the widgets with **grid()**



■ Fig. 22.6 Arrangement of the widgets with `grid()` and the option `sticky = 'we'` for the button



■ Fig. 22.7 Arrangement of the widgets with `grid()` and the options `padx` and `pady`

However, the alignment of widgets within their cells can be easily influenced using the **sticky** option. As seen several times before, **sticky** is a compass direction specification. **sticky = 'w'** therefore means that the widget should be aligned “in the west”, i.e., on the left edge of the cell. At the same time, the cardinal directions can also be used to adjust the width of the widget so that, if it were smaller than the cell, it would still fill it completely. For example, **sticky = 'we'** causes the widget in its cell, whose width is determined by the wider input field in the row above it, to extend “from west to east”, i.e., over the entire width of the cell.

You can see the result in ■ Fig. 22.6.

Sometimes, however, this is not enough, and you want a widget to span multiple columns or multiple rows. In this case, the **rowspan** and **columnspan** options help, specifying the number of rows or columns spanned respectively. With

```
login.grid(row = 1, column = 0, sticky = 'we', columnspan = 3)
```

our button would extend from the left edge of the prompt to the right edge of the input field.

The **padx** and **pady** options, already known from **pack()**, allow you to specify a left/right and top/bottom spacing, respectively, which generally equalizes the display somewhat, as can be seen in the example in ■ Fig. 22.7, where **padx** and **pady** values of 5 were used.

The total size of the grid in your application window depends on where you have placed the widgets positioned furthest out. In our example, the grid has two

columns and two rows; if we were to place a widget to the right of the input field using `widget.grid(row = 0, column = 2)`, a third column would be added. Grid is a very popular Geometry Manager because it allows you to design effective surfaces with little effort, while providing intuitive positioning. The example in ► Sect. 22.2.6, where we develop a full **tkinter** application, will also use **Grid**.

22.2.4.3 Place

The third, last and probably least frequently used Geometry Manager in practice is **Place**. **Place** is used to place widgets either at an absolute position specified by **x** and **y** coordinates using the **x** and **y** arguments, or at a relative position. Relative means that the position relative to the upper left corner of the window is measured as a percentage of the window width or height. The statement

```
login.place(relx = 0.5, rely = 0.5)
```

positions the **login** button exactly in the middle of the window, namely after 50% of the window width (**relx**) and 50% of the window height (**rely**) from the upper left corner of the window. More precisely: The *upper left corner* of the button is placed there. With the **anchor** option, already known from **pack()**, another corner can also be specified as the “anchor” of the positioning. Again, the corner is specified by geographical directions, for example **'se'** for the southeastern, i.e., the lower right corner. The size of the widgets can also be specified, either absolutely with the help of the **width** and **height** options, or by a relative size specification. In this case, **relwidth** and **relheight** can be used to specify a width or height as a percentage of the window width or height.

The advantage of specifying relative positioning and sizes is that the position and size of the widgets adapt when the window size changes, an effect that can also be achieved with the Geometry Managers **pack** and **grid** (although a bit more tedious with the latter), but this is not the case when using **place** with absolute position specifications.

22.2.5 Events

■ The Command Option of the tkinter Widgets

So far, we have put together responsive interfaces, but they are largely without function. Nothing happens when you click on one of our buttons or menu items. That will change now.

Some widgets, such as **Button** or **Menu**, bring with them the ability to specify a function that will be called whenever the control is “triggered” by the user. Consider the following simple example of the now well-known conversion between Kelvin and degrees Celsius. A small converter with a graphical interface might look like this:

```

from tkinter import *

def convert():
    lb_result['text'] = 'Conversion result: ' + \
        str(round(float(en_kelvin.get()) - 273.15, 2)) + ' °C.'

def closeapp():
    quit()

win = Tk()
win.title('Kelvin-Celsius Conversion')
win.geometry('400x150')

menu_top = Menu(win)
win.config(menu = menu_top)
actionmenu = Menu(menu_top, tearoff = 0)

actionmenu.add_command(label = 'Convert', command = convert)
actionmenu.add_separator()
actionmenu.add_command(label = 'Close', command = closeapp)

menu_top.add_cascade(label = 'Action', menu = actionmenu)

lb_input = Label(text = 'Temperature in Kelvin:')
en_kelvin = Entry()
bt_convert = Button(win, text = 'Convert', command = convert)
lb_result = Label(width = 30)

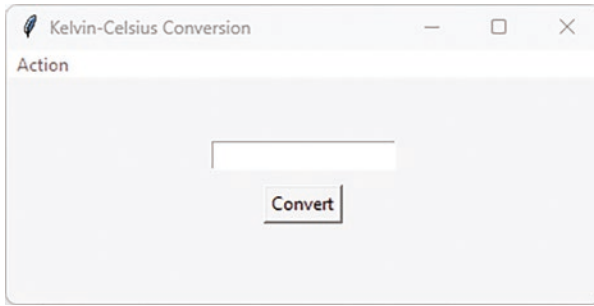
lb_result.pack()
en_kelvin.pack()
bt_convert.pack(pady = 10)
lb_result.pack(pady = 10)

win.mainloop()

```

The user interface of the program is shown in ■ Fig. 22.8.

In contrast to the previous sections, this time we used the **command** option when calling the button constructor method **Button()** to add menu items with **add_command()**. The **command** option is assigned a function, called an *event handler* or *callback function*, that is automatically called whenever the control is triggered by the user. For example, if our “Convert” button is clicked, the **convert()** function is automatically called. We defined the **convert()** function earlier in the program (the function definition must *precede* the assignment to the **command** option in the program source code); it simply performs the conversion and prints the result on the **lb_result** label. Note that when assigning the event handler function to the



■ Fig. 22.8 Kelvin to Celsius converter

command option, no round brackets are specified after the function name. This is because the **command** option is simply passed the function *object*; you'll recall that in Python, functions are also objects—only when defining a function (as you'll see below) or calling a function would we need to include the round brackets, but not when we mean the function object itself.

■ Binding Event Handlers

- **tkinter** knows another, much more powerful way to react to events. This is done with the help of the widget method **bind()**. With the statement

```
bt_convert.bind('<button-1>', convert)
```

- we could have achieved the same effect as with the **command** option in the **Button()** constructor above. The **bind(event, eventhandler_function)** method binds an event handler function to an event. Henceforth, our event processing **main-loop()** watches to see if the event is triggered and calls the event handler function if it is. The string '**<Button-1>**' represents the event that the left/primary mouse button ("Button 1") is pressed (the right mouse button, by the way, would be '**<Button-3>**', '**<Button-2>**' the middle mouse button). Besides these button events, there are a variety of such events to which we can bind event handlers; here are some examples:
 - **<DoubleButton-1>**: Double click with the left mouse button.
 - **<Enter>** and **<Leave>**: The user has entered the area of the control with the mouse pointer or the mouse pointer has left the area of the control.
 - **a, b, c, ...**: The respective letter was pressed.
 - **<Key>**: *Any* letter was pressed.
 - **<F1>**, ...: The respective function key was triggered.
 - **<Escape>**, **<BackSpace>** (remove), **<Delete>**, **<Tab>** (tabulator), **<Return>** (Return or Enter), **<Shift_L>** (Shift), **<Control_L>** (Ctrl), **<Alt_L>** (Alt), **<End>**, **<Home>**, **<Left>** (left arrow), **<Up>** (Up arrow), **<Right>** (Right arrow), **<Down>** (Down arrow), **<Print>**, **<Insert>**: The respective special key was pressed.

Key combinations can also be represented with this method: If, for example, you want to bind a function to the event that <CTRL> and <S> were pressed simultaneously, you can simply specify '<Control_L>S' as the event.

Let's take a closer look at the event handlers that we bind to an event with **bind()**. These functions are automatically passed an argument of type **Event**. Accordingly, we have to adapt our previous event handlers, which we assigned to the **command** option of our button, for example, because these event handlers do not require any argument. The change is marginal, but avoids a runtime error:

```
def convert(ev = None):
    lb_result['text'] = 'Conversion result: ' \
        + str(round(float(en_kelvin.get()) - 273.15, 2)) + ' °C.'
```

We don't need to do anything with the **Event** object **ev** but the event handler function must provide the argument. By giving the argument a default value (namely **None**), we also make the function callable for the **command** option of our menu item "Convert", because it calls the event handler *without* an argument. So, the event handler has to cope with being called with an argument as well as with the argument being omitted.

But what is the content of this event object? The event object provides some information about the event, especially:

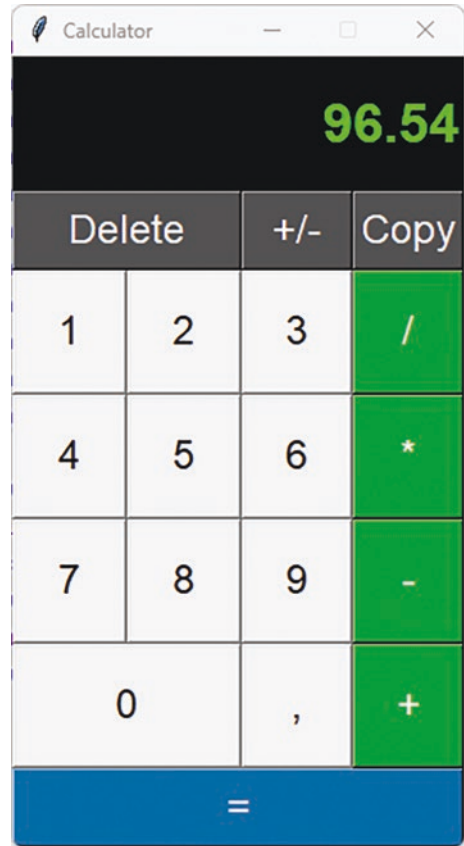
- **x,y**: The mouse position (relative to the upper left corner of the window) where the event was triggered (interesting for click events).
- **widget**: The widget that triggered the event.
- **char**: The pressed character key (interesting especially for the <Key> event).

By the way: You can also bind events directly to the application window, in our case to the **Tk** object **win**. If an event is triggered for a widget, the system first automatically checks whether an event handler is bound to this event for the respective widget. If this is not the case, it is checked whether the "next higher" object, in our case the application window, has an event handler for this event. In this sense, the existence of event handlers is checked from the "specific to the general"; the event handlers form a hierarchy, so to speak.

22.2.6 Example: Calculator Application

In this section we will develop a simple calculator using **tkinter**. The calculator should handle the four basic arithmetic operations and allow copying the result of calculations to the clipboard. You can see the result in ■ Fig. 22.9.

■ **Fig. 22.9** The interface of the calculator application



Now let's look at the code step by step:

```
1  from tkinter import *
2  from tkinter.font import *
3  from functools import *
4
5
6  # Define eventhandler functions for buttons
7  def digit_operator_press(digit_operator):
8      display['text'] = display['text'] + digit_operator
9
10
11  def delete_press():
12      display['text'] = ''
13
14
15  def copy_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 equal_press():
25     display['text'] = str(eval(display['text']))
26
27
28 # Define eventhandler for <ENTER> key
29 def enter_press(ev):
30     equal_press()
31
32
33 # Create application window
34 win = Tk()
35 win['background'] = '#000000'
36 win.title('Calculator')
37 win.geometry('268x470')
38 win.resizable(height=False, width=False)
39
40 # Define fonts for button and display
41 digit_font = Font(family = 'Arial', size = 18)
42 display_font = Font(family = 'Arial', size = 24,
43                     weight = 'bold')
44
45 # Create the display
46 display = Label(text = '',
47                 background = '#000000',
48                 foreground = '#00FF00')
49 display['width'] = 13
50 display['font'] = display_font
51 display['height'] = 2
52 display['anchor'] = 'e'
53
54 # Define buttons
55 delete_op = Button(win,
56                    text = 'Delete',
57                    width = 9,
58                    height = 1,
59                    font = digit_font,
60                    foreground = '#FFFFFF',
61                    background = '#4C4E4F',
62                    command = delete_press)
63 plusminus_op = Button(win,
64                       text = '+/-' ,
65                       width = 4,
66                       height = 1,
67                       font = digit_font,
68                       foreground = '#FFFFFF',
69                       background = '#4C4E4F',
70                       command = plusminus_press)
71 copy_op = Button(win,

```

```
72         text = 'Copy',
73         width = 4,
74         height = 1,
75         font = digit_font,
76         foreground = '#FFFFFF',
77         background = '#4C4E4F',
78         command = copy_press)
79     digit1 = Button(win,
80                    text = '1',
81                    width = 4,
82                    height = 2,
83                    font = digit_font,
84                    command = partial(digit_operator_press, '1'))
85     digit2 = Button(win,
86                    text = '2',
87                    width = 4,
88                    height = 2,
89                    font = digit_font,
90                    command = partial(digit_operator_press, '2'))
91     digit3 = Button(win,
92                    text = '3',
93                    width = 4,
94                    height = 2,
95                    font = digit_font,
96                    command = partial(digit_operator_press, '3'))
97     digit4 = Button(win,
98                    text = '4',
99                    width = 4,
100                   height = 2,
101                   font = digit_font,
102                   command = partial(digit_operator_press, '4'))
103     digit5 = Button(win,
104                    text = '5',
105                    width = 4,
106                    height = 2,
107                    font = digit_font,
108                    command = partial(digit_operator_press, '5'))
109     digit6 = Button(win,
110                    text = '6',
111                    width = 4,
112                    height = 2,
113                    font = digit_font,
114                    command = partial(digit_operator_press, '6'))
115     digit7 = Button(win,
116                    text = '7',
117                    width = 4,
118                    height = 2,
119                    font = digit_font,
120                    command = partial(digit_operator_press, '7'))
121     digit8 = Button(win,
122                    text = '8',
123                    width = 4,
124                    height = 2,
125                    font = digit_font,
```

```

126         command = partial(digit_operator_press, '8'))
127     digit9 = Button(win,
128         text = '9',
129         width = 4,
130         height = 2,
131         font = digit_font,
132         command = partial(digit_operator_press, '9'))
133     digit0 = Button(win,
134         text = '0',
135         width = 9,
136         height = 2,
137         font = digit_font,
138         command = partial(digit_operator_press, '0'))
139     divide_op = Button(win,
140         text = '/',
141         width = 4,
142         height = 2,
143         font = digit_font,
144         foreground = '#FFFFFF',
145         background = '#10a605',
146         command = partial(digit_operator_press, ' / '))
147     multiply_op = Button(win,
148         text = '*',
149         width = 4,
150         height = 2,
151         font = digit_font,
152         foreground = '#FFFFFF',
153         background = '#10a605',
154         command = partial(digit_operator_press, ' * '))
155     minus_op = Button(win,
156         text = '-',
157         width = 4,
158         height = 2,
159         font = digit_font,
160         foreground = '#FFFFFF',
161         background = '#10a605',
162         command = partial(digit_operator_press, ' - '))
163     plus_op = Button(win,
164         text = '+',
165         width = 4,
166         height = 2,
167         font = digit_font,
168         foreground = '#FFFFFF',
169         background = '#10a605',
170         command = partial(digit_operator_press, ' + '))
171     point_op = Button(win,
172         text = ',',
173         width = 4,
174         height = 2,
175         font = digit_font,
176         command = partial(digit_operator_press, '.'))
177     equal_op = Button(win,
178         text = '=',
179         width = 10,

```

```

180             height = 1,
181             font = digit_font,
182             foreground = '#FFFFFF',
183             background = '#0570A6',
184             command = equal_press)
185
186     # Define eventhandler for Enter key
187     win.bind('<Return>', enter_press)
188
189     # Place buttons on interface
190     display.grid(row = 0, column = 0, columnspan = 5,
191                 sticky = 'news')
192     delete_op.grid(row = 1, column = 0, columnspan = 2,
193                   sticky = 'news')
194     plusminus_op.grid(row = 1, column = 2, sticky = 'news')
195     copy_op.grid(row = 1, column = 4, sticky = 'news')
196     digit1.grid(row = 2, column = 0, sticky = 'news')
197     digit2.grid(row = 2, column = 1, sticky = 'news')
198     digit3.grid(row = 2, column = 2, sticky = 'news')
199     digit4.grid(row = 3, column = 0, sticky = 'news')
200     digit5.grid(row = 3, column = 1, sticky = 'news')
201     digit6.grid(row = 3, column = 2, sticky = 'news')
202     digit7.grid(row = 4, column = 0, sticky = 'news')
203     digit8.grid(row = 4, column = 1, sticky = 'news')
204     digit9.grid(row = 4, column = 2, sticky = 'news')
205     digit0.grid(row = 5, column = 0, columnspan = 2,
206                sticky = 'news')
207     point_op.grid(row = 5, column = 2, sticky = 'news')
208     divide_op.grid(row = 2, column = 4, sticky = 'news')
209     multiply_op.grid(row = 3, column = 4, sticky = 'news')
210     minus_op.grid(row = 4, column = 4, sticky = 'news')
211     plus_op.grid(row = 5, column = 4, sticky = 'news')
212     equal_op.grid(row = 6, column = 0, columnspan = 5,
213                  sticky = 'news')
214
215     # Event loop
216     win.mainloop()

```

■ ■ Lines 1–3: Import Instructions

We first import all classes from the **tkinter** module. Furthermore, the “display” of our calculator will use a special font, so we also import everything from the **font** module. We need the last import statement to make the event functions that are supposed to react to the user’s digit input a bit simpler. More about this shortly.

■ ■ Lines 6–30: Event Handler Functions for Buttons

These are the event functions that react to the different user actions. We’ll go into these functions in a little more detail at the end, but first we want to look at the interface. With knowledge of the interface, it is also easier to understand how the event handlers work.

■ ■ Lines 33–38: Window of the Application

Our calculator should have a black background and a fixed window size that cannot be changed by the user.

■ ■ Lines 40–52: Fonts and Display

Next, we define two fonts, **digits_font** for the digits on the calculator keys, and **display_font** for the display on the calculator. The display itself has a green foreground and a black background color. With **display['anchor'] = 'e'** we align its content “east”, i.e., right aligned.

■ ■ Lines 54–184: Create the Buttons

Next, we create the buttons for our window **win**, which are characterized by a number of properties, namely their label (**text**), their **width** and **height** in text characters, the **font**, the **foreground** and **background** color, and the event handler (**command**) that is called when the user clicks on the button.

For the event handlers, we use a little trick to avoid having to write a separate event handler for each of the 10 digits and each arithmetic operator. We have defined only one event handler named **digit_operator_press()**, which we call with one argument, namely the respective digit or the respective operator. But the option **command** of the **tkinter button** object must be passed a function object, not a call to a function with arguments. Therefore, we use the **partial** function from the **functools** module to create a function object in which the argument is already “baked in”. Since the return value of **partial()** is an object not unlike the **function** object, but which already contains the argument value, we can use this return value as the value for the **command** option of our buttons.

■ ■ Lines 186–187: Event Handler for the <ENTER> Key

To make it possible to trigger the computation not only by clicking the equals sign button on the interface, but also by pressing <ENTER> on the keyboard, we bind an event handler (lines 29–30) to the event of pressing <ENTER>, which does nothing but call the event handler that is triggered when the equals sign button is pressed (lines 24–25). We still need to separate the two event handlers, because the event handler that processes the button press is passed an event object as an argument by default, which we must accept even if we don’t process it at all.

With that in mind, let’s take a quick look at the other event handlers:

- **digit_operator_press()** (lines 6–8): The event handler is passed the clicked digit or arithmetic operator as an argument. The value of the pressed button is simply appended to the existing display content in the display.
- **delete_press()** (lines 11–12): Clears the contents of the display.
- **copy_press()** (lines 15–17): First clears the clipboard, then appends to the clipboard the current contents of the display. The functions **clipboard_clear()** and **clipboard_append()** are used for this purpose, both of which are conveniently provided by the **Tk** class and are therefore also available for our window object **win**.
- **plusminus_press()** (lines 20–21): When the plus/minus button is pressed, we simply prepend a minus to the current display content. Strictly speaking, we would

have to check if there is already a minus and then remove it. But just as with the other inputs (for example, we don't check whether the user enters two operators in a row), we make it easy for ourselves at this point and rely on the common sense of the user.

- **equal_press()** (lines 24–25): This is the event handler mentioned earlier, which is called when the user requests the result of the calculation. Here we use the **eval()** function, which evaluates a Python expression and returns its result. In our case, the Python expression is simply the sequence of numbers and operators entered by the user and displayed in our calculator display. However, **eval()** can also be used to execute completely arbitrary Python code passed as a string argument.

■ ■ Lines 189–213: Placing the Buttons on the Interface

Now we just need to place the display and the buttons on the surface. To do this, we need the **grid** Geometry Manager, so we use the **grid()** method to assign the individual controls their place in the grid by specifying their **row** and **column** number. With the **columnspan** option, we make some elements extend horizontally across more than one cell of the grid, for example the display. With the value **news** (*north + east + west + south*) of the **sticky** option we specify that the elements in their respective grid cell should extend completely, i.e., fill the cell completely.

■ ■ Lines 215–216: Event Loop

With **mainloop()** we start the event processing for our calculator. From now on, the user interface reacts to the user's input by calling the corresponding event handlers.

22.3 Working with Files

Working with files is very simple in Python. It is done in three steps, not unlike most other programming languages:

1. The file is opened (possibly created in the first place)
2. The file is processed (it is read from it, written to it or appended to it)
3. The file is closed after all work is completed

■ Opening Files

A file is represented by a *file object*. We create such a file object with the help of the standard Python function **open(filepath, mode)**. The argument **mode** describes the editing mode in which the file is to be opened. Possible values of the mode are:

- **"w"**: The file is opened for *writing*. The file pointer is set to the beginning of the file. Any existing contents of the file are completely replaced. If the file does not yet exist, it will be newly created. Reading from the file in this mode is not possible.
- **"a"**: The file is opened for *appending*. The file pointer is placed at the end of the file. Contents that are written to the file are appended to it. Reading from the file is not possible in this mode.

- **"r"**: The file is opened for *reading*. The file pointer is set to the beginning of the file. Writing to the file is not possible in this mode.
- **"r+"**: The file is opened for reading *and* writing.

Note that if you are working on a Windows system, you must *escape* the backslashes that separate the path components in the path specification with another backslash, otherwise Python will regard them as an attempt to escape the *following character* and thus assign a special control function to it (if you are no longer familiar with escaping, go back and review ► Sect. 11.2.2). For example, if we wanted to open the file **test.txt** in the directory **C:\Programming** for writing, we would first have the function **open()** create a corresponding file object (which we call **file** here for simplicity):

```
file = open("C:\\Programming\\test.txt", "w")
```

This object has a number of properties that allow us to better understand its character: **file.name** gives us the file name as a full path specification, **file.mode** the mode in which we opened the file. Furthermore, whether the file is readable and/or writable can be determined with the methods **file.readable()** and **file.writable()**, which return a **bool** value each.

■ Editing the File

To *write* to the file, the file object provides the methods **write(text)** and **writelines(lines)**.

write() simply writes *a* string to the file, without a trailing newline unless **text** contains the escape sequence **\n** at the end (for example, **"This is a newline\n text"**).

writelines(), in contrast, writes *several* strings that are passed as an array; the name of the function is somewhat misleading, because **writelines()** also does not write a newline at the end of each string. So, if you want a line break after each string, you have to add it yourself:

```
lines = ["line 1\n", "line 2\n"]
file.writelines(lines)
```

The write methods each return the number of characters written as a function value.

To *read* from a file the functions **read()**, **readline()** and **readlines()** are available. **read()** reads the *entire* file contents and returns it as a string. With an optional argument, a *certain number of characters* (measured from the current position of the file pointer) can be read. The file pointer starts at the beginning of the file and moves accordingly with each read operation. Consider the following example file:

```
Line number one
One more line
Last line
```

With **read(3)**, after opening the file (the file pointer is then at the beginning of the file), we would first read the string "**Lin**". After that, the file pointer is on the "**e**" of "**line**". Another **read(19)** would then return the next 19 characters, i.e. "**e number one\nOne mo**". Note that the line break also counts as a character, and as *exactly one* character (although it is represented as an escape sequence in the form **\n** with two characters). After this reread, the file pointer is now on "**r**" of "**more**".

The **readline()** and **readlines()** functions proceed differently than **read()**; they read in one or more *lines* respectively. **readline()** always reads in exactly the next line, while **readlines()** reads in all lines or the number of lines passed as an optional argument, returning an *array of strings*. The **readline()** and **readlines(1)** method calls thus differ in that the result of the **readline()** call results in a **str** value, while **readlines()** returns an array, which in this case contains only a single string as an element.

If the file pointer is not at the beginning of a line, but in the middle of a line, as after the call to **read(3)** above, then **readline()** and **readlines()** read from the character on which the file pointer is currently located. The beginning of the line (in our example, the first three characters already read with **read()**) is then no longer read.

When reading, you can use **seek(characterindex)** to set the file pointer to the character specified by **characterindex**, counting from the beginning of the file. The current position of the file pointer is returned by the **tell()** method of the file object.

By the way: If you open the file in "**r+**" mode (read *and* write), you can actually use the same file object for both operations. However, writing is always done at the end of the file, reading at the current position of the file pointer, which behaves the same way as when opening the file in "**r**" mode.

■ Close the File

After editing the file, close it with the **file.close()** method. The file object will continue to exist, but its property **file.closed** will now take the value **True**, indicating that the object is not available for reading or writing any more. However, since the file object still carries the path to the file as its **name** property, you can simply “reactivate” it:

```
file = open(file.name, "r")
```

? 22.2 [15 min]

Write a program that queries the user for the name of a file (with its complete path). A certain percentage (specified by the user) of the file content is then to be displayed as a preview. Line breaks shall be removed from the preview display so that the display is as compact as possible.

22.4 Exercise: Development of a Simple Text Editor

The following exercise combines many of the things you have learned in this chapter to develop a useful little application.

The task is to program a simple text editor with **tkinter**. This should allow you to create new files or open existing files, and then edit and save the files again, either under the current name or a new name. Also, the user should be able to copy text to the clipboard and paste it from the clipboard. The commands of the editor shall be selectable via a menu as well as via a button bar.

You will want to use the **ScrolledText** widget from the **tkinter** module **scrolled-text**. Therefore, make sure you include the line

```
from tkinter.scrolledtext import ScrolledText
```

in your imports even if you import the rest of the **tkinter** widgets with **from tkinter import ***.

Test your program extensively!

The estimated time to complete this task is 120 min. You should have a quiet space to focus on this piece of development work. If the task still seems too challenging, do not spend hours trying to develop the editor yourself, but read the code in the sample solution and try to understand it, at first without referring to the explanatory notes in the solution.

22.5 Summary

In this chapter, we looked at how to input and output data via the console. We also looked at how graphical user interfaces can be implemented in Python to allow the user to interact with your program conveniently.

Be sure to take the following points from this chapter:

- In the Python console, you can always print objects using the built-in Python function **print(object)**.
- Information can be requested from the user using the **input(prompt)** method, which always returns the user input as a string (so the user input must be converted if necessary).
- Graphical user interfaces (GUIs) can be easily implemented with the **tkinter** library, which is part of the standard Python package.
- A **tkinter** program always consists of creating a **Tk** object using the constructor function of the same name, creating and configuring the controls (widgets), defining the arrangement of the controls (using a Geometry Manager) and starting event processing (method **mainloop()** of the **Tk** object).
- The most important controls (widgets) for graphical user interfaces in **tkinter** are **Button**, **Menu**, **Entry** (text input), **Label** (text display), **Checkbutton** (multiple selection of options), **Radiobutton** (single selection of options) and **Listbox** (list-like display of text entries with single or multiple selection).
- Important standard dialogs that can be used from within **tkinter** (more specifically, the **tkinter** module **filedialog**) are: **messagebox** (with several variants that differ in the displayed icons and buttons) for displaying text messages; and

askopenfilename() and **asksaveasfilename()** for querying file paths when opening or saving files.

- The widgets are configured via options; some options (but not their values!) are common to almost all widgets (for example, the **background** color and the **font**), others are specific to the particular control.
- All widgets have the **config(option = value, ...)** method, which can be used to set the values of the options. In addition, the options can be accessed in the form **widget['option']** as if they were a dictionary.
- Widgets are arranged on the program surface with the help of a Geometry Manager; **tkinter** has three such arrangement tools with **pack** (arrange directly next to/below each other), **grid** (arrange along an imaginary grid) and **place** (arrange by specifying coordinates relative to a reference point), that can be called with the standard methods **pack()**, **grid()** and **place()** of each widget.
- To read and write data from or to files, the file in question is first opened with the built-in Python function **open(filename, mode)**; this returns a **File** object.
- Modes for editing files are **r** (read), **w** (write), **a** (append), and **r+** (read and write).
- The methods **read()** and **readlines()** as well as **write()** and **writelines()** of the **File** object can be used to read from or write to the file.
- The **close()** method of the **File** object closes the file again after processing is complete.

22.6 Solutions to the Exercises

■ Exercise 22.1

```
# First option: Three print()-commands (each automatically #
# ending with \n)

print('First line')
print('Second line')
print('Third line')

# Second option: One string, lines separated by the \n escape
# sequence.
print('First line\nSecond line\nThird line')

# Third option: Output of three string objects, using the \n
# escape sequence as separator.
print('First line', 'Second line', 'Third line', sep = '\n')
```

■ Exercise 22.2

The program could look like this:

22

```
filename = input("Please enter a filename (incl. path): ")

percent = input("Percent of content to be previewed " +
               "(whole number, e.g. 10 for 10%): ")

previewfile = open(filename, "r")
contents = previewfile.read()
previewfile.close()

content = contents.replace("\n", "")

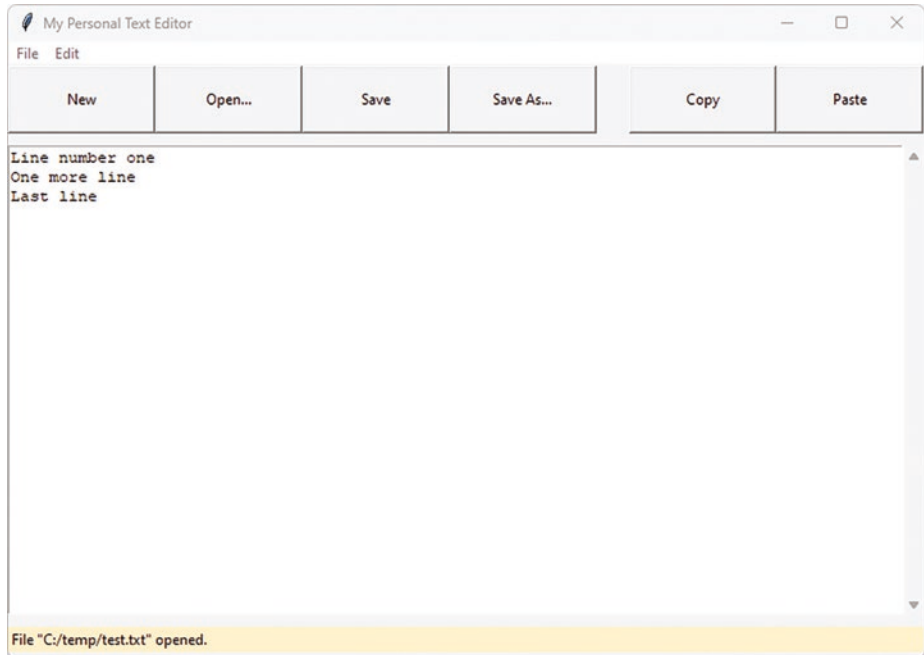
length_total = len(content)
length_preview = int(length_total * int(percent) / 100)

print("### Preview: ", length_preview, " characters of ",
      length_total, "characters ###")
print(content[0:length_preview], "\n###\n")
```

The file is first opened in read mode ("r") and its entire contents are read with **read()**. After that, the file can be closed again, because the string variable **content** now contains the entire file content, and we will only continue to work with this content. After we have cleaned line breaks from the content by removing the escape sequence **\n** with the string method **replace()**, we select the desired number of characters in the last statement, which we have previously calculated from the preview percentage specified by the user, and display them on the screen. When selecting characters from the string, it is important to ensure that the selection limits are integers. We achieve this by using **int()** to store the result as an integer variable when calculating the preview length.

■ Programming Task Text Editor

Many variants are of course possible in the development of the text editor. The user interface of the solution presented here can be seen in ■ Fig. 22.10.



■ Fig. 22.10 User interface of our text editor

The code looks like this:

```

1  from tkinter import *
2  from tkinter.filedialog import *
3  from tkinter.scrolledtext import ScrolledText
4
5  # Define event handler functions for buttons and menus
6
7  def new_press():
8      global filename
9      text.delete(1.0, 'end')
10     filename = ''
11     status['text'] = 'Unsaved new file'
12
13
14  def open_press():
15      global filename
16      fname = askopenfilename(defaulttextextension = 'txt',
17                              filetypes = [('Text Files', '*.txt'),
18                                             ('All Files', '*.*')],
19      title = 'Open...',
20      initialdir = 'C:\\Windows')
21      textfile = open(fname, 'r')
22      text.delete(1.0, 'end')
23      text.insert(1.0, textfile.read())
24      textfile.close()

```

```

25
26     status['text'] = 'File "' + fname + '" opened.'
27     filename = fname
28
29
30     def saveas_press():
31         global filename
32         fname = asksaveasfilename(defaultextension = 'txt',
33             filetypes = [('Text Files', '*.txt'),
34                 ('All Files', '*.*')],
35             title = 'Save File As...',
36             initialdir = 'C:\\Windows')
37         textfile = open(fname, 'w')
38         textfile.write(text.get(1.0, 'end'))
39         textfile.close()
40
41         status['text'] = 'File "' + fname + '" saved.'
42         filename = fname
43
44
45     def save_press():
46         global filename
47         textfile = open(filename, 'w')
48         textfile.write(text.get(1.0, 'end'))
49         textfile.close()
50
51         status['text'] = 'File "' + filename + '" saved.'
52
53
54     def copy_press():
55         selection = text.selection_get()
56         text.clipboard_clear()
57         text.clipboard_append(selection)
58
59
60     def paste_press():
61         text.insert(text.index('insert'), text.clipboard_get())
62
63
64     def copy_press_key(event):
65         copy_press()
66
67
68     def paste_press_key(event):
69         paste_press()
70
71
72     def quit_press():
73         win.quit()
74
75
76     # Create application window
77     win = Tk()
78     win.title('My Personal Text Editor')

```



```
79 win.geometry('760x490')
80 win.resizable(height = True, width = True)
81
82
83 # Set up the menu
84 menubar = Menu(win)
85 win.config(menu = menubar)
86
87 file_menu = Menu(menubar, tearoff=0)
88 edit_menu = Menu(menubar, tearoff=0)
89
90 menubar.add_cascade(label = 'File', menu = file_menu)
91 menubar.add_cascade(label = 'Edit', menu = edit_menu)
92
93 file_menu.add_command(label = 'New',
94                       command = new_press)
95 file_menu.add_command(label = 'Open...',
96                       command = open_press)
97 file_menu.add_command(label = 'Save',
98                       command = save_press)
99 file_menu.add_command(label = 'Save As...',
100                      command = saveas_press)
101 file_menu.add_separator()
102 file_menu.add_command(label = 'Quit',
103                      command = quit_press)
104
105 edit_menu.add_command(label = 'Copy',
106                      command = copy_press)
107 edit_menu.add_command(label = 'Paste',
108                      command = paste_press)
109
110
111 # Create the control elements
112 new_button = Button(win,
113                    text = 'New',
114                    height = 3,
115                    width = 16,
116                    command = new_press)
117 open_button = Button(win,
118                    text = 'Open...',
119                    height = 3,
120                    width = 16,
121                    command = open_press)
122 save_button = Button(win,
123                    text = 'Save',
124                    height = 3,
125                    width = 16,
126                    command = save_press)
127 saveas_button = Button(win,
128                    text = 'Save As...',
129                    height = 3,
130                    width = 16,
131                    command = saveas_press)
132 seplabel = Label(win,
```

```

133     text='',
134     height =3,
135     width= 3)
136 copy_button = Button(win,
137     text = 'Copy',
138     height = 3,
139     width = 16,
140     command = copy_press)
141 paste_button = Button(win,
142     text = 'Paste',
143     height = 3,
144     width = 16,
145     command = paste_press)
146
147 text = ScrolledText(win)
148 text.bind('<Control-c>', copy_press_key)
149 text.bind('<Control-v>', paste_press_key)
150
151
152 # Set up the status bar
153 status = Label(win,
154     text = 'No file openend.',
155     anchor = 'w',
156     background = '#FFEF4')
157     filename = ''
158
159
160 # Place buttons on surface
161 new_button.grid(row = 0, column = 0, sticky = 'news')
162 open_button.grid(row = 0, column = 1, sticky = 'news')
163 save_button.grid(row = 0, column = 2, sticky = 'news')
164 saveas_button.grid(row = 0, column = 3, sticky = 'news')
165
166 seplabel.grid(row = 0, column= 4, sticky = 'news')
167 copy_button.grid(row = 0, column = 5, sticky = 'news')
168 paste_button.grid(row = 0, column = 6, sticky = 'news')
169
170 text.grid(row = 1, column = 0, columnspan = 7, pady = 10,
171     sticky = 'news')
172
173 status.grid(row = 2, column = 0, columnspan = 7,
174     sticky = 'news')
175
176
177 # Event loop
178 win.mainloop()

```

■ ■ Lines 5–73: Eventhandler Functions for Buttons and Menus

We'll take a closer look at how event handlers work below once it's clear which components will make up the interface.

■ ■ Lines 76–80: Application Window

The window **win** is created and scaled as a **Tk** object. It should be resizable for the user.

■ ■ Lines 83–108: Menu Setup

A new menu bar is created for the **win** window and two drop-down menus, **file_menu** and **edit_menu** are placed on the bar. The menu items are then gradually added to the menus.

■ ■ Lines 111–149: Creating the Remaining Controls

This is where the buttons and the text input field are created. In addition, event handlers are bound to two events of the text input field to process the pressing of the key combinations <CTRL>+<C> (copy) and <CTRL>+<V> (paste).

■ ■ Lines 152–157: Preparing the Status Bar

We create a label as a yellowish colored status bar, on which the name of the currently opened file is displayed.

■ ■ Lines 160–174: Placing the Controls on the Interface

The controls are arranged on the user interface using the **grid** Geometry Manager, in such a way that they fully fill their respective “grid cells” (**sticky = 'news'**, i.e., *north east west south*).

■ ■ Lines 177–178: Event Loop

mainloop() starts event processing for our editor. This makes the user interface react to the user's input.

■ ■ Event Handlers (Lines 5–73)

- **new_press()**: Creates a file by first deleting the contents of the **ScrolledText** field; deletes from line 1 (line numbering starts at 1 for text widgets in **tkinter**), column 0 (column numbering starts at 0) to the end. Instead of the string **'end'** the constant **END** could have been used.
 - The global variable **filename**, to which we secure access with the statement **global filename** (otherwise a *local* variable with the same name would be created within **new_press()**!) is reinitialized with an empty string and the text of the status bar is updated.
- **open_press()**: Here we use the **askopenfilename()** function to retrieve the path to the file we want to open. After that, the file is opened in read mode with **open()** and the content read from it with **read()** is put into our text field before the file is closed again with **close()**.
- **save_as_press()** and **save_press()**: For save, we basically do the same as for open; however, the file is opened in write mode so that the **write()** method can write

the text on the editor that we get using the **get()** method of the **ScrolledText** widget. In **save_as_press()**, we ask for the filename using **tkinter**'s **asksaveasfilename()** function, and for **save_press()**, we save the file using the name we're already using. So, this can only work if a file has been opened or the content of the **ScrolledText** widget has already been saved to a file.

22

- **copy_press()** and **paste_press()**, **copy_press_key()** and **paste_press_key()**: When copying text, we first determine the contents of the text selection using the **selection_get()** method of the **ScrolledText** widget, then clear the clipboard using **clipboard_clear()**, and then insert the text to be copied to the clipboard using the **clipboard_append()** method.
- When inserting text, we first determine the current insert, i.e. cursor position in our **ScrolledText** widget with **index('insert')** and then get the clipboard content with **clipboard_get()** to insert it at this position.
- The **copy_press_key()** and **paste_press_key()** functions that we bound to the keystroke events (lines 148/149) call the event handlers we discussed earlier, but they are necessary because the event handlers bound to an event using **bind()** are passed an event object as an argument. These event handlers must therefore provide an argument, while the event handlers we bound to buttons and menu items using the **command** option must have no arguments.



Functions & Methods: How Do I Work with Program Functions to Work with Data and Trigger Actions?

Contents

- 23.1 Working with Functions – 328**
 - 23.1.1 Definition of Functions – 328
 - 23.1.2 Function Arguments – 329
 - 23.1.3 Return Values – 334
 - 23.1.4 Local and Global Variables – 335
- 23.2 Using Functions as Class Methods of Objects – 339**
- 23.3 Working with Modules and Packages – 342**
 - 23.3.1 Modularizing Program Code – 342
 - 23.3.2 Importing Elements from Modules – 344
 - 23.3.3 Using the Community: The Python Package Index (PyPI) – 345
- 23.4 Summary – 348**
- 23.5 Solutions to the Exercises – 349**

Overview

So far, we’ve covered how to get Python programs running, how to work with variables/objects, and how to input and output data. In this chapter, we’ll focus on what happens in between, namely how to process data. If input and output of data are the bun halves of our “program burger”, then we are now mainly concerned with the meat in the middle (although there are, of course, also many functions for input and output).

The main way to manipulate data in programs is to call functions that modify the data or trigger other actions. Functions are so important because we can use them to perform certain tasks, even without knowing exactly how that actually works in detail. We simply call the function, and the function does what it is supposed to do, without us having to program this functionality ourselves and without us having to understand the function’s inner workings. Of course, we can also define functions ourselves. Functions allow us to encapsulate a particular functionality and make it accessible from the outside. With functions, we ultimately offload pieces of code from the main program code and make them callable from anywhere.

In this chapter you will learn:

- How to program functions in Python and how to call functions (your own or those provided by others)
- how to combine functions (usually as methods of classes) into modules and packages and import them into your program when we want to use them
- how to work with the *Python Package Index (PyPI)* to include functionality provided by other developers in your program

23.1 Working with Functions

23.1.1 Definition of Functions

Remember our calculator application that we developed in the last chapter using the **tkinter** package (► Sect. 22.2.6)? There we defined some functions of our own, for example this one:

```
def delete_press():
    display['text'] = ''
```

This is a function that we don’t call ourselves, but an event handler function that is automatically called when the user clicks the “Clear” button to clear the current display. With this simple example, however, you can already see what a function is all about. Its definition begins with the statement **def**, followed by the name of the function. The name of the function is followed by round brackets that hold the function arguments. However, even if the function has no arguments at all (as in this case), the round brackets must be written in the definition (and later when the function is called).

The **def** statement, which is ultimately the *function head* of our function, is terminated with a colon, indicating that what now follows is the program code that is executed when the function is called, in short: What the function *does*. The function body is a code block that is identified, as always in Python, by the fact that it is indented. In this example, the code block consists of only one line of code. Incidentally, the code block must be at least one line. The program:

```
def my_function():  
  
    print('This is the main program.')
```

leads to an error message:

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

Apparently, an indented code block is expected after the **def** statement. Sometimes you already know that you need a function, you want to call it from the main program, but you don't know exactly how the function should work. What to do? Writing an “empty” function will result in the above error message, even if you write a comment as the only line in the code block; that is because the comment is not considered an executable statement that Python will accept as a code block. To solve this problem, Python has a special statement, namely **pass**. **pass** does absolutely nothing, but it is an executable statement. So, if you write **pass** as the only instruction in your function code, it will make your function syntactically correct without the function doing anything. Note that **pass** (just like **def**) is a Python *statement*, not a function, and is therefore not called with (empty, round) parentheses.

23.1.2 Function Arguments

Let us return to an example from the first part of the book (from ► Sect. 12.2.2), namely the temperature conversion from Kelvin to degrees Celsius. This conversion can be written as a function. Unlike the function `delete_press()` from above, however, our function needs an argument, namely the temperature to be converted:

```
def kelvin_to_celsius(kelvin):  
    print(kelvin, 'Kelvin are', round(kelvin - 273.15, 2),  
          'degrees Celsius.')
```

We could now call this function, for example with:

```
kelvin_to_celsius(300)
```

Then we received as output:

```
300 Kelvin are 26.85 degrees Celsius.
```

Of course, functions can have more than one argument. Consider the following example, in which we output a welcome message to the screen:

23

```
def welcome(name, greeting, message):
    print(greeting, ' ', name, '! ', message, sep = '')
```

We could now call the function as follows, for example:

```
welcome('Sophie', 'Welcome', 'Glad to have you with us.')
```

The result of this call looks like this in the (run) console:

```
Welcome, Sophie! Glad to have you with us.
```

■ Positional and Keyword Arguments

Which of the strings specified in the function call is assigned to the argument **name**, which to the argument **greeting**, and which to the argument **message**, is decided by the order in which they are passed, i.e., their position; these arguments are, as you already know, also referred to as *positional arguments*. However, we can specify the name of the argument when calling the function. This allows a call like the following:

```
welcome('Sophie', message = 'Glad to have you with us.',
        greeting = 'Welcome')
```

Here we pass the first argument by its position, but the following two by their names. Note that we pass the values for the **message** and **greet** arguments in a different order than the argument order from the function definition above. However, because we explicitly address the arguments by their names, Python can still correctly map the passed values to the function arguments. So, the advantage of these named arguments (also called *keyword arguments*) is that the order in which we specify the arguments doesn't matter, which is especially nice if the called function has a whole bunch of arguments. It would be tedious to have to look up the exact order of the arguments first.

Be careful, however, if you mix position and keyword arguments, as we did in the example above: Because then the position arguments must always be at the

beginning. So, you can't pass the first argument as a keyword argument and the second as a position argument.

By the way, we have already used a keyword argument in the definition of our function, namely when calling `print()`: Here we used the `sep` argument to specify that the individual strings should not be output separated by a space, which would otherwise have caused annoying white space in inappropriate places, for example before the exclamation mark that concludes the greeting.

■ Optional Arguments

The `sep` argument of the `print()` function just mentioned is an example of an *optional* argument that we *can* specify, but *don't have to* (in the Kelvin-to-Celsius conversion, for example, we called `print()` *without* the `sep` argument). `sep` has a default value, namely ' ' (i.e., a blank space), which is used whenever the `print()` function is called without explicitly specifying a value for `sep`. You can see this clearly when you call the help for `print()` in the Python console. There it says:

```
print(value, ..., sep=' ', end='\n', file=sys.stdout,
flush=False)
```

The optional argument `sep` is therefore “preallocated” with a default value, while the argument `value` is not an optional argument. If we were to refrain from specifying it in the function call, we would get an error message. Now, if we wanted to modify our `welcome()` function so that you don't have to specify a greeting, all we have to do is add the default value of the `greeting` argument in the function header:

```
def welcome(name, message, greeting = 'Welcome'):
    print(greeting, ' ', name, '! ', message, sep = '')
```

After that, we could also call our function like this:

```
welcome('Sophie', 'Nice to have you.')
```

Note that we have swapped the order of the `greeting` and `message` arguments in the function header. The reason is that in Python, arguments with default values must come last in the function definition. If multiple optional arguments follow, they must be passed as keyword arguments when the function is called, i.e., by specifying their names, otherwise Python will not know which of the passed values to assign to which optional arguments—after all, any of the optional arguments may or may not be provided with a value.

■ Indefinite Number of Arguments

For some functions, you don't know a priori with which arguments they will eventually be called. A good example of this is the `print()` function. It can take a whole

set of variables or values and print them all on the screen. Each time **print()** is called, the number of arguments may be different.

Suppose we wanted to extend our **welcome()** function so that it welcomes not only one, but several people. We want the number of people to be variable. We achieve this by using a *tuple* argument that is marked with an asterisk in front of the identifier:

23

```
def welcome(*name, message, greeting = 'Welcome'):
    print(greeting, ', ', end = '', sep = '')
    print(*name, end = '', sep = ', ')
    print('!', message)
```

We can now call this function with several names, for example like this:

```
welcome('Sophie', 'Marc', 'Celine',
        message = 'Glad you're here.')
```

The output then reads:

```
Welcome, Sophie, Marc, Celine! Glad you're here.
```

Two things are noteworthy about our feature:

1. Compared to the previous version of **welcome()** we work here with several print statements. This is necessary to achieve the correct positioning of commas and spaces. By setting the (optional) argument **end** of the **print()** function to an empty string (default value is **'\\n'**, i.e. newline) we ensure that we always continue writing on the same line despite several successive **print()** calls.
2. The argument **name** now collects the “loose” arguments, i.e., the names of the persons we specify in the function call, and packs them into a tuple with the identifier **name**. We can then work with this tuple in the function body. In our example, we “unpack” the tuple (using the asterisk again) and output its elements using **print()**. If we had omitted the asterisk in the call to **print()**, we would have simply output the tuple en bloc, which is not as visually appealing (try it out!).

An alternative to this approach would have been to pass only one argument when calling for the names of the people to be greeted, namely a list, which we would then have had to process accordingly in our function. With such an adapted function, the function call would then look like this:

```
welcome(['Sophie', 'Marc', 'Celine'],
        message = 'Glad you're here.')
```

Note that in this case the three names represent only *one* argument, namely our list of names, whereas in the previous call to the function we passed *three different* name arguments, namely 'Sophie', 'Marc' and 'Celine', which Python just conveniently collects for us and puts into a tuple that we can then work with. This way of calling the function is a bit more 'natural' and intuitive, and therefore preferable to the list solution. If a function uses such an undefined tuple argument, all *subsequent* arguments must be called as keyword arguments, i.e., with their identifier; this is obvious, because how else is Python supposed to distinguish whether a passed value still belongs to the tuple argument or already belongs to the next argument.

By the way, every now and then you will see functions with arguments prefixed with a double asterisk (**); the **tkinter** function **config()**, which we can use to set the options of **tkinter** widgets (see ► Sect. 22.2.3.1) is one such function (check the help!). Such arguments are also collection arguments, but for keyword arguments. In this way, different arguments passed as keyword arguments can be collected and put into a dictionary whose keys are the argument names and whose values are the values passed for those arguments. Unlike the collection arguments for non-named arguments (i.e., the “* arguments”), a collection argument for keyword arguments must *always* be placed *at the end* of the argument list in the function definition.

■ Data Types of Function Arguments: Type Hint Annotations (Function Annotations)

You have probably already noticed that the arguments of functions are *not typed*, that is, the data types of the arguments are not specified. Accordingly, Python cannot check whether a function is called with arguments of the correct type. When we wrote our **welcome()** function, we naturally assumed that the arguments **name**, **greeting**, and **message** are all **strings**, i.e. arguments of type **str**. But this does not have to be the case at all! A user of our function who is not aware of the exact meaning of the individual arguments could, for example, specify a boolean value as the argument **greeting**, expecting it to be an option that determines whether a greeting message is displayed or not.

To counter this problem, Python has a concept called *type hints* (also known as *function annotations*). They are built into the function definition and specify the expected type of the argument. For our original **welcome()** function, this might look like this:

```
def welcome(name: str, message: str,
            greeting: str = 'Welcome'):
    print(greeting, ' ', name, '! ', message, sep = '')
```

As you can see, each argument identifier is followed by a colon, followed by the type expected for that argument.

Now, it is by no means the case that Python automatically checks whether the user really adheres to the type default. The default is only indicative. Nevertheless, it is useful for two reasons: First, the type hints are also included in the automati-

cally generated help for the function and are visible there for every user of the function. Second, there are development tools that evaluate these type hints. One of these tools is *PyCharm*. For example, if you try to call our **welcome()** function with a boolean value for the **greeting** argument:

```
welcome('Sophie', 'Glad you could join us.', True)
```

23

PyCharm highlights the boolean value in the code editor window. If you hover over it, a small popup window will inform you of the error with the message **Expected type 'str', got 'bool' instead**.

23.1.3 Return Values

Our function **kelvin_to_celsius()** from the last section was passed a Kelvin temperature value as an argument, converted it to degrees Celsius and output the result on the screen. Of course, we could also forgo the output and simply *return* the calculated Celsius value instead. In this case, it is a function with a *return value*.

The return is accomplished with the **return** statement.

```
def kelvin_to_celsius(kelvin: float):
    return kelvin - 273.15
```

We can now call this function from our main program, store its result in a variable first and then output it to the (run) console:

```
temp = kelvin_to_celsius(290)
print(temp)
```

We have added a type hint to the argument **kelvin** in the function definition. We can do the same with the return value:

```
def kelvin_to_celsius(kelvin: float) -> float:
    return kelvin - 273.15
```

To do this, the type of the return value is written after the actual head of the function with an arrow **->**; the colon again introduces the following code block, i.e., the function body (note: we had used the colon for type hints in the function arguments).

Sometimes you will want to return more than one return value. In that case, it's a good idea to wrap the different elements of the return value in a *tuple*. In the following, our **kelvin_to_celsius()** function is modified to return both the calculated Celsius and the original Kelvin value, as two elements of a tuple:

```
def kelvin_to_celsius(kelvin: float) -> float:
    return (kelvin, kelvin - 273.15)
```

The brackets around the two values can also be omitted and are only written here to make clear that a tuple is created. In fact, the **return** statement automatically creates a tuple if it is followed by several objects separated by commas. The elements of the tuple can be easily accessed after the function call, either by indexing or by “unpacking” the tuple directly in the assignment:

```
temp = kelvin_to_celsius(290)
print(temp[1])

cel, kel = kelvin_to_celsius(290)
print(kel)
```

The **return** statement automatically exits the function. Therefore, **return** should always be the last instruction in a function. All code after it would not be executed anyway.

23.1.4 Local and Global Variables

Consider the following program:

```
greet = 'Good morning'

def welcome(name, message, greeting):
    greet = greeting + ', ' + name + '! ' + message
    print(greet)

welcome('Sophie', 'Nice to have you with us!', 'Hello')
print(greet)
```

Think about what this code will output in the (run) console. Do you have an idea? Then try it out in Python. Was your expectation correct?

If you start the program, the following is output:

```
Hello, Sophie! Nice to have you with us!
Good morning
```

The interesting question now is why the variable **greet**, when we **print()** it, still has the value **'Good morning'**, which we assigned to it at the beginning of the program.

Before we print its value, we call the function `welcome()`, which changes the value of `greet`, in our example to **"Hello, Sophie! Nice to have you with us!"**. Shouldn't `greet` contain this value at the end of the program?

The solution to the puzzle is that the variable `greet` in our main program and the variable `greet` in the `welcome()` function are ultimately *two different* variables. The variable `greet`, which we create in the function body of `welcome()`, only exists within this code block, its *scope* is limited to the function `welcome()`. However, whenever we access the variable `greet` within the function body of `welcome()`, we are working with the variable created in this code block, not the variable we defined in the main program. So you could say that the variable `greet` in our `welcome()` function “hides” the variable of the same name in the main program. We cannot get to the variable of the same name in the main program from within the function; the variable defined in the function is “in the way”, so to speak.

However, there is a way to access the variable of the main program. Consider the following, slightly adapted code:

```
greet = 'Good morning'

def welcome(name, message, greeting):
    global greet
    greet = greeting + ', ' + name + '! ' + message
    print(greeting)

welcome('Sophie', 'Nice to have you with us!', 'Hello')
print(greet)
```

As you can see, we have only added the `global greeting` statement to the `welcome()` function. The effect of this is that Python does not create a new variable that is only valid within the `welcome()` function—this scope is also referred to as the function's *namespace*. Instead, the global namespace, i.e., the namespace of our main program, is searched for a variable with this name; if one is found, it is used, otherwise an extra variable is created in the smaller namespace of the function, which ceases to exist as soon as the function is exited (try it out and change all occurrences of `greet` to a different identifier within the function).

You might now argue that the situation in our example is somewhat artificial and that its problems are ultimately only due to the fact that variables with the same name are used in the definition of the function and in the main program. Imagine, however, that you wanted to change a *global* status variable from within the function, i.e., a variable defined in the namespace of the main program, for example, a variable that indicates whether the document currently being processed has already been saved or not. This status variable should, of course, exist independently of the function and still be available when your function has long since been exited and all *local* variables, i.e., those defined in the function itself (more precisely: in its namespace), have long since been deleted again. If you access the presumable global status variable without a `global` statement in your function, a local variable of the same name is created in the namespace of the function, which dis-

appears when the function has been completely executed. The global status variable remains unaffected. Only by using the **global** statement do you make it clear to Python that you are working with the global variable and do not want to create a new local variable inside your function. Your function then changes its environment (in the form of the status variable). Such *side effects* are generally avoided to make the function more independent of the code that calls it.

? 23.1 [20 min]

Write a function called **create_website()** that takes as arguments a title, a heading, and a text (each as strings), creates an HTML document (i.e., ultimately a simple web page) from them, and saves it in a file called **website.html**. Document your function as much as possible.

An HTML document has the following basic structure (the indentations are only for better clarification of the structure; they do not have to be written into the file):

```
<html>
  <head>
    <title>Here is the title of the website</title>
  </head>
  <body>
    <h1>Here is a headline</h1>
    <p>Here is a text</p>
  </body>
</html>
```

Ultimately, the HTML document is a collection of elements such as **h1** (for header 1) or **p** for paragraph. The beginning and end of the elements are marked by tags, whereby the end tags correspond to the beginning tags, supplemented by a leading slash. Between the start and end tags is the content, whereby HTML elements can also contain other HTML elements (for example, the enclosing **html** element contains the elements **head** and **body**, which in turn can contain other elements).

The function **create_website()** is supposed to create such a HTML document, “build in” the information passed as arguments and save the whole thing as **website.html**. After that you can view this file with your web browser!

? 23.2 [10 min]

The following two functions have errors. Rewrite the functions so that they are syntactically correct and serve their purpose:

(a)

```
from random import *

def dice() -> int:
    ''' Generates a random number analogous to a dice roll '''

    dice_result = randint(1,6)
```

(b)

```
def create_phonenumber(country: str, areacode: str) : -> str
    '''Creates a formatted phone number based on its
    components; country is the country's ISO code'''

    countries_dict = {'US': '1', 'DE': '49', 'FR': '33',
                      'AT': '43', 'NL': '31', 'BE': '32',
                      'PL': '48', 'DK': '45', 'CZ': '42'}
    return '+' + countries_dict[country] + areacode + number
```

23

? 23.3 [5 min]

Develop a function **deal_with_myself()** that deals only with itself, taking a variable number of named (i.e., keyword) arguments and printing first their names and then their values to the screen.

? 23.4 [5 min]

What output does the following program produce, and why?

```
from datetime import *

total_sales = 0.00
last_sale = 0.00

def new_sale(amount, article):
    global total_sales
    last_sale = amount
    total_sales = total_sales + amount
    currenttime = datetime.now()
    print(currenttime.strftime('%Y-%m-%d %H:%M:%S') +
          ' -- New sale: ' + str(round(amount, 2)) +
          ' dollars with article number ' +
          article + '.')

new_sale(10.99, 'US07011981')
new_sale(24.99, 'CA25101878')

print('Total sales: ' + str(total_sales))
print('Last sale: ' + str(last_sale))
```


23.2 Using Functions as Class Methods of Objects

In many places we have already worked with functions that are part of classes as a matter of course. As you know, such functions are also called *methods* (if you are no longer familiar with this, it is best to go back and review ► Sect. 11.7.4).

For example, when we set the arrangement of the controls of our **tkinter** user interface with **grid()**, we had call a method like **mybutton.grid(row = 1, column = 4, sticky = 'news')** (see the calculator example in ► Sect. 22.2.6). In this case, we call the method of the **mybutton** object, which is an instance of the **tkinter** class **Button**. To let Python know exactly which object's methods we want to call, we attach the function call to the identifier of the object in question using the dot operator. So in general, method calls have the form *object.method(...)*.

Since methods are nothing more than normal functions that we can use to work with the object they belong to, we don't really need to discuss them separately here. However, we should take a brief look at three special topics in connection with methods, namely:

- How methods are *defined* as part of classes
- What special role *constructor methods* play in this
- What useful *standard methods* classes in Python have that we can adapt for our purposes

■ Defining Methods as Part of Classes

In ► Sect. 21.7, we had defined a class **Product** as follows.

```
class Product:
    name = ''
    description = ''
    item_number = ''
    manufacturer = ''
    price = 0.0
```

This class consists solely of properties/attributes. Suppose we now wanted to give our class a special **show()** method that displays the properties of a product in a nice way. To do this, we would simply have to extend the code block of the class with the definition of the method:

```
class Product:
    name = ''
    description = ''
    item_number = ''
    manufacturer = ''
    price = 0.0
```

```
def show(self):
    print('Product:', self.name,
          '\nDescription:', self.description,
          '\nItemnumber:', self.item_number,
          '\nManufacturer:', self.manufacturer,
          '\nPrice:', self.price, '\n')
```

23

As you can see, the function definition header is indented the same way as the attributes. The **show()** function is thus part of the class definition of **Product**. For all objects of type **Product**, the method can be called from now on. For example, we could define the following product:

```
p = Product()
p.price = 10.99
p.name = 'Garden shovel'.
```

To display the product properties, we can then conveniently call our self-defined method:

```
p.show()
```

This results in a cleanly structured output:

```
Product: Garden shovel
Description:
Article number:
Manufacturer:
Price: 10.99
```

The properties to which we have not explicitly assigned values (such as **description**) are displayed with their default values (i.e. “empty” string).

You may have noticed the **self** argument in the definition of our **show()** method. **self** always represents the object for which the method is called. This way, we can conveniently access the properties (and possibly other methods) of the current class instance for which our method is called. This argument does not necessarily have to be called **self** (it just has to be first in the argument list), but it is good practice to use the easily understandable identifier **self**.

Methods, like properties, are also inherited. In ► Sect. 21.7.2 we derived a class **Book** from the class **Product**, i.e., a special type of product. This class inherited all the properties of the parent class, and also had additional properties that are only relevant for books, such as the number of pages. Inheritance of methods works in a similar way.

There is also the possibility to *overload* methods. This means that the more special class (in our case **Book**) has its own **show()** method, which perhaps also dis-

plays the special properties of books like author and number of pages. Thus, both the parent class and the class derived from it now each have a method **show()**.

When we call the **show()** method for an object of type **Book**, Python first looks to see if that *class itself* has a corresponding method; if so, it executes it. However, if **Book** itself does not have a **show()** method, it checks whether the next higher class in the class hierarchy, i.e. the parent class **Product**, has such a method. In this way, it is possible to provide classes at different levels of the class hierarchy with methods of the same name, but with behavior specific to that class. The user can simply call the same method every time and does not have to deal with the specifics of the different classes. A great strength of object-oriented programming!

■ The Special Role of the Constructor Method

In the example above, we had created an instance of the **Product** class using its default constructor **Product()**. This simply returns a new object of that class. However, we can alternatively define our *own* constructor, perhaps taking some arguments, such as the name of the product and its price as the two most important properties. The constructor would then assign the values passed as arguments to the corresponding attributes and return the object defined in this way.

We create our own constructor by adapting the default **__init__()** method (two underscores each!) So, we would add the following method definition to our class definition:

```
def __init__(self, name, price):
    self.name = name
    self.price = price
```

The constructor function takes **self** as the first argument, in this case the object that is created by it. We have defined the further arguments freely. With this, we could now also create the previously used product with the name "**garden shovel**" at a price of **10.99** as follows (**self** does not have to be specified in the call, because, fortunately, Python takes care of that itself):

```
p = Product('garden shovel', 10.99)
```

Note that we are not calling the **__init__()** constructor function by its name at all, but the constructor of the class, whose identifier is identical to that of the class. In the background, however, Python then calls the **__init__()** method, either the default version, or, if we have overloaded it, our own variant. Normally, you won't call **__init__()** yourself unless you want to call the constructor of the parent class from within the constructor of a derived class.

■ Two Useful Standard Methods of Classes

If you create an object of our class **Product** in the Python console (you have to execute the definition of the class in the console first!) and then enter the name of the object, for example **p**, you get a rather unattractive output:

```
> p
<__main__.Product at 0x1ff91106048>
```

Fortunately, this can be changed, and we can do so by overloading the default `__repr__()` method. This method is called whenever the user enters the identifier of an object in the console. It returns the string to be displayed in the console.

We could overload the method by adding a corresponding method definition to our **Product** class definition, for example:

```
def __repr__(self):
    return 'Product: ' + self.name + '\nPrice: ' +
        str(self.price)
```

If we now enter the identifier of our object into the console, we get a nicer output (note: after adjusting the class definition, you need to create a *new* object of this class **Product** so that it has the new method `__repr__()`):

```
> p
Product: Garden shovel
Price: 10.99
```

Similarly, we can determine what should happen when the user calls the `print()` function and tries to display our object. `print()` automatically calls the `__str__()` method in the background and outputs its return value. Thus, in the definition of our **Product** class, we could overload the `__str__()` method as follows:

```
def __str__(self):
    return "Product '" + self.name + "' ($" + str(self.price)
```

Then we can call the `print()` method with our object **p** as an argument and get a prettier representation:

```
> print(p)
Product "Garden shovel" ($10.99)
```

23.3 Working with Modules and Packages

23.3.1 Modularizing Program Code

Python allows you to offload code to other files for better reuse. For example, you could combine functions or entire classes that you have developed and want to use in different programs into one Python file and then access them from other pro-

Name	Date modified	Type	Size
__pycache__	8/22/2020 11:14 PM	File folder	
test	8/22/2020 11:14 PM	File folder	
__init__.py	7/20/2020 4:03 PM	JetBrains PyChar...	170 KB
__main__.py	7/20/2020 4:03 PM	JetBrains PyChar...	1 KB
colorchooser.py	7/20/2020 4:03 PM	JetBrains PyChar...	2 KB
commondialog.py	7/20/2020 4:03 PM	JetBrains PyChar...	2 KB
constants.py	7/20/2020 4:03 PM	JetBrains PyChar...	2 KB
dialog.py	7/20/2020 4:03 PM	JetBrains PyChar...	2 KB
dnd.py	7/20/2020 4:03 PM	JetBrains PyChar...	12 KB
filedialog.py	7/20/2020 4:03 PM	JetBrains PyChar...	15 KB
font.py	7/20/2020 4:03 PM	JetBrains PyChar...	7 KB
messagebox.py	7/20/2020 4:03 PM	JetBrains PyChar...	4 KB
scrolledtext.py	7/20/2020 4:03 PM	JetBrains PyChar...	2 KB
simplifiedialog.py	7/20/2020 4:03 PM	JetBrains PyChar...	12 KB
tix.py	7/20/2020 4:03 PM	JetBrains PyChar...	78 KB
ttk.py	7/20/2020 4:03 PM	JetBrains PyChar...	58 KB

■ Fig. 23.1 Directory structure of the Python module **tkinter**

grams. Such files, which hold outsourced program code, are called *modules*. A module is therefore nothing more than program code that has been combined into its own Python file for reuse.

Several modules with related content can be combined into a *package*. While a module is technically nothing more than a Python (**.py**) *file*, a package is a *directory* containing *several* modules, i.e., several **.py** files. For Python to know that this directory should be a package, there must also be a file named **__init__.py** (double underscores!) in the directory. This file may be empty, it just tells Python that this directory should be considered a package. Of course, **__init__.py** may also contain code itself. A good example of this is the package **tkinter**, of which we have already made extensive use. ■ Figure 23.1 shows the directory structure of this package.

Here you can also see that the file **__init__.py** is pretty large, so it is by no means just an empty shell but contains quite a lot of code.

23.3.2 Importing Elements from Modules

After we have seen that modules and packages allow us to extract program code from a program, the question naturally arises as to exactly how we can access the extracted program code, i.e., the classes and functions of module. After all, their program code is no longer in our main program file, so it has to be made “available” somehow. This making available is called *importing* and is something we have already done several times without discussing it in detail.

■ Import Selected Classes

In Python, import statements look like this:

```
from tkinter import Tk, Button, Label
from tkinter.font import Font
```

You can see that these follow the structure:

```
from modulname_or_package_name import classlist
```

For example, the first statement imports the classes **Tk**, **Button**, and **Label** from the **tkinter** module. The second statement imports the class **Font** from the module **font**, which belongs to the package **tkinter** (therefore **tkinter.font**).

If you take another look at the directory structure of the **tkinter** package (▣ Fig. 23.1), you will see that there is indeed a file **font.py**, the module **font**, from which we import the class **Font** with the second statement. On your hard disk in the Python installation, find the path **\Lib\tkinter**, and open the file **font.py**. In it you will find, among other things, a definition of the class **Font** that we import.

Now, what about the first import statement? Here we import three classes, apparently directly from the package **tkinter**. But which file contains these three classes? You have probably already guessed: These classes are in the **__init__.py** file, which you can easily see for yourself by opening this file. After we have imported the classes with an **import** statement, we can use them in our program, without further ado, simply by using their identifier, for example by calling the constructor method of the **Tk** class: **win = Tk()**.

■ Import the Entire Contents of a Module

Instead of explicitly specifying the list of classes we want to import, we could have used a *wildcard* and just imported *all* classes that way:

```
from tkinter import *
```

This is exactly how we did it for simplicity in the last chapter. However, this approach is frowned upon among Python programmers because you don’t know exactly what you are importing, and this may lead to naming conflicts with other

classes that you have already used in your code. The controlled approach with explicit specification of the classes to be imported is therefore usually preferred.

■ Import the Entire Module

Another way of importing is to simply import the entire module:

```
import tkinter as tk
```

The last part of the statement, **as tk**, can also be omitted. However, using **as** makes it easier to access the module, especially if the module name is long. This is because with this type of import, the module name *must always* be included when accessing the module's classes, for example:

```
clear_button = tkinter.Button(win,  
    text = 'Clear',  
    command = clear_press)
```

When using renaming with the help of **as**, this shortens to:

```
clear_button = tk.Button(win,  
    text = 'Clear',  
    command = clear_press)
```

as, by the way, can also be used to “rename” individual classes imported with **from module import class**; this is useful, especially to avoid naming conflicts with existing classes (possibly developed by yourself!)

23.3.3 Using the Community: The Python Package Index (PyPI)

Python comes with a number of out of the box modules and packages, including the already used package **tkinter**. In addition to the packages installed by default, the *Python Package Index (PyPI)* at ► <https://pypi.org/> offers a variety of packages for almost every imaginable task. Each package has its own page with some important information about the package, such as the author, the license under which it is provided, or the Python version needed to use the package. Generally, a short description of the package is also provided, which is important for deciding whether a package that sounds good by name serves the desired purpose. However, the descriptions on the *PyPI* page of the package are often quite poor, and so it is good that some packages have their own home page linked from the *PyPI* page which then provides more detailed information about the package. An example of such a *PyPI* page is ► <https://pypi.org/project/numpy/>, the page of the well-known package **NumPy**, which adds data types to Python for efficient work with multidimensional arrays.

The *Python Package Index* offers an incredible number of packages. However, since the packages vary greatly in scope and quality, it is advisable not to simply “search” the *Python Package Index*, but to first find interesting “package candidates” on the Internet in the relevant discussion forums, on blogs and other websites and then try them out. If necessary, simply install several packages for the same purpose.

23

■ Installing Packages with PyCharm

Once you have found a package you want to use, you must install the package before you can use it. In *PyCharm*, you can do this via the graphical user interface by going to the *Settings* dialogue (click the gear button in the title bar of the *PyCharm* application window) then *Project | Project Interpreter*. Here you will see a list of packages that are currently available. By clicking on the small plus button, you will get to a dialog where you can select and install a package from the *Python Package Index*.

■ Installing Packages with pip

If you prefer to work via the console of the operating system rather than via the graphical user interface of *PyCharm*, you can use **pip**, a program that allows the administration of packages. By the way, **pip** is a recursive abbreviation, it contains itself in the long form: *pip installs packages*.

If you want to work with **pip**, you must first make sure that it is installed. To do this, change to the **scripts** directory of your Python installation. If you are working with Microsoft Windows and have installed Python in **C:\python37**, **pip** has the path **C:\python37\Scripts\pip.exe**. If **pip** is not there, change to the directory where **python.exe** is located (typically **C:\python37** in our example) and execute the following statement:

```
python -m ensurepip --default-pip
```

pip checks itself whether it is up to date. If this is not the case, you can easily update to the current version:

```
python -m pip install --upgrade pip
```

After that, you install packages with **pip** by executing the **pip install *packagename*** statement, for example:

```
pip install NumPy
```

But **pip** can do much more. You can display information about a package with **pip show**, for example for **NumPy**:


```
pip show NumPy
```

With **pip** `deinstall packagename` you can also uninstall a package, with **pip** `search` you can search the *Python Package Index* directly from the command line, for example:

```
pip search webscraping
```

For help with **pip** and its many options, use the command

```
pip -h
```

or

```
pip -help
```

■ Virtual Environments

Python can install the package into a *virtual environment* if needed. Then the package is not added to the general package library but installed in a separate library for your current project. This allows you to work with different versions of the same package in different projects which can be important if, for example, your project requires an older version of a package because it cannot run with the current version. With a *virtual environment*, you keep your project running while you can work with the current version of the package elsewhere. This is made possible by the fact that you can also specify the version to be installed when you install packages, so that you do not necessarily have to use the latest version.

By the way, the same is true for Python itself. You can specify which Python interpreter you want to work with. With Python version 3.x, some significant changes have been made to the language definition compared to the older 2.x versions, and Python projects developed under Python 2.x are not necessarily fully executable under version 3.x. In *PyCharm*, you can easily specify which should be the *Project Interpreter*, you want to use in the current project. This way, an older project developed under Python 2.x will still be executable without the need for extensive rebuilding. When you create a virtual environment, the interpreter you want to work with is also copied into the virtual environment.

But of course, you can also select a project interpreter without creating a virtual environment. So, if you simply want to work with the Python 2.x interpreter and the packages installed for that interpreter (in its “main” installation), you can simply switch the project interpreter in *PyCharm* to that interpreter.

In any case, it is recommended to refrain from creating a virtual environment right from the start of your project, unless you absolutely need one (which, as a rule, should not be the case).

23.4 Summary

23

In this chapter, we saw how functions are defined and used in Python; we also looked at how modules and packages work and learned about the *Python Package Index (PyPI)* as an important source of useful program code.

Be sure to take the following points from this chapter:

- Functions are defined in Python with the **def** statement and consist of the function header, containing the function's identifier and arguments, and the function body, the (indented) block of code that is executed when the function is called.
- Optional arguments are assigned their default value in the function header of the function definition (***argument = default_value***).
- The function arguments are in the function definition without data type, but can be provided with a type hint of the form: ***datatype***, which is not binding, but is processed by many IDEs and is also displayed in the help for the function; in addition to function arguments, the return values of the function can also be provided with type hints of the form ***→ datatype***.
- Return values are returned with the **return** keyword.
- Variables defined within functions, like function arguments, are local variables and therefore can only be used within the code block of the function; if you want to access a global variable from within a function, you must use the keyword **global**.
- Even if a function has no arguments, the round (but in this case empty) argument brackets must be written when it is called (as well as when it is defined).
- When calling a function, the arguments can also be passed as keyword arguments, that is, with their names (in the form ***argument = value***); then the order of the arguments does not matter.
- Python code can be grouped into modules, and multiple modules can be grouped into packages.
- Classes from modules used in the program must first be imported, either by explicitly specifying the classes to be imported in the form ***from module_name_or_package_name import classlist*** (recommended procedure) or by importing *all* classes in the form ***from module_name_or_package_name import ****; also, the module can be imported with a statement of the form ***import module_name_or_package_name***.
- The most important source for Python modules is the *Python Package Index (PyPI)*; here you can find solutions for many different programming tasks. Researching for a suitable package is always worthwhile before you start programming the functionality you are looking for yourself.
- The installation of modules of the *PyPI* is done either with the help of the command line program **pip** or via the an IDE like *PyCharm*.

23.5 Solutions to the Exercises

■ Exercise 23.1

The function `create_website()` could look like this:

```
def create_website(title: str, header: str, text: str):
    ''' Creates a simple website and saves it as
        website.html.

    Arguments:
    -- title: Title of the website
    -- header: Header text
    -- text: The actual content'''

    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()
```

A possible call would then be:

```
create_website(My first Python website', 'Section 1',
    'Here could be a longer text')
```

The function is called with three string arguments for title, heading and text content of the website to be created. It does not have a return value. Instead, it generates the HTML code of the website as string variable **html_content** and then writes it to the file **website.html**. Instead of first creating a large string with the file content and then writing it to the file, it would also have been possible to use several **write()** statements to write the file content step by step to the file without already completely assembling it at the beginning. If you now call the finished function, you can open the website it generates in your web browser afterwards.

In addition to the actual function, however, the task also required documentation. For this purpose, we first work with a docstring. It briefly describes what the function does and what its arguments mean. If you copy the function to the Python console and execute it there, you can then use **help(create_website)** to view the function's help, which is fed from exactly this docstring. For further documentation, we have included type hints for the arguments in the function header. A commenting of the code in the function body was omitted because the code is quite simple.

■ Exercise 23.2

- (a) The function **dice()** generates a random number with the help of the function **randint()** imported from the module **random**. This random number is stored in the integer variable **dice_result**. However, we forgot to return the result with **return**. If you call this function and catch its return value in a variable or output it to the console, you will notice that you get the special value **None** every time. It indicates that the function returns no value. However, if you now add the missing **return** statement, you will get back a random integer value between 1 and 6, just like rolling dice.
- (b) The function **create_phonenumber()** suffers from two problems: First, the colon in the function header is placed *before* the type hint, although it should actually be placed at the end of the function header after all, it introduces the following code block, i.e., the function body. Secondly, the argument **number** is used in the body of the function, as the actual subscriber line number without prefixes. However, it does not appear at all in the argument list in the function header. Of course, the code of the function cannot access an argument that is not passed to the function. After these corrections, the function can then be called to generate a nicely formatted phone number, for example in this call:

```
print(create_phone_number('DE', '171', '3456789'))
```

The country code **'DE'** is translated into the corresponding country code with the help of a dictionary.

■ Exercise 23.3

The function **deal_with_myself()** could look like this:

```
def deal_with_myself(**args):
    print(list(args.keys()))
    print(list(args.values()))
```

As you recall, a principally indefinite list of named, or keyword, arguments can be accessed with an argument prefixed with two asterisks in the function header. This argument is then a dictionary with the names of the arguments as keys and the argument values passed as values of the dictionary entry. Accordingly, we can use the **keys()** and **values()** methods to get the keys, that is, the argument names and their values. A call to this function might then look like this:

```
deal_with_myself(first_argument = 'An str argument',
    still_an_argument2 = 5)
```

■ Exercise 23.4

This program, which “books” new sales, generates the following output:

```
24.08.2019 12:48:24 - New sale: 10.99 dollars with article
number      US07011981.
24.08.2019 12:48:24 -- New sale: 24.99 dollars with article
number      CA25101878.
Total sales: 35.98
Last sale: 0.0
```

The first two outputs are generated directly by the **new_sale()** function. The following two outputs, namely on the total turnover and the turnover of the last sale, are simply outputs of the variables **total_sales** and **last_sale**. These two variables are created in the main program and initially default to the value **0.00**. The function then attempts to change these variables, which it seems to succeed in doing in the case of **total_sales**. To the current value of **new_sale()**, it adds the value of each newly posted sale. The sum of our two sales is indeed 35.98. But what about **last_sale**? Although this variable is also assigned a new value in the code of the function **new_sale()**, it still has the value at the end with which it was initialized at the beginning of the program. What happened? The two variables, the one initialized at the beginning and the one used in the function body of **new_sale()**, are two different objects. The variable used in the function body is a *local* variable that ceases to exist at the end of the function. Value assignments to this variable have no effect on the variable initialized at the beginning of the program, outside the function. If we wanted to change this variable instead, we would have to tell Python to do so with a **global** statement, as we did for **total_sales**. This way Python knows that we don't want to create a local variable but write to the global variable of the same name that was created outside the function.



Conditional Statements & Event Handling: How Do I Control the Program Flow and Make the Program React to User Actions and Other Events?

Contents

- 24.1 If-Else Constructs – 354**
 - 24.1.1 Simple If-Else Constructs – 354
 - 24.1.2 Nested if-else Constructs – 356
 - 24.1.3 If-Else Construct with Composite Conditions – 358
 - 24.1.4 If-Else Construct with Alternative Conditions (elif) – 359
- 24.2 Events – 361**
- 24.3 Summary – 362**
- 24.4 Solutions to the Exercises – 362**

Overview

Next, we will look at how to branch in Python programs and—depending on the situation—sometimes execute one part of the code, sometimes another. This way, our program flow becomes really interesting and is not just the execution of exactly the same sequence of Python statements over and over again.

In this chapter you will learn:

- How to branch in the program code with If-Else constructs
- how If-Else constructs are nested within each other
- how multiple alternative conditions can be considered with the `elif` statement
- how conditions for If-Else constructs are built, and which comparison operators can be used for this purpose
- how to combine several conditions into one overall condition using logical operators
- how to make your program react to events

24.1 If-Else Constructs

24.1.1 Simple If-Else Constructs

Let's review our conversion of temperatures between the units Kelvin and degrees Celsius. In the previous chapter (► Sect. 23.1.3) we defined the following function:

```
def kelvin_to_celsius(kelvin: float) -> float:
    return kelvin - 273.15
```

Suppose now that we wanted to improve this simple function by making it less prone to erroneous input. A temperature in Kelvin, as you recall, can never be negative. Absolute zero is 0 K, which is -273.15°C . Lower temperatures are physically impossible. At 0 K, there is simply no more heat at all, and accordingly it cannot get any colder. Thus, if we want to prevent our function from returning an invalid value because the function argument was invalid, we need to check whether the function argument **kelvin** is greater than or equal to 0. If **kelvin** is less than 0, i.e., illegal, our function should respond, it could do so by displaying an error message on the console, or it could return a special error code that the programmer calling our function can use to check whether the conversion was successful. We will take the latter route, which is preferable because it allows the programmer to react to an error the way they want to, whereas just displaying our error message does not give the user any option other than starting the program again.

Now, to make our function robust to incorrect input, we need to add an If-Else construct that checks if the **kelvin** value is invalid, i.e., less than 0, and in that case return an error indicator, for example `None`. The adapted function might then look like this:

```
def kelvin_to_celsius(kelvin: float) -> float:
    if kelvin < 0:
        res = None
    else:
        res = kelvin - 273.15
    return res
```

You can see that we have built in a program hint here: The keyword `if` is followed by the condition to be checked, in our case whether the Kelvin temperature is less than 0. If the condition is met, the following code block is executed, which—like the code blocks for functions—begins after a colon and is indented. In our example, the code block contains only a single statement, namely an assignment in which we assign `None` to an intermediate variable called **res**, but of course there could be any number of further statements here. This is followed—at the same indentation level as **if**—by the **else** keyword, again followed by a block of code that is executed only if the **if** condition is not met. The **else** branch is, in a sense, the logical inverse of the **if** condition; whenever the **if** condition is not met, the program jumps directly to **else** without executing the program code in the **if** code block. In our example, moving to the **else** code block means that our kelvin argument is valid and can thus be converted to a Celsius value.

The **return** statement after the If-Else construct is executed in any case, no matter whether the program branches into the **if** or **else** branch. In this way, we ensure that the function value stored in **res** is always returned, which—depending on whether the **if** or **else** branch has been traversed—contains the converted Celsius value or the error indicator `None`.

In our main program we could now integrate this function, for example, like this:

```
kelv = input('Please enter a temperature in Kelvin: ')
cel = kelvin_to_celsius(float(kelv))
if cel is None:
    print('You have entered an invalid Kelvin temperature!')
else:
    print(round(float(kelv), 2), 'Kelvin are', round(cel, 2),
          'degrees celsius.')
```

As you can see, we ask the user for a temperature in Kelvin, convert it, and then look to see if there was an error; depending on whether there was, we either output an error message or the converted value in Celsius.

You will have noticed the double equal sign that we use to compare the variable **cel** with the error indicator `None`. Python, like many other languages, distinguishes between the *assignment operator* `=` and the *comparison operator* `==`. The mathematical inequality sign ($\neq$) is equivalent in Python to the comparison operator `!=`, i.e. “not equal”, because the exclamation mark is, as in many other programming languages, the logical operator *NOT*, which turns the truth of a statement around.

In this case the statement that the two values to the left and right of the equal sign operator are equal.

The **else** branch in an If-Else construct can also be omitted. So our function **kelvin_to_celsius()** could also look like this:

```
def kelvin_to_celsius(kelvin: float) -> float:
    res = None
    if kelvin >= 0:
        res = kelvin - 273.15
    return res
```

24

This formulation of the function does exactly the same as the previous option. We first set the variable **res** to **None**, that is, to error. So, if nothing happens now, the function returns the error value. But something else happens if the **kelvin** argument is a valid value. Then the value of **res** (currently the error code) is replaced by the value converted to Celsius.

By the way: We could have moved the whole logic of checking whether the user input is valid or not from the function **kelvin_to_celsius()** to the main program and simply call the function only if the input was valid. Otherwise, a corresponding error message would be output.

? 24.1 [15 min]

Modify the text editor code from ► Sect. 22.4 so that

- (a) A file is read when opening and written when saving only if a file name actually exists (i.e., the user did not click “Cancel” in the file open dialog without selecting a file).
- (b) A click on “Save” automatically opens the “Save File As” dialog if no file name has been specified yet.

With these changes, you avoid the nasty error messages in the run console when, for example, the user clicks “Save” without first specifying a filename; in this case, Python tries to write to a file with an empty filename (“”), which of course fails.

Handling situations in which further execution of the program would lead to an error is a common use of branching with the help of If-Else constructs.

24.1.2 Nested if-else Constructs

Now suppose we wanted to convert our function **celsius_to_kelvin()** to a function **convert_temperature(temperature: float, toCelsius: bool)**, which converts a temperature specification either from Kelvin to Celsius or from Celsius to Kelvin. A new **bool** argument **toCelsius** should specify in which direction the conversion should take place. If this argument has the value **True**, then Kelvin is converted *to Celsius*, otherwise vice versa. Also, in the new function **convert_temperature()** we want to make sure that only permissible temperatures are specified as argument temperature, i.e. temperatures greater than 0 if it is a Kelvin value (**toCelsius ==**

True) or greater than -273.15° if temperature contains a Celsius value (**toCelsius** **== False**).

Our function could then look like this:

```
1  def convert_temperature(temperature: float,
2      toCelsius: bool) -> float:
3      if toCelsius == True:
4          if temperature >= 0:
5              res = temperature - 273.15
6          else:
7              res = None
8      else:
9          if temperature >= -273.15:
10             res = temperature + 273.15
11          else:
12             res = None
13      return res
```

We could now call the function, for example, with

```
print(convert_temperature(100, False))
```

to convert 100 °C to Kelvin.

If you take a closer look at the function, you will notice that two If-Else constructs have been nested here: The “outer” one checks (line 2) in which direction the user wants to convert the temperature and branches accordingly; the “inner” constructs (lines 3 and 8) check whether the argument temperature has a valid value in the respective program branch. If this is the case, the conversion is performed and stored in the local variable **res** (lines 5 and 10), otherwise (in case of an invalid temperature specification) **res** is assigned the error value **None** (see Sect. 21.3.4) (lines 6 and 11).

The “outer” **if** and the “outer” **else** are each followed by a code block (indented, of course), which in turn contains another If-Else construct. Theoretically, there are no limits to the depth of nesting, but in practice the code becomes more and more difficult to read with increasing nesting depth, almost like a natural language sentence that opens many subordinate clauses and closes them again in quick succession at the end of the sentence. For the sake of clarity, it is advisable to make an appropriate comment, at the latest when more than two constructs are nested, especially to describe to which condition the many different **else** clauses belong. The same applies if the code within the **if** and **else** blocks is long (in the example above, these code blocks each comprise only a single line), because then you read a lot of code and suddenly come across an **else** you no longer know to which if condition it belongs.

By the way: Instead of **if toCelsius == True**, we could have formulated the “outer” condition more simply as **if toCelsius**; the comparison with the value **True** may thus be omitted. This is because Python, if no comparison value is specified, compares with **True** by default.

24.1.3 If-Else Construct with Composite Conditions

We could have written the `temperature_convert()` function from the previous section like this:

```

1      def convert_temperature(temperature: float,
2                                toCelsius: bool) -> float:
3          if toCelsius == True and temperature >= 0:
4              res = temperature - 273.15
5          else:
6              if toCelsius == False and temperature >= -273.15:
7                  res = temperature + 273.15
8              else:
9                  res = None
10         return res

```

This formulation of the function does the same as the function above but is structured differently with regard to the nesting of the If-Else constructs. The “outer” construct checks two conditions at once, namely whether a conversion to Celsius is desired (`toCelsius == True`) and whether the specified (Kelvin) temperature is permissible for it (`temperature >= 0`). These two partial conditions are combined with **and** to a total condition, which is true only if *both* partial conditions are true. In this case, the conversion is performed (line 3). But if at least one partial condition is not true, i.e., either `toCelsius == False` or `temperature < 0` (or even both!), the program is continued in the **else** block of the outer If-Else construct (line 4).

There again a compound condition is checked, namely whether a conversion from Celsius to Kelvin is required and whether the argument temperature is permissible for this conversion. If both partial conditions are fulfilled, the conversion is carried out (line 6). But if this condition is not fulfilled either, the program goes to the “innermost” **else**-block: Since `toCelsius` can only take the values **True** or **False** (at least when the function is called “properly”), there is now only the possibility that the argument `temperature` has a value that is not permissible for the required conversion. So, in this case the variable `res` is assigned the error value **None** (line 8).

In this example, we have linked two partial conditions with a logical *AND*. Other important logical operators besides **and** are:

- **or**, the logical *OR*, with which two conditions are linked in such a way that the total condition is true if *either* one or the other or *both* conditions are true.
- **not**, the logical *NOT*, with which the truth of a statement can be reversed.

So the condition `toCelsius == True and temperature >= 0` could have been written—admittedly somewhat awkwardly—as `not(toCelsius == False or temperature < 0)`. Then the overall condition would have required that it must not be true that either `toCelsius` equals **False**, or `temperature < 0`, or both. In other words: Only if `toCelsius == True` and `temperature >= 0`, the overall condition is fulfilled and the

following code block (namely the conversion from Kelvin to degrees Celsius) is executed. By the way: You do not always have to link compound conditions with logical operators. For example, instead of

```
if x >= 0 and x < 10
```

Python (unlike many other programming languages) also allows the more compact notation

```
if 0 <= x < 10
```

This causes exactly the same check, namely whether the value of the variable **x** is at least 0 but less than 10.

? 24.2 [10 min]

Write a function `age_in_seconds(age_years: int) -> int`, which converts an age in years passed as argument `age_years`, into seconds; the function shall thus calculate and return the *minimum* seconds already experienced. Specific error messages are to be output if the argument `age_years` is not an integer variable, or the age is less than 0 or greater than 120.

If you don't remember how to check whether a value is an integer, scroll back to ► Sect. [21.4.1](#).

24.1.4 If-Else Construct with Alternative Conditions (elif)

Sometimes you want to check several similar conditions. Many other languages provide a Switch-Case construct for this, as we saw in ► Sect. [14.6](#). Switch-Case is excellent for checking an expression (for example, a variable) for several different values. There is no such construct in Python. However, with the **elif** keyword, there is a very efficient way to check multiple alternative conditions to the **if**-condition of an If-Else construct; this way of formulating the conditions is even more flexible than a conventional Switch-Case construct.

As an example, suppose you have developed a program that allows the user to select a file and then specify what to do with the file. In doing so, the file can be renamed, deleted, copied to another directory, or moved there. To select the desired action, the user simply enters the first letter of the desired action, i.e., **r** (rename), **d** (delete), **c** (copy), or **m** (move). After the user enters this action request, it is stored in a **str** variable `action`. Now, what is the condition that checks which action the user wants to perform? Using nested If-Else constructs as in ► Sect. [24.1.2](#), we could choose the following formulation:

```
if action == 'r':
```

```

    # Code executed if the user enters 'r'
    # (rename the file)
else:
    if action == 'd':
        # Code executed if the user enters 'd'
        # (delete the file)
    else:
        if action == 'c':
            # Code executed if the user enters 'c'
            # (rename the file)
        else:
            if action == 'm':
                # Code executed if the user enters 'm'
                # (move the file)
            else:
                # Code executed if the user enters
                # something else

```

This nested formulation checks in sequence which action code has been entered. If 'r' was not entered, 'd' is checked next, if this was not entered either, then 'c' and so on. Due to the nesting, the whole thing appears somewhat confusing. With the help of the **elif** keyword it is possible to structure this part of the program much more clearly:

```

if action == 'r':
    # rename
elif action == 'd':
    # delete
elif action == 'c':
    # copy
elif action == 'm':
    # move
else:
    # no valid action code

```

With **elif**, on the same level as the introductory **if**, further conditions can be checked. The (optional) **else** code block at the end is used if neither the **if** condition nor one of the **elif** conditions is true. If one of the **elif** conditions is in fact true, its code block is executed and the program is continued *after* the **else** code block, i.e., after the whole If-Elif-Else construct. The other **elif** conditions are no longer checked, because **elif** formulates *alternative* conditions. In this way, a highly convoluted and difficult to understand construct can be replaced by a very clear, easy to read notation.

? 24.3 [10 min]

Consider the If-Elif-Else construct in the following program excerpt.

- What values do the variables **a** and **b** take after running the program extract if their values at the beginning are **a = 100, b = 50**?
- What values do the variables **a** and **b** take after running the program extract if their values at the beginning are **a = 110, b = 40**?
- What combinations of values of **a** and **b** result in **b = 0** in the end?

If you don't get the solution directly in subtasks (a) and (b), enter the code in Python and try it out.

```
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
```

24.2 Events

Like If-(Elif-)Else constructs, *events* are used to control the program flow, such as to be able to react to user input. However, unlike If-(Elif-)Else constructs, events do not simply run through the program linearly from beginning to end, branching into certain parts and “skipping” others. Instead, events cause a specific function, the *event handler*, to be called and the code it contains to be executed.

This can be seen very nicely in our **tkinter** programs, such as the calculator application from ► Sect. 22.2.6. The function **equal_press()**, for example, is an event handler that is called whenever the user has clicked on the button with the equal sign. When the code in the event handler is completely processed, the program jumps back to the main loop of the program—in **tkinter** programs, this is the function **mainloop()**. In this loop, the program “waits” for the next event to occur for which an event handler is defined. Once that happens, the event handler gets control; after the event handler has been passed, the program goes back into “lurking” mode. In ► Sect. 25.2, we will use simple means to build a program with a main loop and event handlers that runs in the Python (run) console, i.e., without a graphical user interface.

24.3 Summary

In this chapter, we have dealt with how branches can be built into the program code, so that not always the entire code, but, depending on conditions, only individual parts are executed. The central tools for this are the If-Else construct and events.

Be sure to take the following points from this chapter:

- If-Else constructs allow you to make the execution of parts of the program code dependent on conditions; **if** (and **else**) is followed by the code block that is to be executed if the condition (or its alternative) is fulfilled; the **else** branch can also be omitted; the general structure is thus: **if condition: code_block else: code_block**.
- If-Else constructs can be nested within each other.
- With **elif** several further conditions can be formulated, which are only checked if the **if**-condition and possibly further, preceding **elif** constructs were not fulfilled.
- Conditions are regularly formed by comparisons; the operator for checking equality is the double equal sign (**==**), the single equal sign is used for assignments; for inequality relations **!=** is used.
- Several (sub-)conditions can be linked with logical operators (especially **and**, **or**, **not**) to form a composite condition.
- Events are processed by defining a special function (event handler) that is called whenever the event is triggered. Events are especially useful in programs with a graphical user interface.

24

24.4 Solutions to the Exercises

■ Exercise 24.1

The (event handler) functions **saveas_press()**, **save_press()** and **open_press()** have to be adapted. The supplemented If-Else constructs are highlighted in *italics* in the following sample solution:

```
def open_press():
    global filename
    fname = askopenfilename(defaultextension = 'txt',
                             filetypeypes = [('Text files', '*.txt'),
                             ('All files', '*.*')],
                             title = 'Open File...',
                             initialdir = 'C:\\Windows')

    if fname != '':
        textfile = open(fname, 'r')
        text.delete(1.0, END)
        text.insert(1.0, file.read())
        textfile.close()
```

24.4 Solutions to the Exercises

```

        status['text'] = 'File "' + fname + '" opened.'
        filename = fname

def saveas_press():
    global filename
    fname = asksaveasfilename(defaultextension = 'txt',
                              filetypes=[('Text files', '*.txt'),
                                           ('All files', '*.*')],
                              title = 'Save File As...',
                              initialdir = 'C:\\Windows')
    if fname != '':
        textfile = open(fname, 'w')
        textfile.write(text.get(1.0, END))
        textfile.close()

        status['text'] = 'File "' + fname + '" saved.'
        filename = fname

def save_press():
    global filename
    if filename != '':
        textfile = open(filename, 'w')
        textfile.write(text.get(1.0, END))
        textfile.close()

        status['text'] = 'File "' + filename + '" saved.'
    else:
        saveas_press()

```

■ Exercise 24.2

A possible formulation of the function could look like this:

```

def age_in_seconds(age_years: int) -> int:
    if not isinstance(alter_years, int):
        print('Error: The age is not an integer value!')
    else:
        if age_years < 0 or age_years > 120:
            print('Error: Implausible age information!')
        else:
            return age_years * 365 * 24 * 60 * 60

```

First, the **isinstance()** function is used in an outer If-Else construct to check whether the argument **age_years** is not (!) an integer value. If this is the case, an error message is issued and the function is exited. We take advantage here of the fact that **isinstance** returns a **bool** value and Python compares an expression in a condition

against **True** by default. Alternatively, we could have written **if not isinstance(age_years, int) == True** or **if isinstance(age_years, int) != True**.

However, if **age_years** is indeed an integer value, the inner If-Else construct checks whether the age is outside the plausible limits. In this case, the function is again exited with an error message. If this hurdle is also successfully overcome, the function returns the age in seconds.

You could then call the function for example with

```
print(age_in_seconds(38))
```

24

■ Exercise 24.3

- (a) **a = 100, b = 5** (the condition applies: **a >= 90** or **b <= 50**)
- (b) **a = 5, b = 20** (the conditions: (**a > 90** and **b < 50**) or (**a == 100** and **b > 50**) and **a >= 100** or **b <= 50** apply)
- (c) None. For this, the condition **a >= 95** and **b < 15** would have to be fulfilled. But this is an alternative condition to **a > 90** and **b <= 20**. Every combination of **a** and **b** that satisfies **a >= 95** and **b < 15** also satisfies **a > 90** and **b <= 20**. The alternative condition therefore never comes into play, it is “dead code”.



Loops: How Do I Repeat Program Instructions Efficiently?

Contents

- 25.1 Counting Loops (for) – 366**
 - 25.1.1 Simple for-Loops – 366
 - 25.1.2 Nested for-Loops – 370
 - 25.1.3 List Comprehension Expressions – 371
- 25.2 Conditional Loops (while) – 372**
- 25.3 Exiting Loops Prematurely and Restarting Them – 375**
- 25.4 Summary – 377**
- 25.5 Solutions to the Exercises – 377**

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1007/978-3-658-42912-6_25.

Overview

Next, we focus on the effective repetition of (almost) identical program instructions, a technique that allows for the elegant resolution of a lot of problems that would be hard so solve otherwise. Python employs both counting (**for**) and conditional (**while**) loops. A unique feature of Python is list comprehension expressions **derived from for loops**, enabling the concise comfortable of lists.

In this chapter you will learn:

- How to formulate (counting) **for**-loops
- how to formulate (conditional) **while** loops
- how to nest loops inside each other
- what list comprehension expressions are, and how you can use them to create lists in an elegant way
- how to end loops or individual passes of loops early, and when this is useful

25

25.1 Counting Loops (**for**)

25.1.1 Simple **for**-Loops

■ Structure of **for**-Loops

Counting loops in Python are introduced with the keyword **for**; therefore, we will refer to them simply as **for**-loops in the following. With **for**-loops in Python, a run variable is not incremented with each loop pass, as is the case in many other languages, but an *object* is always *passed through* that has elements that can be addressed one after the other. Such an object is also called *iterable* in Python because you can “shimmy” from one element to the next step by step.

Examples of such objects are lists or tuples. They all have elements that a counting loop can process one after the other. It is not important that the elements in the object are in a fixed order, as is the case with lists and tuples. The only thing that matters is that (except for the last one, of course) a *next* element can always be found. This is also the case, for example, with sets and dictionaries, in which the elements are not stored sequentially, or in an order that can be exploited by the programmer. Nevertheless, the elements of sets and dictionaries naturally do have an internal sequence (usually the sequence in which the elements were added to the set or dictionary) and thus a *next* element can always be determined for these objects as well.

If the iterability of objects and the traversal of their elements seems rather abstract to you, don’t worry: In the following example, you can see how simple **for**-loops are constructed in Python. Here we have a loop that does nothing but output the numbers between 1 and 10 on the screen:

```
for i in [1,2,3,4,5,6,7,8,9,10]:
    print(i)
```

In Python, **for**-loops also have a run variable, in our example **i**. It is defined directly after the **for** keyword. The run variable takes the value of an element of the traversed object on each loop pass. In our example, the object being passed through is a list with the numbers 1 to 10. In sequence, the run variable **i** is now loaded with the value of a list element; in the first loop pass with 1, in the second with 2, until the run variable finally takes on the value of the last list element, 10. Our list is iterable, so Python always knows which element is next in line for the next loop pass. We can also work with the changing value of the run variable in the code block that is executed on each loop pass, and that's exactly what we're doing here in the example: the code block is indented following the *head* of the loop introduced with the **for** keyword and terminated with a colon. In our example, the code block consists of only one statement, the output of the run variable. If you execute the code of the example, you will get the numbers from 1 to 10 as output, as expected.

It is not always practical to specify a list of numbers as explicitly as we did in our example, especially if the list is very long, or if its bounds are not yet known at the time the program is developed and result from variables in the program. In these cases, the function **range(start, stop, step=1)** is a useful tool. It generates a sequence of numbers between **start** and **stop** in the interval of **step**, where the latter is an optional argument and is assumed to be 1 if it is not explicitly specified when calling the range. Note: The value **stop** itself is not part of the generated sequence of numbers. You may recognize here the analogy to indexing with an index range; in ► Sect. 21.6.1.2 we discussed this kind of indexing using the colon operator. You can easily visualize this by typing **list(range(1,10))** in the Python console. You get a sequence of integers from 1 to 9. The return object of **range()** is an object of class **range** and must first be converted to a list for output, because the **__str__()** method (see ► Sect. 21.1 for details on **__str__()**) of the **range** class does not display the sequence of numbers itself—try it!. However, this object can be iterated.

This would simplify our example above to:

```
for i in range(1,11):  
    print(i)
```

Now consider the following example, where we adjust the value of our run variable during the loop:

```
my_range = range(1,11)  
for i in my_range:  
    i = 2  
    print(my_range)
```

The output generated by the final **print()** statement shows that the object **my_range** has not changed, although we assign the value 2 to our run variable **i**, which represents a different element of **my_range** on each loop pass. However, this obviously has no effect. The reason is that the run variable in a **for**-loop always represents only a *copy* of the element that is currently in focus. Thus, by assigning **i=2**, we do

not change the respective element of our **range** object at all, but only the run variable itself.

■ Examples of for-Loops

Now, outputting a sequence of numbers is not necessarily a particularly useful application of **for**-loops. Therefore, in the next example, we look at a simple encryption algorithm, also known as *ROT13*. Each character of a string can be interpreted as a numeric code that uniquely identifies the character in the particular character set. For example, in the very common UTF-8 character set, the number code 65 represents the capital “A”, 66, the capital “B”, and so on. The lowercase letters follow from 97. The ROT13 algorithm takes advantage of this convertibility of letters into numbers by simply incrementing each letter code by a specific number—13 in the original. Thus, “A” (code 65) becomes “N” (code 78), “B” (code 66) becomes “O” (code 79), and so on.

We want to develop a function **rot13(s: str, decode: bool, shift: int=13)** that takes the string argument **s**, and depending on the **bool** argument **decode** either encodes or decodes this string. This should use a character code shift of 13 by default, but it should also be possible to set a different shift using the argument of the same name.

Of course, the question arises as to where exactly a **for**-loop comes into play here. But you probably already guessed: As you already know, the string **s** can of course be broken down into a list of characters; we can run through this list and thus process each individual character of the string in turn. To do this, we use the functions **ord(character)** and **chr(code)**, which return the numeric code for a given character and the character for a numeric code, respectively.

The function **rot13()** could then look like this:

```
def rot13(s: str, decode: bool, shift: int = 13) -> str:
    s = list(s)
    res = list()
    for c in s:
        if decode == True:
            res.append(chr(ord(c) - shift))
        else:
            res.append(chr(ord(c) + shift))
    return ''.join(res)
```

The **for**-loop runs through the string **s**, which we have previously converted into a list of individual characters. The run variable **c** therefore represents a different character of the string **s** in each loop pass.

With **append()** we gradually build a new list containing characters of the encoded or decoded string. Finally, we **join()** the characters of the list to a string, append it to an empty string (whose **join()** method we use for this) and return it as a return value.

We can now call the function with **print(rot13('HELLO WORLD', False))**, for example, which prints **URYYY-d\YQ** to the console. With **print(rot13('URYYY-d\YQ**

YQ', True)) we can decode the encoded text again. If the end result is **'HELLO WORLD'** again, you know your function is working correctly!

To conclude this section, let's look at another example. This time, we want to write a program that lets the user enter a directory and then lists the subdirectories and files it contains. To do this, we will use functions from the standard Python package **os**. This package provides ways to access operating system functionality and resources. This includes functions that allow you to work with the file system. The beauty of **os** here is that it is designed to be cross-platform; therefore, a program that uses **os** functions and works on your Windows computer will work just as well on a Mac or Linux computer. You do not have to worry about the specifics of these different operating systems. The **os** package provides a uniform, cross-platform programming interface for all of these systems that abstracts from the specifics of the systems.

The code of our program could now look like this:

```
from os import listdir, sep
from os.path import isdir, isfile, getsize

directory = input('Please enter a directory: ')

if isdir(directory):
    if directory[directory.__len__() - 1] != sep:
        directory = directory + sep
    print('\nContents of:', directory)

    content = listdir(directory)

    print('\n----- Directories:')
    for elem in content:
        if isdir(directory + elem):
            print(elem)

    print('\n----- Files:')
    for elem in content:
        if isfile(directory + elem):
            print(directory + elem, '\tSize: ',
                  getsize(directory + elem), sep = '')
```

You can tell from the imports which elements of the **os** module we want to use. The **listdir(path)** function returns the contents of the specified directory as a list. The constant **sep** represents the separator used in path specifications on the respective system, i.e., the backslash (****) for Windows, the forward slash (**/**) for Linux. Finally, using the **isdir(path)** and **isfile(path)** functions from the **path** module, we can determine whether a given directory or file path leads to a valid directory or file. Both functions return a **bool** value. The last imported function, **getsize(filename)**, returns

the size of the specified file in bytes. Please note: Whenever you specify a file, the full path must be included, otherwise the file will not be found.

The code uses two **for**-loops that step through the directory contents determined by **listdir()**. The first loop processes only the directories, so that these are then at the top in the program output, the second only files, which then appear below the directory list in the output.

? 25.1 [10 min]

Write a function that removes all vowels from a string.

? 25.2 [10 min]

Write a function that completely reverses a string, converting it to uppercase.

25

25.1.2 Nested for-Loops

for-loops can also be nested within each other. Consider the following example:

```
letters = ['A', 'B', 'C', 'D', 'E', 'F', 'G']
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 8, 10]
for b in letters:
    for z in numbers:
        print(b + str(z), ' ', end = '')
    print('\n')
```

Here we see two **for**-loops: an *outer* loop that runs through the list of letters from A to G and an *inner* loop that takes care of the numbers from 1 to 10. The inner loop is indented in the code block of the outer loop. With each run of the outer loop, the inner **for** construct is run through completely, i.e. the numbers from 1 to 10 are processed. After that, the outer loop goes into its next pass, that is, into processing the next letter. What do you think this little program will output?

This is what we see when we run the program:

```
A1 A2 A3 A4 A5 A6 A7 A8 A10
B1 B2 B3 B4 B5 B6 B7 B8 B10
C1 C2 C3 C4 C5 C6 C7 C8 C10
D1 D2 D3 D4 D5 D6 D7 D8 D10
E1 E2 E3 E4 E5 E6 E7 E8 E10
F1 F2 F3 F4 F5 F6 F7 F8 F10
G1 G2 G3 G4 G5 G6 G7 G8 G10
```

Thus, for the letter currently being passed through in the outer loop, **b from** in the list **letters**, the program writes the combination of this letter and *all* numbers from 1 to 10 passed through in the inner loop, one after the other; note the argument assignment **end = ''** in the function **print()**, which causes *no* line break to occur after each output. Only after the inner loop has been completely traversed is a new-

line (**\n**) output and the outer loop goes into the next pass. Note, that the **print('\n')** is on the indentation level of the outer, not the inner loop. So, the outer loop code block consists of the complete inner **for**-loop and the **print()** statement that creates the newline.

Of course, not only two but even more **for**-loops can be nested in this way. However, you have to be careful with the runtime of the program, because each additional **for**-loop basically multiplies the number of loop passes by its own pass count.

25.1.3 List Comprehension Expressions

A special feature in Python are *list comprehension expressions*. What may sound scary at first is nothing more than a **for**-loop that generates a list. For such **for**-loops there is a compact, elegant notation, the list comprehension expression.

In the last section, we used the **listdir()** function from the **os** module to determine the contents of a directory as a list and then used the **isfile()** function in a **for**-loop to check each element to see whether it was a file (and not a directory). Only if it was indeed a file did we output the item to the screen. We did the same with directories in another **for**-loop. This way we could cleanly separate directories and files in the output. It would have been easier to create two lists from the beginning, one with the files, one with the directories, and then display them with two simple **for**-loops. This is a task that is tailor-made for the use of list comprehension expressions. Consider the following formulation:

```
content = listdir(directory)
files = [f for f in content if isfile(directory + f)]
directories = [f for f in content if isdir(directory + f)]
```

First, we get the contents of our selected directory with **listdir()**. Then we use two list comprehension expressions to filter out the files and directories from the list **content**. The list comprehension expression returns something, namely the elements that should be in the result list. What is returned is initially enclosed in the square brackets that signal the construction of a list; in our example, this is **f**. Now follows a **for**-loop, namely **for f in content**. This loops through each element of the list **content**. However, only those elements are returned that satisfy the condition defined by **if**, i.e., in our first example, all those elements for which the function **isfile()** returns the value **True**. The list comprehension expression thus goes through all elements of **content**, the list with the directory contents, and returns those that are files.

The **if** condition is optional. For example, if we wanted to get a list of all items in the directory, but in upper case, we could omit the condition, but we would still modify the result before returning it by calling the **upper()** function:

```
content_capitalized = [f.upper() for f in content]
```


This example also shows that the expression that is “fed” with new elements by the **for**-loop of the list comprehension expression can also be a more complex expression than simply the object itself returned by the **for**-loop; in this case, a method is called for this object.

Another example is the following, where we square the value of the variable returned by the **for**-loop if the variable itself is evenly divisible by two, that is, an even number. To do this, we use the modulo operator `%`, which returns the remainder of a division.

```
squares = [x*x for x in range(1,11) if x % 2 == 0]
```

25

Here, the expression that is fed with values by the **for**-loop is just `x*x`.

So, list comprehension expressions generally have the following form (where, we have already seen, the condition is optional):

```
list = [expression for variable in iterable_object
        if condition]
```

? 25.3 [5 min]

Write a list comprehension expression that summarizes the uppercase letters from A to Z as a list. Tip: Work with functions the `ord(characters)` and `chr(code)` that we also used in ► Sect. 25.1.1.

What would an equivalent **for**-loop look like that produces the letter list without a list comprehension expression?

25.2 Conditional Loops (while)

Python allows a conditional loop construction with the **while** loop, i.e., a loop that runs as long as a run condition is fulfilled. Python’s **while** loop is *head-controlled*, which means that the condition is checked *before* each run. If the condition is not fulfilled even before the first potential run, the loop is not run at all.

A **while** loop can of course be used to simulate the same behavior as a **for**-loop. In the following example, a **while** loop outputs the numbers 1 to 10 on the screen:

```
i = 1

while i <= 10:
    print(i)
    i = i + 1
```

We initialize a variable **i**, which we use as a run variable, with the value 1. Before each run, the loop checks whether the condition that **i** is less than or equal to 10 is still met. If it is, the block of code that begins after the colon, indented as usual, is run. The loop then jumps back to the loop head, where the run condition is checked again.

The most common mistake in such a construction is to forget to also increment the run variable (as happened to the author in his first attempt), because unlike the **for**-loop, the **while** loop does not perform this task itself. The **while** loop stubbornly continues to run as long as the run condition is met. We have to take care of the rest ourselves. If the run condition is *always* fulfilled, we have created an infinite loop, which is usually undesirable.

The **while** loop shows its true strength, however, when—unlike in the example above—the number of loop passes cannot be determined in advance. For example, because the passing through the loop depends on an event that results from an interaction with the user. Consider the following example. It lists all files contained in a directory selected by the user and allows him to open or start one of these files. We accomplish the latter with the **startfile(file)** function from the **os** module, which we have already used in the previous sections. In our simple application, the user can specify the file to be opened/started by a number with which the file is listed. We also offer the user the possibility to change the directory or to quit the program completely. The core of the program consists of a **while** loop, in which we ask the user again and again for his action wishes (variable **choice**), and which enters via a letter ('d' for read directory, 's' for open/start file, 'q' for quit). The condition of the **while** loop is that the user's choice is different from 'q'. As long as this is the case, the program continues to run, if this is no longer the case, the program ends.

Here is the complete program code:

```
from os import listdir, startfile
from os.path import isfile, isdir

choice = ''
dir_exists = False

while choice != 'q':
    choice = input('Read directory (d), start file (s), ' \
                  'quit (q)? ')
    if choice != 'q':
        if choice == 'd':
            directory = input('Please enter a directory: ')
            if isdir(directory):
                dir_exists = True
                files = [f for f in listdir(directory)
                        if isfile(f) == True]
                number = 0
```

```

        for f in files:
            if isinstance(directory + f):
                number = number + 1
                print(number, ': ', f, sep = '')
        else:
            print('"' + directory,
                  '" is not a valid directory!',
                  "Please try again.", sep = '')
            dir_exists = False
    elif choice == 's':
        if dir_exists == True:
            number = int(
                input('Please enter file number: '))
            if number >= 1 and number <= files.__len__():
                startfile(directory + files[number - 1])
            else:
                print('No directory loaded!')
    else:
        print('Input ', choice,
              ' is not permitted! Please try again.',
              sep = '')

```

When you run the program, please note that the directory specification must always end with a path separator, i.e. the backslash (`\`) on Windows systems or the forward slash (`/`) on Mac and Linux systems, otherwise the file path assembled in the program will be invalid.

Within the **while** loop, the query of the action desired by the user takes place first. If this choice is different from **'q'**, the actions **'s'** (start) and **'d'** (read directory) are checked and processed. Otherwise, if the user has entered **'x'**, nothing happens inside the **while** loop, so the loop jumps back to checking the run condition in the loop head and determines that the run condition **choice != 'q'** is no longer met. Therefore, the program processing continues after the code block of the **while** loop. In our example, this terminates the program.

As you can see from this program example, with **while** loops we can ultimately create event control by waiting for user input, processing it when it comes, and then waiting again for the next user input, until the user wants to exit the program. The **mainloop()** function in the **tkinter** programs from ► Sect. 22.2 ultimately does nothing else. Event control is therefore not witchcraft but can be implemented with classic control structures such as **while** and **if**.

? 25.4 [20 min]

Write a program that uses the **convert_temperature()** function from ► Sect. 24.1.2 to convert temperatures between Kelvin and degrees Celsius (and vice versa). The user of the program should be offered a “menu” of actions **'k'** (conversion from degrees Celsius to Kelvin), **'c'** (conversion from Kelvin to degrees Celsius) and **'q'** (exit program) until they enter **'q'** to exit the program.

25.3 Exiting Loops Prematurely and Restarting Them

■ Ending Loops by Statement

With the statements **break** and **continue**, Python offers possibilities to control the flow of **for** and **while** loops even more precisely. Calling the **break** statement causes the loop to terminate and program execution and to continue *after* the loop code block.

Consider the following example:

```
while True:
    x = input('Your input: ')
    if x == 'x':
        break
    print('Your input was: ', x)

print('Loop completed.')
```

This **while** loop has a condition (**True**), which by definition is always met. Principally, it would run forever. However, we put in a condition that makes sure it only runs until the user enters 'x'. When that happens, the loop is terminated with the **break** statement in place. The call to **print()**, which is inside the loop code block but after the **break** is no longer executed.

The following function uses **break** to exit a **for**-loop early. This can be useful, for example, when determining whether something exists. Once an instance of what is being searched for has been found, further loop passes no longer make sense, since they can no longer change the result of the search. In this situation, it is smart to exit the loop to save computation time and speed up program execution.

The following example shows a function that checks the files in a directory to see if there is an Excel file with a size of more than 1 MB (= 1,000,000 bytes).

```
from os import listdir
from os.path import isfile, getsize

def big_excel(directory: str) -> bool:
    files = [f for f in listdir(directory)
              if isfile(directory + f)
              and (f[-5:].lower() == '.xlsx'
                   or f[-4:].lower() == '.xls')]

    res = False
    for f in files:
        if getsize(directory + f) > 1000000:
            res = True
            break
    return res
```

To do this, a list of Excel files in the directory to be searched, which is passed to the function as the **directory** argument, is first generated, using a list comprehension expression. This checks which file names have an extension that indicates Excel files (for the sake of simplicity, we will only search for **.xls** and **.xlsx** files here and ignore other possible file name extensions of Excel files). The list comprehension expression thus checks the condition that the respective directory content item is a file and that its name ends in **.xls** or **.xlsx**.

Then a **for**-loop goes through the list generated by the list comprehension expression and checks with the **getsize(file)** function whether any of these files is larger than 1 MB. Once such a file is found, the loop is exited with **break**. Before that, the return value **res** of the function is switched from its default value **False**, to which it was initialized before the start of the loop, to **True**. After the loop, the **return** statement is executed, which returns this value to the caller.

Of course, the condition that the file size must exceed 1 MB could have been included in the list comprehension expression and then the **for**-loop could have been omitted altogether (How? Try it out!) But the example shows how you can make a search loop efficient by terminating it as soon as the thing you are looking for has been found. The same effect could have been achieved with a **while** loop, which would have to look like this:

```
i = 0
res = False
while i <= files.length() and res == False:
    if getsize(directory + files[i]) > 1000000:
        res = True
    i = i + 1
```

The formulation with the **for**-loop seems a bit more intuitive (and avoids the risk of forgetting to increment the run variable **i**).

■ Continue Loops per Instruction with the Next Run

While **break** completely exits the loop within which it is called, the **continue** statement causes the loop execution to simply continue with the next loop pass. This can be seen in the following example, where we loop through the numbers from 1 to 10 in a **for**-loop, but “pinch off” the **print()** statement located in the loop code block for all even numbers by sending the loop into its next pass beforehand with **continue**:

```
for i in range(1,10):
    if i % 2 == 0:
        continue
    print(i)
```

Here we use the *modulo operator* **%**, which returns the division remainder; if this is 0, it is an even number and the loop continues with the next pass, the call to **print()**

at the end of the loop code block is then not reached at all. It is only reached for odd numbers, so that only these are output to the screen.

25.4 Summary

In this chapter, we have looked at **for** and **while** loops, as well as list comprehension expressions based on the **for**-loops. We also saw how loops or the current loop pass can be exited early.

Be sure to take the following points from this chapter:

- Python knows the **for** construct as a counting loop that iterates through the elements of an iterable object; an object is iterable if Python can “shimmy” from one element/component of the object to the next (as is the case with lists, tuples, sets, or dictionaries, for example).
- The basic structure of the **for**-loop is: **for** *variable* **in** *iterable_object*: *code_block*; the **variable** represents a different element that is part of *iterable_object* on each loop pass.
- Changes to the run variable do *not* change the element of the traversed iterable object that it represents.
- **for**-loops can be nested arbitrarily.
- A special kind of **for**-loop is the list comprehension expression. It produces a list with a very compact syntax; the general form is: **list** = [*expression* **for** *variable* **in** *iterable_object* **if** *condition*], so for example the list of squared even numbers from 1 to 10: **squared** = [*x***x* **for** *x* **in** *range*(1,11) **if** *x* % 2 == 0], where % is the modulo operator that returns the remainder of a division.
- Conditional loops are constructed with **while**; their general form is: **while** *condition*: *code_block*.
- Both **for** and **while** loops can be exited early with the **break** statement.
- The **continue** statement terminates the current loop pass and continues the loop with the next pass.

25.5 Solutions to the Exercises

■ Exercise 25.1

A function that removes all vowels from a string might look like this:

```
def remove_vowels(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 whether the function has the desired effect:

```
print(remove_vowels('Hello world!'))
```

■ Exercise 25.2

To reverse the characters in the string, the string, which has been converted into a list of individual characters, is run through *from back to front* in a **for**-loop. The increment, the third argument of the **range()** function, is therefore **-1**. Because **range()** runs to *before* the specified boundary, the second range boundary must also be **-1** so that the last index value returned is 0, thus allowing access to the first character of **s**.

25

```
def invert_capitalization(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))
```

The function could then, for example, be called like this:

```
print(invert_ capitalization('Hello world!'))
```

■ Exercise 25.3

A list comprehension expression that summarizes the letters from **A** to **Z** as a list might look like this:

```
capital_letters = [chr(b) for b in range(ord('A'), ord('Z')+1)]
```

So we run through the character codes with the variable **b** starting with the code of the letter **A**, up to the code of the letter **Z** (Note: **range()** returns a range of values which no longer contains the specified right/upper limit, therefore **+1**). We then convert these codes back into letters using the **chr()** function.

An equivalent **for**-loop would look like this:

```
capital_letters = list()
for b in range(ord('A'), ord('Z')+1):
    capital_letters.append(chr(b))
```

■ Exercise 25.4

For example, the program might look like this:

```
choice = ''

while choice != 'q':
    choice = input('Conversion Celsius to Kelvin (k),
                  Kelvin to Celsius (c), quit (q)? ')
    if choice == 'k':
        temp = input('Please enter a temperature in
                      degrees Celsius: ')
        orig_unit = 'Grad Celsius'
        target_unit = 'Kelvin'
        to_celsius = False
    elif choice == 'c':
        temp = input('Please enter a temperature
                      in Kelvin: ')
        orig_unit = 'Kelvin'
        target_unit = 'degrees Celsius'
        to_celsius = True
    elif choice == 'q':
        pass
    else:
        print('"' + choice + '" is not a valid input.
              Please try again.', sep = '')

    if choice == 'k' or choice == 'c':
        res = convert_temperature(float(temp), to_celsius)
        if res != -1000:
            print(temp, orig_unit, 'are', res, target_unit)
```

Note that the program uses the `convert_temperature()` function from ► Sect. 24.1.2. You will need this code to run the program. Furthermore, please note that the variable `choice`, which takes the user's command, must already be initialized before the `while` loop. Otherwise, the `while` loop will encounter a non-existent variable when checking its run condition, which will trigger an error message.



Debugging & Error-Handling: How to Find and Fix Errors in a Structured Way

Contents

- 26.1 Error Handling at Runtime – 382**
 - 26.1.1 Catching Errors Through Targeted Checks – 382
 - 26.1.2 Try-Except Constructs – 383
- 26.2 Troubleshooting During Development – 386**
 - 26.2.1 Breakpoints – 386
 - 26.2.2 Display of Variable Content and Use of Watches – 388
 - 26.2.3 Step-by-Step Execution – 389
- 26.3 Summary – 390**
- 26.4 Solutions to the Exercises – 391**

Overview

Errors are part of everyday programming, an unpleasant but unfortunately irrefutable truth. Therefore, to conclude our tour through the basics of Python, we will deal with how errors can be found, analyzed and handled.

In this chapter you will learn:

- How to make the program more robust by appropriate checks in your program code, for example against user input errors
- how to catch errors at runtime with the help of exceptions and how to process them in an orderly manner
- how to use *PyCharm*'s debugging tools to find and analyze bugs during development

26

26.1 Error Handling at Runtime

26.1.1 Catching Errors Through Targeted Checks

In ► Chap. 16, we defined errors *at runtime* as errors that arise only from circumstances of the concrete program execution. It is true that there are application scenarios in which the code achieves exactly the intended goal without errors; and it was precisely these scenarios that were the focus of development. In other words, the program was tested based on such scenarios. However, situations can occur during execution in which the program does *not* do what it is supposed to do. This is especially the case when program execution gets out of control and the program “crashes”, i.e., terminates with an error and (without restarting the program) cannot be used any further.

A common cause of such errors is problems with type conversions of variables. Consider the following example:

```
from math import sqrt

x_str = input('Enter an integer number: ')
x_int = int(x_str)
print(sqrt(x_int))
```

The program reads a number from the user and calculates the square root from it. The `sqrt()` function from the `math` module is used for this.

Since the `input()` function always returns a string, the user input must first be converted to a float variable, i.e., a floating-point number, using a normal type conversion (if you are no longer familiar with this, scroll back to ► Sect. 21.5). If you now start the program and enter, for example, 25, you will get the square root as output, i.e., 5. The program apparently works without errors, at least in the

application scenario we tested. But what happens if the user enters not a number but a string (for example, 'hello')? We immediately get a nasty error message from Python:

```
x_float = int(x_str)
ValueError: could not convert string to float: 'hello'.
```

Of course, the error can be easily caught by first checking in an **if** statement whether the user's input is actually a number. Our program could then look like this, for example:

```
from math import sqrt

x_str = input('Enter a positive integer number: ')
if x_str.isnumeric():
    x_int = int(x_str)
    print(sqrt(x_int))
else:
    print('Your input "', x_str,
          '" is not a positive integer!', sep='')
```

In this way, we “immunize” the program against user input errors. This makes the program more “robust”, it can now handle more scenarios that might occur in the real world without crashing.

Perhaps you have identified another possible source of error: the input of negative numbers. We don't seem to catch this kind of “incorrect input” here. In fact, however, we do, because **isnumeric()** checks whether the string contains only digits. Decimal separators or minus signs contained in the string automatically cause the string not to be considered a number. This would throw our home-made error message, so the **sqrt()** function will never be called with a negative number. Irritatingly, Python does not inherently have a function to determine if a string contains a number of any kind, i.e., a number that may include signs, decimal separators, and exponents. Therefore, we have made it simple here and only allowed the input of a positive integer.

26.1.2 Try-Except Constructs

Not all problems that can occur at runtime can be described so explicitly that we can catch them with a condition such as the example above. For example, when opening files for writing, there can be numerous causes for the opening to fail. For example, the file may be read-only if it already exists, or the disk may be read-only, or there may not be enough space left on the disk to write the file as desired.

Now, of course, it is good to catch as many of these potential error sources as possible in order to be able to tell the user precisely what causes the error and thus enable him to fix the problem, e.g., to remove write protection or to free up sufficient space on the disk. In fact, even if one eliminates some possible causes of errors by targeted checks, there will still be other potential sources of errors that one overlooks or cannot foresee, especially in operations as complicated as working with the file system. Therefore, there is another way to deal with errors in Python, namely with a *Try-Except construct*, which we learned about in ► Sect. 16.2. This construct in Python consists of the keywords **try**, **except**, **else**, and **finally**, the latter two being optional.

With that, our program from the top could look like this:

```
x_str = input('Enter a number: ')
try:
    x_float = float(x_str)
    print(sqrt(x_float))
except:
    print('Your input "', x_str,
          '" is not a number!', sep='')
```

The keyword **try** is followed by a code block with statements whose execution is “tried”. If this execution does not succeed without errors, the code block after the keyword **except** is executed. However, if everything runs as planned, the **except** code block is skipped and the execution of the program is continued after it. Here we can now enter fractional numbers with decimal separators, as well as negative numbers. Enter a negative number! What happens? You get the error message **'Your input is not a number!'** This is not true, of course; the problem is elsewhere, namely in taking the square root. The program terminates with an error because you are trying to take the square root of a negative number. The Try-Except construct catches *all* errors that could occur, including the one caused by the negative argument of **sqrt()**.

We now have two options: Either, we formulate the error message in a more general way (“An error has occurred”), which is of course not very helpful for the user, or we try to differentiate the error causes more sharply. This is possible via *exception classes*. In Python, as in many other programming languages, errors are referred to as “exceptions”, that is, situations in which the program does not behave as it should. Now, an exception class covers errors (i.e., exceptions) of *a certain kind*. The exception class **ZeroDivisionError**, for example, covers the exception that the divisor is 0 in a division.

By the way, the exception class of an error is also displayed in the Python error messages. Thus you can determine which exception class you have to work with. The **except** keyword allows us to explicitly catch certain classes of exceptions. In our example, however, this helps us only to a limited extent, because both the attempt to convert a string that does not contain a number into a **float** variable and the calculation of the square root of a negative number lead to an exception of the

same exception class, namely **ValueError**. But now suppose we extend our program to include a line that determines the reciprocal of the input number. Then we can certainly differentiate the two possible exception classes:

```
x_str = input('Enter a number: ')
try:
    x_float = float(x_str)
    root = sqrt(x_float)
    reciprocal = 1/x_float
except ValueError:
    print('Your input "', x_str,
          '" is not a number or is a negative number!',
          sep='')
except ZeroDivisionError:
    print('The reciprocal of 0 cannot be calculated!')
except:
    print('Other error!')
else:
    print('Root:', root)
    print('Reciprocal:', reciprocal)
```

In this variation of our example, we catch exceptions of the types **ValueError** and **ZeroDivisionError** separately and can therefore issue more specific error messages. In addition, you will notice two further differences to the structure of the Try-Except construct used so far. First, in addition to the two **except** keywords, each of which addresses specific exception classes, you will find another **except** that is specified without an exception class. This construction ensures that any *other* errors that do not fall into either of the two explicitly specified exception classes are also handled. Second, you see an **else** block; the statements it contains are always executed if no exception has been raised (“thrown”, as it is called in programmer jargon).

Of course, the **print** statements could just as well have been included in the **try** code block. There, too, they would only be executed if no exception occurs. However, one generally tries to keep the **try** block as small as possible to make it clear what the “risky” action is here that you fear could throw an exception.

The keywords **try**, **except** and **else** must always be used in this order. **else** is optional, however, and can therefore be omitted.

We have not considered the keyword **finally**, which can optionally follow the **else** code block and whose code block contains statements that are executed *in any case*, regardless of whether an exception was thrown or not. The use of **finally** is especially interesting if, for example, within an **except** block the current function is exited with **return**, or the current loop is exited with **break**. The statements behind the entire Try-Except construct are then no longer executed, but those statements

contained in the **finally** code block are. Only after their execution is the function or loop exited.

? 26.1 [15 min]

Write a program that accepts from the user a number, an operator (addition, subtraction, multiplication, division) and another number, then executes the arithmetic operation described by it and displays the result. The program should be as robust as possible against all possible errors that can occur during its execution.

? ! 26.2 [15 min]

What might be an alternative formulation of the program from Task 26.1 that exhibits similar robustness but forgoes the use of the **try...except** construct?

26

26.2 Troubleshooting During Development

Several handy tools are available for debugging during development when working with an integrated development environment like *PyCharm*. The most important of these are *breakpoints*, *variable and expression watches*, and *step-by-step execution* of the program code. One slightly more advanced topic that we will skip here are automated tests. Such tests allow for a piece of code, for example, a function, to have an expected result defined and then automatically check whether the current code of the function actually produces this result. Thus, the testing occurs entirely automatically; one does not need to perform the test manually. Python libraries that allow for such tests include, for example, *unittest* and *pytest*.

26.2.1 Breakpoints

■ Stop Sign of the Program Execution

A breakpoint is a special marking of a program line which causes the program to be executed only up to this line, and the execution is then stopped. It can be resumed later, with all variables restarting with the value they had at the time of the breakpoint stop.

Breakpoints can be useful, for example, to test at which point an error is triggered in the program that causes the program to crash. If the program runs through to the breakpoint without errors, the cause of the error is definitely not in the code part above the breakpoint. If you set a breakpoint in the code block of an If-Else construct, you can easily determine whether this branch of the construct is actually executed, i.e. whether the respective condition is fulfilled. If the program stops, this is the case, but if it does not stop and runs to the end, the condition was obviously not met. You can also use breakpoints to pause program execution to look at the contents of variables used in the program; this is done with *watches*, which we'll look at in more detail next.

```

1  from os import listdir, startfile
2  from os.path import isfile, isdir
3
4  choice = ''
5  dir_exists = False
6
7  while choice != 'q':
8      choice = input('Read directory (d), start file (s), quit (q)? ')
9      if choice != 'q':
10         if choice == 'd':
11             directory = input('Please enter a directory: ')
12             if isdir(directory):
13                 dir_exists = True
14                 files = [f for f in listdir(directory) if isfile(directory + f)]
15                 number = 0
16                 for f in files:
17                     number = number + 1
18                     print(number, ': ', f, sep = '')
19             else:
20                 print('""', directory, '" is not a valid directory! Please try again.', sep = '')
21                 dir_exists = False

```

■ Fig. 26.1 Code with breakpoint set in *PyCharm*



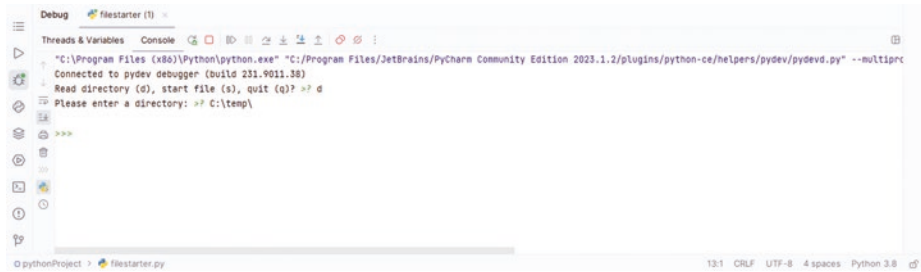
■ Fig. 26.2 *PyCharm* toolbar with debug icon (beetle to the right of arrow icon)

■ Set and Remove Breakpoints

Before you can start debugging properly, you must first set a breakpoint. To do this, click in the margin strip next to the input area in the code editor, where you can also see the line numbers. A red marker will appear to indicate the breakpoint. You can see this in ■ Fig. 26.1. Clicking again in the same place removes the breakpoint again. Of course, you can also set several breakpoints at different positions in the program, if required.

Now you can execute the program in debug mode. This is done by clicking on the beetle icon in the toolbar (■ Fig. 26.2), the same icon on the toolbar attached to the left of the debug console, or choosing “Debug” from the context menu (right click into the code editor), or by pressing the key combination <SHIFT>+<F9>. Note that if you run the program using the normal Run function (for example, using the green arrow button from the toolbar), it will *not* run in debug mode! This means that your program will then not stop at the breakpoint. If you want to debug your program, you must start it in debug mode.

After the program reaches the breakpoint, further execution is first suspended. In the lower part of the *PyCharm* application window appears the *Debug* console. The *Debug* console, which you also see in ■ Fig. 26.3, is ultimately nothing more than a *Run* console with some special functions. Execution always stops *before* the line where the breakpoint is set. You can resume the program by clicking on the *Resume Program* icon in the horizontal toolbar of the *Debug* area, or by pressing the <F9> on your keyboard. The program will then continue to run until the next



■ Fig. 26.3 Debug console in *PyCharm*

breakpoint, or until the end of the program if no other breakpoint stands in the way of program execution.

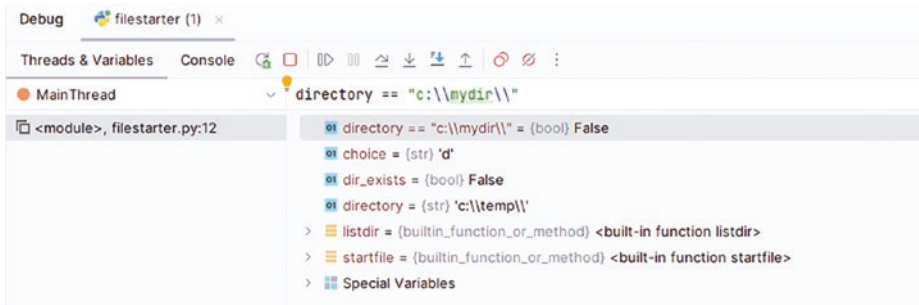
You can stop the program at any time with the red “Stop” symbol, the one in the toolbar of the *Debug* area or the one in the title bar of the *PyCharm* application. Then the program is completely reset, i.e., at the next start in debug mode the program runs again from the beginning and stops again at the first breakpoint reached during execution.

By right-clicking on the red, round breakpoint icon in the margin next to the line of code where you installed the breakpoint, you can add a condition to the breakpoint. For example, in our example from ■ Fig. 26.1, we could enter the condition `isdir(directory)==True`. This way, program execution in debug mode would stop at the breakpoint only if the variable **directory** is indeed a directory. Otherwise, the program would continue to run as if the breakpoint did not exist at all.

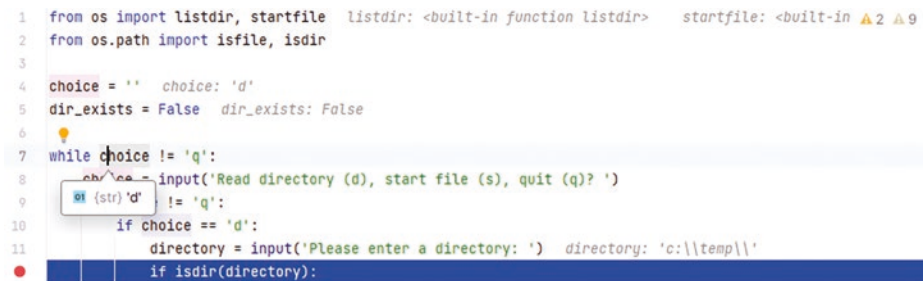
26.2.2 Display of Variable Content and Use of Watches

You will have noticed that when a breakpoint is reached, the *Debug* area switches from the *Console* tab to the tab called *Threads & Variables*. On the right side of this area, you can see the variables used in the program up to this point and their values. The values of variables are also displayed directly in little pop-up windows if you click a variable name in the code editor (■ Fig. 26.4).

However, the debugger does not only allow you to display the contents of variables. You can also formulate entire expressions based on variables and then display their value at the breakpoint. Because you monitor the values of these expressions, they are also referred to as *watches*. Enter the expression whose value you want to observe into the edit box on the top of the right side of the *Threads & Variables* tab in the *Debug* area and press <ENTER>. For example, you may want to check if the variable **directory** contains the directory `c:\mydir`. On the right side a new entry for the expression `directory == "c:\mydir\\"` appears. As soon as you run the program and it reaches the breakpoint in debug mode, the



■ Fig. 26.4 Display of the variable content behind the code lines and in mouseover popup



■ Fig. 26.5 Watches in the *Threads & Variables* tab

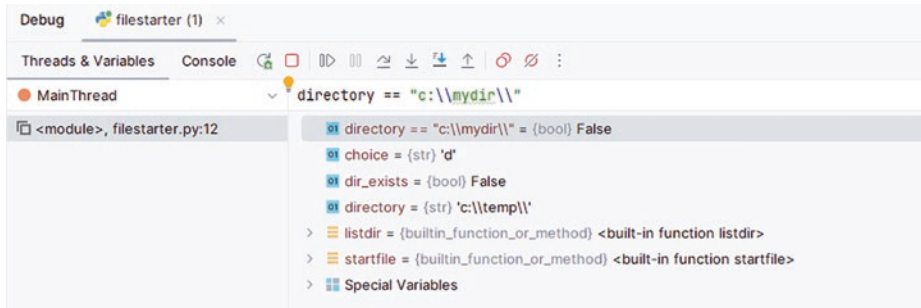
value of the expression will be shown, as you can see in ■ Fig. 26.5. Of course, you can also make use of the Python functions when formulating your watched expressions.

■ Display the Variable Contents Manually

The debugger offers very practical functionalities for monitoring the values of variables or expressions based on them. However, often a much simpler procedure is already sufficient to support debugging by displaying the values of variables and expressions: Outputting the values in the program code using the **print()** function! So just insert **print()** statements at the relevant places in the program for the time being and you will see the corresponding output in the *Run* console. It couldn't be simpler! As with the breakpoints, you can of course use such output to check whether certain parts of your program are being run at all. Just don't forget to remove these output statements after debugging. The user of your program will thank you for it.

26.2.3 Step-by-Step Execution

Especially when working with watches, but also to determine the exact line where an error occurs, a debugger functionality called *step-by-step execution* is helpful. This executes the program line by line, as if a breakpoint were set in each individual line.



■ Fig. 26.6 Execution of the program step by step with Step Over

You can start this as soon as your program has reached a breakpoint. To continue step by step from there, click the “Step Over” button (■ Fig. 26.6) in the toolbar of the *Debug* area or press the <F8> key. This will execute the next line of your program. Clicking the button again or pressing <F8> will execute one more line for each click. In this way, you can step through your program and observe its execution. If you want to exit step-by-step execution, click the red “Stop” icon, or press <CTRL>+<F2>, as you did when using breakpoints.

26.3 Summary

In this chapter, we looked at how Python handles errors *at runtime* by checking critical states in the program and handling exceptions (runtime errors) appropriately, and how to analyze them beforehand *at development time* using debugging tools.

Be sure to take the following points from this chapter:

- By cleverly checking critical conditions, you can make your program more robust against incorrect input or other situations in which regular program execution is no longer guaranteed.
- Runtime errors are represented in Python by exceptions.
- Statements that are in danger of producing an exception can be wrapped in a Try-Except construct; it “tries” to execute the statement and offers the possibility of error handling with its own program code, should the exception be raised (“thrown”).
- The simplest form of the Try-Except construct is: **try: *code_block* except: *code_block***; also, the exception to be caught can be explicitly specified, so that different error-handling routines can be defined for the same segment of code, depending on which exception is thrown.
- Integrated development environments such as *PyCharm* provide some means of error analysis during development; these include breakpoints (where execution of the program code is temporarily halted), watches (displaying the contents of variables while the program is running), and step-by-step execution (where execution of the next statement is explicitly triggered by the developer).

26.4 Solutions to the Exercises

■ Exercise 26.1

A simple formulation of the program might be this:

```
number1 = input("Number 1: ")
op = input("Operator (+,-,*,/): ")
number2= input("Number 2: ")

try:
    if op == "+":
        result = float(number1) + float(number2)
    else:
        if op == "-":
            result = float(number1) - float(number2)
        else:
            if op == "*":
                result = float(number1) * float(number2)
            else:
                result = float(number1) / float(number2)
    print("Result of ", number1, " ", op, " ", number2, ": ",
          result, sep="")
except:
    print("There was an error during the calculation of ",
          number1," ", op, " ", number2, ".", sep = "")
```

The program section that is to be protected from running into an uncontrolled error is simply wrapped in a large Try-Except construct here. If an error occurs during program execution, the user receives a general error message. A more precise description of the error would be possible if the critical program sections, for example the conversion of the number input of type **str** into type **float**, were executed in separate Try-Except constructs. The rest of the program would then be executed in the **except** block of the preceding Try-Except construct, so that in the end you would have a multiple nesting of these constructs, which is not easy to read, but at least offers the user a more precise analysis of the cause of the error.

■ Exercise 26.2

An alternative to using the Try-Except construct—at least in these cases here—is to catch errors explicitly using appropriate conditions. In the following formulation of the program, a number of such conditions are checked. If an error condition hits, a **bool** variable **error** is set to **True**; in addition, a **str** variable **message** is filled with an appropriate error message. Towards the end of the program, it is then checked whether the program has run through without errors (**error == False**); if so, the variable **message** is supplied with the result of the calculation. Finally, only the content of **message** must be output to the user, which then contains either the result of the calculation or an error message.

Note here the trick we use to check whether the variables **number1** and **number2** really contain a number after reading them with **input()**. The **str** method **isdigit()** checks whether an **str** variable contains only digits. Now, of course, the user may have entered a decimal separator (i.e., a period). Therefore we first eliminate it with **replace()**. We also check whether the decimal separator also appears at most once in the string, because we would also have an invalid “number” if the decimal separator appeared more than once in the string.

So, as you can see, you can catch many errors, even with conditions. However, this is not successful with all errors. When working with files, for example, it is difficult to check all conceivable causes of errors with conditions (perhaps also the existence of a file if you want to read from it). In these cases, the use of **try...except** constructs is a good idea.

JavaScript



Introduction

Overview

In this part, we will now turn to JavaScript, the second programming language we will deal with in this book. Again, as with Python, we will follow the 9-question scheme to get to grips with the language. In this first chapter of this part, we will get a brief overview of the origins and application areas of JavaScript.

27

JavaScript is the language of the web. Probably no other language is used more often when it comes to programming web frontends, i.e. modern web user interfaces, than JavaScript. Hardly any site with a high number of visitors can do without JavaScript these days. When you see forms on web pages that validate your input, for example, checking whether the email address or phone number you entered is in a valid format, JavaScript is usually involved. If you see suggestions for possible search terms after entering just a few letters in a search field, JavaScript is usually doing the job in the background. If a web page works with animations, for example showing or hiding elements depending on your clicks or mouse movements, or highlighting them, there is often a JavaScript program in the background. No wonder, then, that JavaScript appears among the most popular programming languages on virtually all rankings (see also ► Chap. 6).

JavaScript has—contrary to what you might think when reading it for the first time—little to do with another popular, but much harder to learn programming language: Java. There are only a few syntactic similarities between the two namesakes.

The beginnings of the language go back to the time when a program called *Netscape Navigator* was the market-dominating browser. In 1995, shortly before the “browser war” between Netscape and Microsoft broke out for dominance in what was then considered a key market, Netscape released a language called *LiveScript*. It had been developed by *Brendan Eich* and was renamed *JavaScript* a little later in the course of a cooperation with Sun Microsystems. It is not entirely clear whether the naming had to do with the fact that JavaScript was actually supposed to work with small Java applications (also called Java applets) on web pages, or whether the naming was mainly done for marketing reasons, in order to bask a little in the glow of Java’s growing popularity; what is certain, however, is that JavaScript quickly rose to become the dominant programming language of the Web and drove competitors such as Microsoft’s *VBScript* from the market.

Netscape made an early effort to have the language standardized and called the standardization organization Ecma (whose name at that time was still a capitalized acronym for *European Computer Manufacturers Association*). As a result, they published the ECMA-262 standard, which defines a language called *ECMAScript*. From then on, JavaScript was considered an implementation of ECMAScript, alongside other languages that also implement the standard (supplemented by their own specifics), such as Adobe’s *ActionScript* or Microsoft’s *TypeScript*.

In the case of JavaScript, however, the discussion about standardization strictly speaking leads in the wrong direction, because JavaScript is anything but standardized. This is mainly because there are several popular implementations of

JavaScript. JavaScript programs run in the web browser, so they are downloaded by it and then interpreted. Each browser that supports JavaScript brings its own interpreter, a JavaScript engine, and these engines certainly differ between vendors. Microsoft's JavaScript dialect called *JScript*, which follows the ECMA-262 standard but brings several specifics, therefore works differently in nuances than, say, the JavaScript variant interpreted by Google's *Chrome* browser. At worst, these differences can mean that certain features of a web page work in one browser but not another. The devil is in the details. Of course, we will hardly encounter the devil in the following chapters, since we are dealing with language elements that are ECMA-262 compliant and are supported practically everywhere JavaScript is interpreted.

JavaScript is a language that is normally interpreted by the web browser, and thus on the *client side*. In a sense, it is the counterpart to *PHP*, which runs on the web server and can modify the web page before it is even sent to the browser (for example, by reading data records, such as product information, from a database and displaying them on the web page). However, there are runtime environments such as *Node.JS* that provide a JavaScript implementation with which JavaScript can also be executed on the server side.

In the following Chapters, however, we will concentrate on the classic variant, namely JavaScript, which runs in a web page and thus makes the web page dynamic. As in the Python part of this book, we again follow the nine major questions to get an overview of the language.



Tools & Help: What Do I Need for Programming?

Contents

- 28.1 Interpreter – 400**
- 28.2 Code Editors and Development Environments – 400**
- 28.3 Assistance and Documentation – 401**
- 28.4 Summary – 402**

Overview

In this chapter, we will look at how JavaScript is executed, what development tools are available, and where to find more information about JavaScript. As you will see, you don't need much to get started with programming JavaScript. To keep things simple, we will therefore only do with the “basic equipment” of tools and not work with a real Integrated Development Environment, as we did with Python.

In this chapter you will learn:

- How JavaScript code is interpreted, and what that means for the tools you need to run JavaScript code
- how to edit JavaScript code
- where you can get help on the internet if you need it

28.1 Interpreter

We had already seen in the introduction to this part of the book that JavaScript is a programming language that is widely used in the development of web pages. It is therefore clear that the execution of JavaScript programs must function somewhat differently than, for example, that of a Python program. After all, the user of a website cannot be expected to download an interpreter before he can view the website. And in fact, this is not necessary at all. All modern browsers come with JavaScript engine that has the ability to interpret JavaScript. That's why you as a developer don't need to install an interpreter. Your browser does the job!

28.2 Code Editors and Development Environments

There are quite a few integrated development environments (IDEs) available on the market that support JavaScript (among other things). These include, for example, JetBrains' *WebStorm* (from the same vendor that provides the *PyCharm* used in the last part of the book) and *NetBeans* from the Apache Software Foundation. Some are commercial (like *WebStorm*), others are available for free (like *NetBeans*).

However, we will take a different approach here: After we worked with a sophisticated IDE for Python in the last part of this book, we will now make do with a much more modest tool for JavaScript, namely a pure code editor. Since our JavaScript code will be embedded in a web page anyway and then executed in the browser, we don't need to bother with the question of how exactly to integrate the interpreter with the code editor. We simply write the JavaScript code and the web page that uses it in our code editor and then look at the result in the web browser.

The popular editor *SublimeText* is used in the following, which, according to the license terms at the time these lines are written, can be used free of charge during (a principally indefinite) trial period for trying it out, but requires the purchase

of a license for long-term use. Alternatively, you can use other editors, such as Microsoft's *Visual Studio Code* or *Notepad++*. At the end of the day, all that's needed is any editor that allows you to edit the actual JavaScript code and the web page it's embedded in, and we'll do everything else in the web browser. So even though you are completely free in choosing an editor, it is recommended that you choose a tool that supports syntax highlighting for JavaScript and HTML, either out-of-the-box or through an appropriate add-in.

If you just want to try JavaScript first and avoid the hassle of having to embed your JavaScript code in a web page as much as possible, then web services like *JS Bin*, *JS Do* or *Plunker* are of interest to you. These services allow you to type JavaScript code directly and execute it without any further hassle. You then see your code and the result of its execution in the same browser window. In the following chapter, we'll take a quick look at these web services. However, if you are serious about learning and working with JavaScript, you should opt for a code editor or a corresponding IDE.

28.3 Assistance and Documentation

As with all popular programming languages, there are countless sources of information on the Internet for JavaScript. However, there is no official, usable documentation for JavaScript. ECMA, which sets the official language standard, provides the language specification (for version 13 of ECMAScript from June 2023, available at ► <https://262.ecma-international.org/13.0/>). However, this is rather less enjoyable for practical work.

Good application-oriented documentation is provided by the Mozilla Foundation (► <https://developer.mozilla.org/en-US/docs/Web/JavaScript>), the organization that succeeded JavaScript inventor Netscape with its *Firefox* browser. Here you can find, among other things, a help page for all standard JavaScript functions with a general description, the arguments with which the function must be called, the return value it provides and some application examples. In addition, the help page also shows an overview of the extent to which version the JavaScript interpreters of the most important browsers such as Internet Explorer, Edge, Chrome, Safari, Firefox or Opera support the function in each case. As an example, take a look at the help page for the `sqrt()` function, which takes the square root of a number: ► https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math/sqrt.

In addition, as with practically all known programming languages, the programmer forum *StackOverflow* is an excellent source of information for solving concrete problems, where you can find qualitatively very good answers for a multitude of frequently (and less frequently) occurring problems and, if this should not be the case, you can open your own question thread.

Beyond that, of course, there are countless articles, tutorials, blogs, videos, and other formats on the internet that—tailored to a wide range of experience levels—highlight every conceivable aspect of the popular JavaScript language.

28.4 Summary

In this chapter, we looked at how JavaScript is interpreted and what tools are needed to write and run JavaScript code.

Be sure to take the following points from this chapter:

- JavaScript is interpreted by the Web Browser; therefore, you do not need a separate interpreter or compiler.
- A text editor is sufficient to edit the JavaScript code (and the web page into which it is integrated). Integrated development environments such as *NetBeans* or *WebStorm* are also available for JavaScript.
- The official documentation of JavaScript is rather limited. The best documentation to use is the documentation of the Mozilla Foundation.
- As with many programming languages, in the case of JavaScript *StackOverflow* is a good place to go if you have questions.
- In addition, there are a lot of “unofficial” sources of information on the Internet, which provide good services for all practical programming problems.



Program Execution & Hello World: What Do I Have to Do to Get a Program Running?

Contents

- 29.1 Embedding JavaScript Code in Web Pages – 404**
 - 29.1.1 The Script Element in HTML – 404
 - 29.1.2 Safety Aspects – 408
- 29.2 “Hello World” in JavaScript – 409**
 - 29.2.1 Do-it-Yourself: The (Not so) Laborious Way – 409
 - 29.2.2 With a Little Help: Fast Implementation with a Web Service – 411
- 29.3 Summary – 412**

Overview

JavaScript programs are typically integrated into web pages. Therefore, to make a program functional, we first need to understand how to incorporate it into a web page. This chapter covers essential HTML fundamentals necessary for this process and illustrates different methods to link an HTML web page with your JavaScript code. Throughout this section, you will also have the opportunity to write your inaugural "Hello World" program in JavaScript.

In this chapter you will learn:

- How to use the HTML element **script** to refer to a JavaScript script from within a web page (if you are not yet familiar with HTML, this chapter also gives you a short introduction to the “programming language of the web”)
- how to embed JavaScript code directly into a web page using the **script** element
- how you can try out JavaScript thanks to special web services without having to worry about embedding your script in the web page
- how to write your first little “Hello World” program in JavaScript

29.1 Embedding JavaScript Code in Web Pages

29.1.1 The Script Element in HTML

Unlike Python, JavaScript does not need an interpreter in order to run as it is embedded in a web page and interpreted together with the web page by the browser. This means that while we don't have to deal with the question of how to run a JavaScript program, because the browser does that for us, we do have to deal with how to integrate a JavaScript program into a web page.

Web pages consist of code written in the descriptive programming language HTML (*Hypertext Markup Language*); for the conceptual distinction between descriptive and other languages, scroll back to ► Sect. 3.1. If you already know the basics of how HTML works (and basics is all we'll need here), then just read on. But if you think your knowledge could do with a quick refresher, then before you read on, turn first to the HTML (Hypertext Markup Language) background box.

HTML (Hypertext Markup Language)

The Language of the Web

HTML is the language in which web pages are written. It was developed at the end of the 1980s by the physicist *Tim Berners-Lee* at the European nuclear research center CERN. Berners-Lee was not only raised to the peerage by the Queen of England, showered with countless other honors (like the *Turing Award*, widely considered as the Noble Prize in computer science) and placed on the list of the *100 Most Important People of the Twentieth Century* by Time Magazine, but is also generally regarded as the father of the World Wide Web. Today, the HTML standard is defined by the *World Wide Web Consortium* (also known as *W3C*).

HTML Elements and Their Components

HTML documents consist of HTML elements. Such HTML elements basically have three components:

- *Tags*: Tags are symbols that mark the beginning and end of an HTML element; to distinguish them from other parts of the document, they are enclosed in “less than” and “greater than” characters, with a slash (*/*) preceding the element name in the closing tag. A combination of opening and closing tags might thus look like this: `<h1>...</h1>`.
- *Content*: There can often be content between the tags. For example, the **h1** element describes a top-level heading (*header 1*). The text of the heading is placed between the opening and closing tags, so for example: `<h1>Introduction</h1>`.
- *Attributes*: Attributes are inside the opening tag and modify the behavior of the tag. They consist of an attribute name and an attribute value. The value of the attribute is written in quotation marks and assigned to the attribute name with an equal sign. The attribute **dir**, for example, controls the direction of the text display. Its value **rtl** means *right-to-left*, i.e., the text is displayed from right to left (right-justified). Incorporated into our **h1** element, the attribute then looks like this: `<h1 dir="rtl">introduction</h1>`.

Structure of HTML Documents

Many HTML elements can be nested within each other. This results in a hierarchical structure that the web browser reads and displays as a *Document Object Model (DOM)*. This already becomes clear in the structure of an HTML document. HTML documents consist of a head (element **head**) and a body (element **body**). The document head usually contains general information about the document, such as the title of the document or meta information such as keywords or a description of the web page for search engines. The body of the document contains the actual content of the web page; it is therefore usually the disproportionately larger part of the document.

head and **body** are embedded in an enclosing **html** element, which is often called root element, because it is the hierarchically highest element in the Document Object Model.

A simple website might look like this:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Test Page</title>
  </head>
  <body>
    <h1>Introduction</h1>
    <p>Here is some text.</p>
  </body>
</html>
```

The **DOCTYPE** element is optional (and is one of those special kinds of elements that don't have a closing tag), it describes the document as an HTML document. In the **body** of the document, you will see another element, **p**. It marks a paragraph of text and usually causes some spacing to be left above and below the text in the display, making the text appear as a free-standing paragraph.

The tab indentations in the source code of the HTML document have only been added for the sake of better readability, they are ignored in the interpretation of the document content, just like the upper and lower case of the tags (HTML is not case sensitive). Thus, the following code is also a valid web page:

```
<!DOCTYPEhtml><html><head><TITLE>Test Page</title></head><body><h1>Introduction</h1><p>Here is some text.</p></body></HTML>
```

Paste this code (preferably with the indentations) into an empty file using an editor, save it with the extension **.htm** or **.html**, and there's your little web page, which you can now open and view with your browser.

By the way: HTML also has *comments*: They have special opening and closing tags with **<!--** and **-->**. Comments may also be multiline.

Cascading Style Sheets (CSS)

In addition to the HTML source code of web pages and scripts, especially JavaScript programs, which we will deal with in this part of the book, *Cascading Style Sheets* (CSS) are an important part of modern web pages. It used to be common to specify formatting with HTML elements and attributes directly in the HTML source code of the web page; for example, a text can be easily centered with the **align** attribute, which is supported by the **p** element, among others: **<p align="center"><Here is some centered text</p>**. But if a web page wanted to change its design fundamentally or if a "section" from a web page was to be shown on another web page that used a different design, the formatting had to be changed in painstaking and error-prone detail work in the HTML code.

This became easier with the introduction of *Cascading Style Sheets*. The idea behind Cascading Style Sheets is to separate the structure and content of a web page from the design and formatting, so that the design can be more easily replaced or adapted without having to dig deep into the part of the web page that houses the structure and content. This is implemented by describing formatting and design in a separate file (typically with the extension **.css**) and not in the HTML file itself. In the style sheet, i.e., in the CSS file, it can be specified, for example, that **h1** headings should always be displayed in blue and in bold. If the design of the website is revised and the default headline color is changed to red, this change only must be made in one place, namely in the style sheet. The display of all **h1** headlines in the entire HTML document will then change automatically. A related advantage is of course obvious. The same CSS file can be used for many different web pages, a large site with hundreds of subpages can thus very easily adapt its design to new needs.

Sometimes, however, you don't want to format *all* headings of type **h1** in a fixed way, but only certain ones. For this purpose, there is the **class** attribute, which can be used with practically all HTML elements and which you often see when you look at the source code of a web page. For example, a heading might be distinguished like this: `<h1 class="section">News</h1>`. This attribute can now be referred to in the CSS code. This means that some formatting rules stored in the CSS only apply to this class of **h1** elements, but not to other **h1** elements. The CSS could contain the following code for formatting the **h1** heading elements:

```
h1.section {
    color: red;
}
h1 {
    color: blue;
    font-weight: bold;
}
```

Here it also becomes clear what the term *cascading* means. Property attributes are *inherited*. The general **h1** formatting statement in CSS says that the text color should be blue. The formatting for **h1** elements of the class **section** differs from this. For them, a different statement applies, namely coloring in red. The statement **font-weight: bold** from the CSS of the **h1** element, on the other hand, is not overwritten by the special statements for **h1** elements of the class **section**, and therefore also applies there. Thus, an **h1** element of the class **section**, like all **h1** elements, is also displayed in bold.

Like the HTML standard, the CSS standard is managed and developed by the World Wide Web Consortium.

To get a JavaScript program to work, we need to embed it in a web page. This is done with the **script** element, the use of which you can see in the following example:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Test Page</title>
  </head>
  <body>
    <script src="helloworld.js"></script>
  </body>
</html>
```

The name of the script file is specified as the value for the **src** (*source*) attribute.

In the example, we have placed the script element in the body of the web page, but we could also have placed it in the head. However, scripts are usually included in the body, or more precisely: at the end of the body. There is a good reason for this:

scripts are executed in the HTML document where they are placed. If your script is at the end of the web page and needs a little longer to be fully executed—no problem. The rest of the web page is already loaded and displayed at this point. It would be more unpleasant if the viewer had to wait in front of an empty page while the JavaScript code runs, which was included at the beginning of the HTML document and is therefore also executed first, even before any other elements of the page are loaded.

We have included our JavaScript program code here *as an extra file*. This is the usual procedure. However, you can also include (especially short) scripts *directly* in the HTML file. This would then look like this:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Test Page</title>
  </head>
  <body>
    <script type="text/JavaScript">
      console.log("This goes to the console!")
    </script>
  </body>
</html>
```

29

With the attribute **type** of the **script** element, we tell the HTML engine of the browser that the code between the opening and closing tag (in our example only a single line) should be understood as JavaScript code. De facto, JavaScript is the only scripting language used. In the past, however, you might have encountered Microsoft's VBScript here and there, whose use in practice was admittedly quite limited because it was only supported by Microsoft's own browsers and had a powerful competitor in JavaScript.

29.1.2 Safety Aspects

■ Enable Execution of Scripts in the Browser

While many of the more popular websites use JavaScript in one form or another, they do not work or do not work properly if JavaScript code execution is disabled. However, all modern browsers allow you to disable the execution of script code. So, take a look at your browser's settings and see if anything could get in the way of enjoying your first JavaScript program.

When the functionality of a website depends on the ability to run JavaScript, you naturally want to tell the user of the page to enable JavaScript. This can easily be done in the HTML source code of the page by using the **noscript** element and giving it a message to the user as a hint, as in the following example:

```
<noscript>Please activate JavaScript to use this website.
</noscript>
```

The message is only displayed if JavaScript execution is disabled in the browser’s security settings (try it out!).

■ Visibility of the Source Code

Unlike PHP, JavaScript programs are interpreted by the browser (i.e., client-side). To do this, they must first be downloaded, which is not necessary in the case of PHP code, since its interpreter operates entirely on the server and does not use the client’s resources for execution. Therefore, you cannot hide your source code from the user’s eyes, even if you dump it into an extra file that you load from a web server with `<script src=...>`. As you will see in the following section, it is usually easy to see the code. True, there are methods and tools for obfuscation, with which the source code can be practically encrypted. But even these approaches have their limits because the browser has to decode the source code in order to execute it. So, expect your JavaScript programs to be viewable by your site’s viewers. This may not be true for the average user, but those who have the knowledge to understand the JavaScript code will know how to access it.

29.2 “Hello World” in JavaScript

29.2.1 Do-it-Yourself: The (Not so) Laborious Way

The classic “hello world” program in JavaScript is very simple:

```
console.log("Hello World!")
```

If you save this line in a JavaScript file **helloworld.js** and then include this script in an otherwise empty (except for the mandatory elements) HTML web page document, as seen in the example of the previous section, you have already successfully completed this task.

Now when you open the finished HTML document in your browser, you see... nothing!

Didn’t our simple program work? Yes, it did, it just doesn’t output the text in the browser window, but in a different area, the *JavaScript Console*. How you get to this console, which is not displayed by default and is more of a tool for developers, differs from browser to browser: On Windows, for example, press `<CTRL>+<SHIFT>+<J>` in Google *Chrome*, `<F12>` in Microsoft *Edge*, and `<CTRL>+<SHIFT>+<I>` (or `<F12>`) in Mozilla *Firefox*. You can also open the console via the menus, often under menu items like “More tools” or “Developer tools”. If you have more than one browser installed and open the console once in all browsers, you will immediately recognize certain similarities.

The first two tabs, which we will mainly work with here, usually have names like “Elements”, “Inspector” or similar and “Console”. Under “Elements” you see the HTML code of the currently loaded page and can navigate interactively in this HTML code. By marking an element in the HTML source code, the representation

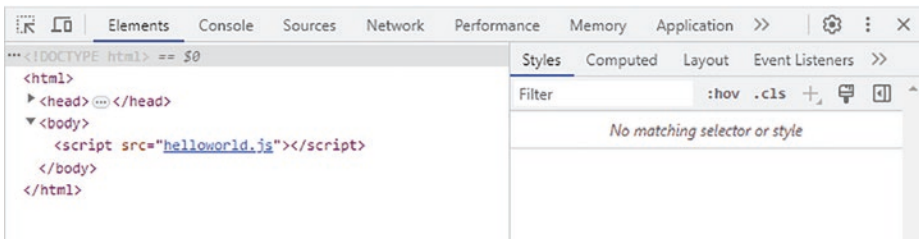
of the element is automatically highlighted in the web page view. This makes it easy to see which part of the source code leads to which part of the web page display. By expanding and collapsing the elements in the source code, you can also display the source code more compactly and better understand the hierarchical nesting of the HTML elements. This may not be very impressive for our small sample page (■ Fig. 29.1) but open the Elements view when you have another website open, such as Wikipedia or YouTube! Then you will see very nicely how practical this function is. Conversely, you can switch from *Elements* to a selection mode where you click on an element of the browser's web page view and then the corresponding location in the page's source code is automatically highlighted in the “Elements” view. For more on this useful feature, see ► Sect. 32.6.2.

The JavaScript programs that are swapped out to other files can also be viewed in this way. If you click on a reference to a swapped-out JavaScript file in the HTML code, the source code immediately opens in another tab. By the way: You can even edit the HTML source code via the *Elements* tab, mostly by using corresponding options in the context menu. You can regularly save the changes as an HTML file with the key combination <CTRL>+<S>.

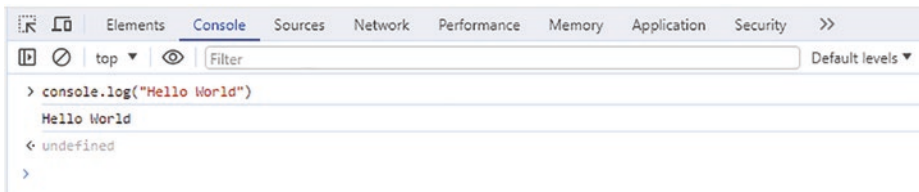
For our “Hello World” JavaScript program, however, we are primarily interested in the second tab, the *Console*. If you open it, you will see that our JavaScript code has indeed produced an output (■ Fig. 29.2).

This means that we can use the console to print outputs that the viewer of the website does not see immediately. This is especially useful in development to output debug messages that do not affect the actual display of the web page in this way. But also the JavaScript interpreter of the browser uses the console to output error or warning messages.

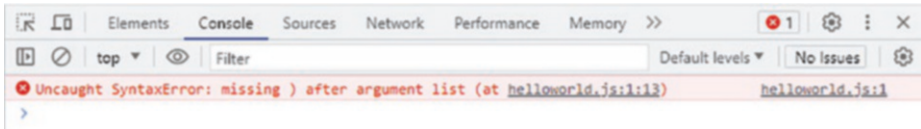
29



■ Fig. 29.1 Developer tools in Google *Chrome*



■ Fig. 29.2 Output of the “Hello World” program in the console



■ Fig. 29.3 Error message in the JavaScript console of Google *Chrome*

In the example of ■ Fig. 29.3, the first quotation mark was forgotten in the source code line `console.log(Hello world!)`. The interpreter points out the syntax error to us when executing the code.

You can also use the console in an interactive mode, that is, you can enter and execute statements directly in the console. Unlike Python, however, the console in JavaScript does not have a separate namespace from the web page. If you execute a program and create variables, these variables are then available in the console for further interactive editing. Most web browsers also support code completion in the console. When you start typing, you will be presented with a pop-up menu with several choices that can help you speed up the typing process.

We will work quite intensively with the console in the next chapter because it offers a simple way to quickly output some information, and the next chapter will focus precisely not on inputs and outputs, but on other concepts to which we want to devote our full attention. In ► Sect. 31.7, where we deal specifically with the input and output of data, we will also see how you can write data directly into the web page.

If you can't wait, and think a “hello world” program that doesn't write directly to the web page isn't a real “hello world” program, try this code in your script file:

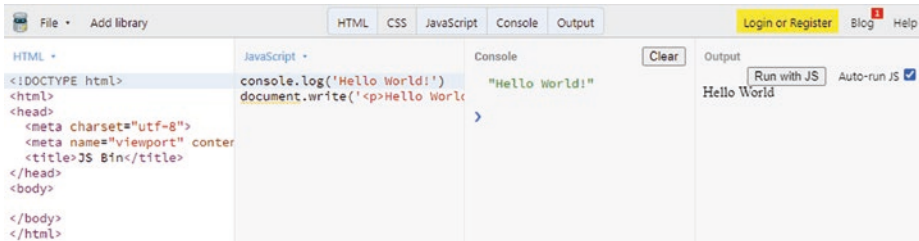
```
document.write('<p>Hello World</p>');
```

After refreshing the view (typically with F5), you see the output in the browser, not in the console, but in the actual page view.

29.2.2 With a Little Help: Fast Implementation with a Web Service

It's not that easy to just start programming in JavaScript, because first of all you have to create a web page and integrate your script into it. It's easier with web services like the open-source project *JS Bin* (► <http://www.jsbin.com>), which allow you to get started at will without any fuss. *JS Bin* executes the JavaScript code you type in the appropriate pane at the end of the HTML body (but without displaying a reference to your script in the HTML code). You can also edit the HTML code manually. At the top of the screen you can use the toggle buttons to determine which views are displayed; the *Output* is the finished website (■ Fig. 29.4).

Tip: If you are working with *JS Bin*, it is recommended to uncheck the *Auto-run JS* checkbox on the right side of the screen and to always start the execution of the



■ Fig. 29.4 The JavaScript web service JS Bin

JavaScript manually via the *Run with JS* button instead. Otherwise, *JS Bin* will re-execute the current state of your JavaScript code every time you hit a key, which of course leads to an unpleasantly long list of useless error messages in the console.

In addition to *JS Bin*, there are a number of other services that offer similar functionalities, such as *Plunker* (► <http://plnkr.co>) or *JS.do* (► <http://www.js.do>), although the structure is somewhat different in each case. What all these services have in common is that they are well suited for trying things out. However, if you want to get serious about JavaScript, it is advisable to work offline with your own “infrastructure” (HTML and JavaScript code files). And that is exactly what we will do in the following chapters.

29.3 Summary

In this chapter, we looked at how to get a JavaScript program running.

Be sure to take the following points from this chapter:

- JavaScript programs typically run in web pages, so they must be integrated into them.
- This is done with the help of the HTML element **script**; this can either be given a reference to a script file with JavaScript with the attribute **src**, or the JavaScript code is embedded directly into the web page between the **<script>...</script>** tags.
- It is good practice to include scripts *at the end of* the web page so that the web page loads first and can be viewed while the script is still loading or running, if applicable.
- The HTML element **noscript** can be used to display a message if the user has disabled JavaScript in his browser (normally JavaScript is already enabled thanks to corresponding browser default settings).
- JavaScript code is always visible to the viewer of the website. So, you cannot “hide” your code.
- Web services like *JS Bin* or *Plunker* allow you to try out JavaScript without having to worry about integrating it into a web page; that’s handy for quick testing, but if you want to get serious about JavaScript, you can’t avoid learning how to integrate scripts into web pages (which is really not complicated).



Syntax, Comments, Code Style & Documentation: How Do I Make Sure That I (And Others) Still Understand My Program Later?

Contents

- 30.1 Design of the Program Code and Naming Conventions – 414
- 30.2 Comments – 416
- 30.3 Summary – 417

Overview

Before we really get into programming with JavaScript, we want to prepare ourselves with some important questions about the design of the program code and—you will have guessed it already—the commenting of the code.

In this chapter you will learn:

- How identifiers for variables, functions and other objects are constructed
- how instructions are completed
- what style guides for the design of your JavaScript code are available for your orientation
- what types of comments you can use in JavaScript

30.1 Design of the Program Code and Naming Conventions

■ Identifiers

JavaScript is case-sensitive, i.e., it distinguishes between upper and lower case. Identifiers like those of variables, objects, and functions, may contain letters, digits, underscores (`_`), and dollar signs (`$`), but may not begin with a digit. In JavaScript, identifiers are encoded according to the Unicode scheme, which means that umlauts and other non-English letters are also allowed in identifiers. However, since these characters do not occur in most natural languages, it is recommended to do without them. If you were to post sample code on an internet forum to ask for help, the international audience might have trouble reproducing these characters if they wanted to edit your code. Think of how the situation would be if an Icelander named a variable `húsið1` (meaning **house1**), and you wanted to edit that code.

The first term in an identifier often starts with a lowercase letter in JavaScript, and the following terms then start with an uppercase letter (even if they are terms that are actually lowercase in natural language). A typical JavaScript identifier would be, for example, `isCustomerActive` or `startSave()`. This kind of writing, also called *camel case* because of the “humps” in the identifiers, may be unusual at first, but makes your program look distinctly “javascript-y” in the eyes of JavaScript programmers.

■ End of Statement and Statement Breaks

Statements in JavaScript *can* be terminated with a semicolon. If several statements are to be on one line, they *must* be separated by semicolons. Unlike some other languages, however, not every statement in JavaScript automatically ends at the end of a line. Instead, JavaScript checks without a semicolon at the end of the line whether the statement would make sense if it were continued on the following line. This strange feature can lead to unexpected and seemingly inexplicable behavior in individual cases. Therefore, we will always end statements with a semicolon here—knowing that some style guidelines for JavaScript recommend otherwise. The fact

that we end statements with semicolons does not apply to those statements we execute in the JavaScript console, where the end of the statement is clearly marked by pressing the <RETURN> or <ENTER> key.

Statements can be wrapped, i.e., continued on the following line. However, line breaks *within strings* are not permitted. For example, the following wrapped output statement for the JavaScript console is syntactically correct:

```
> console.log('Hello',  
  'World')
```

The following would also be permissible:

```
> console.log('Hello'  
  , 'world')
```

Since strings in JavaScript can also be joined with the plus operator, you could also write:

```
> console.log('Hello'  
  + 'world')
```

However, the next output statement would not be allowed because it contains a break within a character string:

```
> console.log('Hello  
World')
```

If you absolutely want to achieve a break within the string, you can create it with the backslash ("\"), which will not be output itself:

```
> console.log('Hello \  
World')
```

If you wrap, it is recommended that you indent the second line accordingly. In the book, program code is often wrapped to fit the page width. The program codes that you can download with the book are of course not subject to this restriction and therefore have less wrapping.

■ General Code Style

As with most other programming languages, smart developers have thought about how to format their JavaScript code so that it is clear and readable. The result is the style guides, of which there are several for JavaScript. They may well give quite different recommendations on one and the same topic, such as the question of the use of semicolons when closing statements.

Well-known JavaScript style guides include the *Google JavaScript Style Guide* (► <https://google.github.io/styleguide/jsguide.html>), the one from Airbnb (► <https://github.com/airbnb/javascript>), and the *JavaScript Standard Style Guide* (► <https://github.com/standard/standard>), which—contrary to its name—by no means takes on an official role, as, for example, Python’s *Python Enhancement Proposal (PEP)* number 8 does. For some style guides, including the Standard Style Guide, there is even software, a *linter*, which can format the source code fully automatically so that it conforms to the style guide’s specifications.

We won’t follow any of these style guides in their entirety in this part of the book. Except in a professional environment, where you are working with a number of other developers, it is certainly not necessary to implement a style guide 100%. Nevertheless, it’s a good idea to browse through some of the style guides and maybe pick up a thing or two.

30.2 Comments

JavaScript knows both single-line and multi-line comments. Single-line comments are introduced by `//` and identify everything to the right of it as a comment that is not processed by the interpreter. Multiline comments start with `/*` and are terminated with `*/` (so the asterisks always face the comment text).

Examples of comments could look like this:

```
/*
    Here comes a multi-line comment explaining
    why the assignment a = 5 is necessary
*/
a = 5;
// This assignment can be explained in a one-line comment
b = 10;
c = 7; // c must also be initialized with a meaningful value
```

The last comment is an *inline comment*, it is also syntactically permissible, but not appreciated because it is hard to read. If you use inline comments, make sure you leave some space (at least two spaces) between the end of the statement and the comment symbol, so that the comment stands out visually from the executable code.

30.3 Summary

In this chapter we have dealt with basic issues of JavaScript code design and commenting.

Be sure to take the following points from this chapter:

- Identifiers in JavaScript can contain letters, digits, the underscore character, and the dollar sign, but cannot begin with a digit.
- Identifiers are UTF-8 encoded and may therefore contain umlauts and other non-standard characters; however, their use is not recommended.
- Typically, identifiers are written in lower case. In the case of identifiers composed of several terms, the following terms are each capitalized (“camel case”).
- Semicolons at the end of a statement are not mandatory but recommended since JavaScript does not always automatically stop interpreting a statement at the end of a line.
- Line breaks in code are allowed as long as they do not occur within a string.
- For the design of the code, which is an area of relative freedom in JavaScript, there are numerous guidelines that represent quite different views on various topics, including the *Google JavaScript Style Guide*, the *JavaScript Standard Style Guide* and the style guide of Airbnb.
- JavaScript knows single and multiline comments.
- Single-line comments start with `//` and mark everything to the right of it as a comment; they can also be used within a line of code (“inline”).
- Multiline comments are placed between `/*` and `*/`.



Variables & Objects: How Do I Store Data to Work With?

Contents

- 31.1 Declaration of Variables – 420**
- 31.2 Elementary Data Types – 421**
 - 31.2.1 Numbers (number) – 421
 - 31.2.2 Text (string) – 423
 - 31.2.3 Logical Values (boolean) – 427
 - 31.2.4 Special Types and Values (null, undefined, NaN) – 428
- 31.3 Converting Variables – 430**
 - 31.3.1 Implicit Conversion – 430
- 31.4 Arrays – 433**
- 31.5 Objects – 441**
 - 31.5.1 Object Orientation in JavaScript – 441
 - 31.5.2 Creating Objects Directly – 442
 - 31.5.3 Accessing Properties of Objects – 443
 - 31.5.4 Creating Objects Using the Object Constructor – 444
 - 31.5.5 Creating Objects Using Constructor Functions – 444
 - 31.5.6 JSON – 445
- 31.6 Summary – 446**
- 31.7 Solutions to the Exercises – 448**

Overview

Now it's time to look at how we work with variables and objects in JavaScript to be able to temporarily store data for processing. That's what this chapter is about.

In this chapter you will learn:

- the elementary data types of JavaScript (called primitives)
- how to work with variables of elementary data types (create them, assign them, convert them, ...)
- how to create fields (arrays) from any data type and how to work with them
- what distinguishes elementary data types from real JavaScript objects
- how object orientation works differently in JavaScript than in other object-oriented languages
- how to create and edit objects in JavaScript
- what JSON is, why it's so important, and how you can turn JavaScript objects into JSON, and vice versa

31.1 Declaration of Variables

Variables *should be declared* in JavaScript, that is, registered before they are used for the first time. It is good practice to declare variables, although a simple value assignment is regularly sufficient to create a variable and a formal declaration is strictly speaking not required. However, you will get an error message if you try to access a variable that has neither been declared nor assigned a value beforehand.

A common problem, if variable declarations are not mandatory, is that you accidentally create a new variable in the program by mistyping. For example, if you want to assign a value to an existing variable **amountInvoice**, but unintentionally write **amountIvoice** in the assignment, you will have a problem at the latest when you continue working with the originally intended variable identifier **amountInvoice**, which, however, never received the value supposedly assigned to it. Instead, the newly created variable **amountIvoice** holds the value. You can avoid such difficulties by running JavaScript in *strict mode*, in which case undeclared variables lead to an error message. Strict mode can be switched on for an entire script (or a single function) by introducing a special statement as the first statement in the script (or function):

```
'use strict';
```

Try it and assign a value to a variable that you have not declared before in your script; you will immediately get an error message. Now remove the **'use strict'** statement and run the script again—no problem, the variable is simply created on the first assignment.

The formal declaration of variables in JavaScript is done with the help of the keyword **var**, which is followed by one or several variable identifiers, separated by commas. The variables can, but do not have to be initialized with a value during the declaration.

```
var x = 0.5, y, z = 'A message';
```

It is noticeable that the variables are created without type specification during the declaration. JavaScript automatically recognizes which type it must be based on the value assigned during the declaration (if you initialize the variable) or later. Programmers say JavaScript is weakly typed). Of course, we can, and sometimes even must, explicitly specify the types of variables by conversion. We will deal with this in ► Sect. 31.3. The type of a variable can also be changed by simply assigning a value of a different type to the variable. JavaScript then automatically adjusts the type of the variable to accommodate the new value (dynamic typing). The keyword **var** can occur several times in the program and does not necessarily have to be at the beginning of the entire program source code. However, it is good practice to merge variable declarations at the beginning to keep track of the registered variables.

31.2 Elementary Data Types

JavaScript has several elementary data types, including numbers (**number**), strings (**string**), and logical values (**boolean**). These (and some special other) types are *elementary* in the sense that they are *not objects* themselves, but *primitives*. Variables of all other types are objects in JavaScript and therefore have attributes and methods to manipulate them. The elementary data types do not, but nevertheless sometimes—as we will see—behave almost miraculously as if they were “real” objects.

31.2.1 Numbers (**number**)

■ Integers and Floating-Point Numbers

number is the data type that holds numbers, regardless of whether they are integers or fractional, i.e., floating-point numbers. JavaScript always reserves 64 bits, i.e., 8 bytes, for storing a number. The range of numbers that a **number** variable can hold therefore depends on how high the precision is to be, i.e., how much memory is required for storing the decimal places. Conversely, the possible precision depends on the size of the integer part before the decimal point: If this is very large, less memory remains for the decimal places.

■ Infinity

A special value that a **number** variable can assume is **Infinity**. If you divide a number by the value 0 in JavaScript, you will not receive an error message—unlike in many other languages—but the value **Infinity** or **-Infinity**. Open the JavaScript console and enter the following code (the greater-than sign here represents the console prompt and must therefore not be entered):

```
> 1/0
Infinity
```

■ Operators

Of course, you can perform the usual arithmetic operations with numbers, including the four basic arithmetic operations. In addition, `%` is a *modulo operator* that returns the integer remainder of a division. All these operators are *binary* operators that process two values into a new value. However, there are also several *unary* operators that are applied to a single value. Of particular interest in this context are the *increment* (`++`) and *decrement* (`--`) operators often found in JavaScript programs, which increment or decrement a **number** variable by the value 1 regardless of whether the variable has a decimal part other than 0 or not. So, `variable++` is a more compact notation for the assignment `variable = variable + 1`.

■ Methods of Number Objects

number variables have (apparently) callable methods, albeit in a manageable number. “Apparently” because a **number** variable is actually a *primitive*, i.e., a variable of an elementary data type, and thus *not* an object, i.e., an instance of a class, for which we would expect the existence of properties and methods. In ► Sect. 31.3.2 we will see how this apparently contradictory behavior comes about.

Among the methods available for **number** variables, one example is **toExponential()**, a method that converts the number to an exponential notation and returns it as a string. The variable itself is not changed in the process.

```
> myNumber = 50000
> scientific = myNumber.toExponential()
> scientific
"5e+4"

> typeof myNumber
"number"

> typeof scientific
"string"
```

Here you first see that we can access the methods (and properties) of objects using the dot operator (just like in Python), and that calling a method always requires the parentheses to be specified, even if no arguments are passed to the method at all.

With **typeof** *objectinstance*, you're about to learn another useful operator that returns the type of a variable as a string. It's not a mathematical operator but still a (unary) operator. As you can see, **toExponential()** returns a string containing the formatted number.

Another handy method besides **toExponential()** that can be applied to **number** is **toFixed(*places*)**, which rounds the number to the specified number of decimal places, and again returns the result as a string:

```
> myNumber = 3.14159
> myNumber.toFixed(3)
"3.142"
```

Constant values (*literals*) also behave like objects in JavaScript. However, to access the methods of these objects, you need to write the value in round brackets, as in the following example, which you can easily try out in the console:

```
> (3.14159).toFixed(3)
"3.142"
```

? 31.1 [3 min]

Show that in JavaScript infinity plus one is still infinity.

31.2.2 Text (string)

■ Assign Strings

Strings are delimited by single or double quotes:

```
var message = "Hello world", message2 = 'Hello world again';
console.log(message, message2);
```

Due to the fact that in JavaScript, similar to Python, two different string delimiters are available, one of them can be used to delimit the string, the other one within the string can be used for “quotes”:

```
var message = "The nerd shouted loudly: 'Hello world'";
```


■ Escaping

Sometimes, however, within a string, you want the character you use to mark the beginning and end of the string to be part of the string. In this case, *escaping* must be used. As we have already seen earlier, the character that is to have a different function within the string than it would normally have is preceded by a backslash (\). In our example, we could escape the double quote as follows:

```
var message = "The nerd called out loudly: \"Hello world\"";
```

By prepending the backslash, the quote character is no longer interpreted as the character it is in JavaScript programs, namely a delimiter for strings. Instead, it is considered part of the string. The last quotation mark is not escaped and therefore actually marks the end.

However, escaping can not only be used to disable the normal control function of certain characters in order to include them as a constituent part of the string; escaping can also be used to give characters that would normally be inconspicuous letters in the string a different function, thereby turning them into a control statement. Consider the following example:

```
> console.log('Some things need to be looked ',  
  'at \nwith a bit of \t distance.');
```

Some things you need to be looked at
with a bit of distance.

31

Again, we see two backslashes used for escaping: Once the letter **n** is escaped; **\n** causes a new line to be inserted into the string at this point. **\t** causes a tab jump to be inserted. In both cases, a letter that would otherwise have no special function and would be a normal part of the string is converted into a control statement.

But what if you use a backslash to escape a character that has no special control function at all, for example, the **i**. The good news is that nothing happens:

```
> console.log('Escape the \i but don't know why.');
```

Escape the i but don't know why.

However, the backslash is not displayed, it is a special control instruction that causes the next character get a special meaning (if it has one). This logically brings us to the next question, namely how a backslash can be represented in a string. After all, programmers working under Windows use the backslash as a separator for path specifications. The solution is simple: The backslash itself is simply escaped:

```
> console.log('The file is located in C:\\Windows\\System.');
```

The file is located in C:\Windows\System.

■ Work with Individual Characters

The individual characters of a string can be accessed in JavaScript like the elements of a (read-only) field/array. The indexing of arrays in JavaScript starts with the index 0 for the first character. The index is enclosed in square brackets when accessed:

```
> var message = 'Hello world'
> message[1]
"e"

> typeof message[1]
"string"
```

As can be easily verified, a character from a string is in turn a string. There is no special data type for single characters in JavaScript. The array containing the characters of the string is read-only, so it cannot be written to. Therefore, you *cannot* replace a character in the string with a statement like **message[2] = 'z'**. To do this, you must work with the methods available for variables of type **string**, which we will learn about below. We will look at the details of working with arrays in more detail in ► Sect. 31.4.

■ Concatenate Strings

Although strings cannot be computed like numbers, **string** variables also support the plus operator (+). In the context of strings, it is used to concatenate several strings:

```
var message = "Hello" + " " + 'World';
console.log(message);
```

As you can see, it doesn't matter whether single or double quotes are used to delimit the strings.

Unlike the plus, the other arithmetic operators cannot be applied to strings. If you do so, you will not get an error message, but you will get the return value **NaN** (*not a number*), which we will look at in more detail in ► Sect. 31.2.4. It signals that we have applied an operator designed to work with numbers to something that is not a number (try it out!).

■ Methods and Properties of Strings

If you create a **string** variable in the JavaScript console and then enter its name followed by a dot operator, the popup menu opens with the list of methods and properties available for strings.

A particularly important property is **length**, the length of the string:

```
> var message = 'Hello World'
> message.length
10
```

So if you want to access the individual characters of the string, you can use **message[message.length-1]** to grab the last character of the string (because indexing starts at 0!).

Instead of accessing a single character via the array of characters, you could also use the **charAt(index)** method of the string:

```
> var message = 'Hello world '
> message.charAt(message.length-1);
"t"
```

Besides **charAt()**, **string** offers a number of other handy methods:

- **indexOf(searchString, fromPosition)**: Searches for **searchString** in the string for which the method is called, starting at index **fromPosition** at the earliest; here, **fromPosition** is an optional argument, so it can be omitted, resulting in a search starting at the beginning of the string. The return value **indexOf** is the index of the beginning of the **searchString** within the searched string, or -1 if **searchString** was not found in the searched string:

```
> message.indexOf("el")
2

> message.indexOf("tel")
-1
```

- **replace(search, replaceBy)**: Searches the string for **search** and replaces each occurrence with the string **replaceBy**:

```
> message.replace("Hello", "Hi")
"Hi word"
```

- **toUpperCase(), toLowerCase():** Converts the string to upper and lower case respectively:

```
> message.toUpperCase(  
"HELLO WORLD"  
> message.toLowerCase();  
"hello world"
```

- **trim(), trimLeft, trimRight():** Removes leading and trailing, leading-only, and trailing-only whitespace from a string, respectively:

```
> message = " " + message + " "  
" Hello World "  
  
> message.trim();  
"Hello World"  
  
> message.trimLeft()  
"Hello World "  
  
> message.trimRight()  
" Hello World"
```

When using these methods, please note that the variable for which they are called does not undergo any change, but a new, changed string is created and returned by the methods. So, you have to take care yourself to catch this return value.

31.2.3 Logical Values (boolean)

The third important elementary data type in JavaScript are logical/truth values. They can only take the value constants **true** and **false** (note lower case!) and are internally coded with the values 1 for **true** and 0 for **false**. This becomes clear when you calculate with the constants:

```
> true * 5  
5  
  
> false - 1  
-1
```

31.2.4 Special Types and Values (null, undefined, NaN)

■ null

JavaScript knows two special values that can signal that a variable has no value: **undefined** and **null**.

If a variable deliberately has no real value—for example, the answer to a choice question in a questionnaire that the respondent deliberately omitted when answering—the (quasi-) “value” **null** is used:

```
questionAge = null;
```

null is thus a special value constant that indicates that a variable is deliberately “empty”, that is, it does not currently hold a “real” value. If you assign **null** to a variable that previously held a “real” value, it changes its object type to the general **object**:

```
> var mynumber = 5
> typeof mynumber
"number"

> mynumber = null
> typeof mynumber
"object"
```

31

Don’t let this confuse you: **null** is a variable of an elementary data type with the same name. The fact that **typeof** still returns “**object**” is strictly speaking wrong and stems from the historical implementation of the **typeof()** function.

■ undefined

undefined is similar to **null** in that it also indicates that a variable has no real value. However, **undefined** does not mean “deliberately empty”, but rather “untouched” or “not yet assigned another value”. Variables carry the value **undefined** after initialization, as can be easily checked:

```
> var value
> console.log(value)
undefined
```

undefined is not only used in JavaScript as the “content” of currently uninitialized variables, but also in other contexts where something is “missing” without someone having consciously decided to use an “empty” content. This is particularly the case for return values of functions that do not return a real value.

Technically, **undefined**, like **null**, is a real variable of the elementary type of the same name, but of which there is only this one variable:

```
> typeof undefined
"undefined"
```

■ NaN

NaN is the abbreviation for *Not a Number* and expresses as a special value that a numeric variable does not contain a valid numeric value, although it should. Often **NaN** is returned as the result of improper arithmetic operations, such as taking the square root of a negative number:

```
> Math.sqrt(-1)
NaN
```

Interestingly, **NaN** is not a real variable of an independent type like **undefined** is, but simply a special value of numeric variables, as you can see in the following code:

```
> var x = Math.sqrt(-10)
> typeof x
"number"
```

You can't calculate with the value **NaN**, as every calculation operation applied to it returns **NaN** again:

```
> NaN + 3
NaN
```

With **isNaN(*expression*)** a special function is available to check whether an expression (variable, calculation rule, function call or similar) has the value **NaN**:

```
> isNaN(Math.sqrt(-100))
true

> isNaN(Math.sqrt(100))
false
```

31.3 Converting Variables

31.3.1 Implicit Conversion

We have already seen an example of implicit type conversion, i.e., type conversion not explicitly instructed by the developer, in ► Sect. 31.2.3, namely the fact that **boolean** variables behave like the values 1 (for **true**) and 0 (for **false**) in calculations. So, in this case, JavaScript automatically converted the type when it seemed appropriate.

Implicit type conversions play an important role in practice in connection with strings.

```
> x=3; y="4"; z="5"
> x * y
12

> y * z
20
```

As you can see, JavaScript performs the calculations without any problems, even though the variables **y** and **z** are actually strings. For the calculation, the values are implicitly converted into numbers (type **number**).

That this works so well here is mainly because the applied (arithmetic) operator has no meaning at all when working with strings. The situation is quite different, however, when we perform an addition:

```
> x + y
"34"

> y + z
"45"
```

In this case, JavaScript assumes that the “target type” of the operation should be **string**. Therefore, the **+** operator is interpreted as a string concatenation operator and the value of the **number** variable **x** is converted to a string for the purpose of concatenation; **y** and **z** are already strings by default, they do not require any further conversion.

So, when working with strings that contain numbers, be careful.

? 31.2 [5 min]

What is the result of the following operations? If you are unsure, try it and explain the result:

- (a) `'ab' + "def"`
- (b) `'98' + "5"`
- (c) `'98' + 5`
- (d) `'98' + "5.3"`
- (e) `'98' * 5`
- (f) `'98' * false`
- (g) `'98' * 'false'`

If you want to consider the two numbers contained in the string variables `y` and `z` of the last section as **number** values and add them, it is not sufficient to link the two variables with the plus operator, because this leads to a string concatenation. Therefore, an *explicit* conversion is required, which converts values into numbers before linking them with the plus operator. The function **Number(notNumberValue)** is available in JavaScript for this purpose:

```
> y='4'; z='5'
> Number(y) + Number(z)
9
```

Number() is a function that returns an elementary **number** value. At the same time, **Number()** is also the constructor function of the **Number** class. With an assignment of the form **variable = new Number()** (a notation we will look at in more detail in ► Sects. 31.5.4 and 31.5.5), you can create an object of the class **Number**. These objects house all the properties and methods we've already worked with in the context of **number** variables. And this is exactly how elementary data types like **number** get their properties and methods: If we access a property or method of the elementary **number** data type, a data type that is *primitive* and thus does not actually represent an object and should therefore have neither methods nor properties, JavaScript converts the variable into an object of type **Number** in the background. The relevant properties and methods are then available for this object. After the work is done, JavaScript's automatic garbage collection disposes of the now unused object, leaving behind the original elementary variable. This "trick" allows JavaScript to make even elementary data types look like real objects with properties and methods. So much for a brief look "under the hood" of JavaScript. Corresponding functions exist with **String(notStringValue)** and **Boolean(notBooleanValue)** also for the other two elementary data types. In the following example, we convert the string `'true'` into a real **boolean variable**.

```
> trueval = 'true'
> typeof trueval
"string"
```



```
> trueval = Boolean(trueval)
> typeof trueval
"boolean"
```

By the way, as you can easily try out, the **Boolean()** function remains completely unimpressed if you ignore the usual rules of case sensitivity and write **"True"** or **"TRUE"** in your string, for example; the conversion still works perfectly. This is because **Boolean()** evaluates to **true** anything that is not **0**, **null**, **undefined**, or **NaN**. Thus, **Boolean("hello")** also returns the value **true**! While the glutton **Boolean()** evaluates everything you give it, **Number()** is much more picky: If you call **Number()** with an argument that is not convertible to a number, the function refuses to work and will return **NaN**.

For the conversion between strings and numbers there are some special functions besides **Number()** and **String()**. **parseInt(string)** and **parseFloat(string)** both process a string to a number and thus at first sight do the same as **Number()**. The special feature of **parseInt()** and **parseFloat()**, however, is that they also process strings which are not exclusively numeric as long as the numeric part is at the beginning:

```
> parseInt('3 twins are one too many')
3

> parseInt('3.1415926535 is the number Pi')
3
```

31

The second example also shows that **parseInt()**—as the name already suggests—only processes the integer part of the number found at the beginning of the string. If nothing is found at the beginning of the string that can be interpreted as a number, **NaN** is returned.

In the opposite direction of conversion—from number to string—there are also some special functions available to help control the representation of the number as a string. The **toString(numbersystem)** method of **number** (actually of the **Number** object, as we know by now), allows you to specify the base of the number system you want to convert to; **numbersystem = 2** would thus lead to a binary representation, **numbersystem = 16** to a hexadecimal representation:

```
> someNumber = 156
> someNumber.toString(2)
"10011100"

> someNumber.toString(16)
"9c"
```

You can use the **toFixed(*decimals*)** and **toExponential(*decimals*)** methods of the **Number** object to affect the number of decimals in a traditional representation as well as a scientific representation:

```
> number = 156.27813
> number.toFixed(2)
"156.27"

> number.toExponential(3)
"1.562e+2"
```

The exponent is chosen in the exponential representation in such a way that exactly one digit before the decimal point is always shown.

31.4 Arrays

■ Creating Arrays and Accessing Individual Array Elements

Arrays in JavaScript are similar to what is known as lists in some other languages (including Python, see ► Sect. 21.6.1). Not only can they hold elements of any type, but the elements can also be of *different* types. Even arrays themselves can be elements of other arrays. In this way, multidimensional arrays can be created, which JavaScript does not know out-of-the-box.

The easiest way to create an array is to specify its elements directly in the form of an *array literal*. The elements are enclosed in square brackets:

```
> primes = [1,3,5,7,11,13]
```

In the same way, however, the objects can also have different types:

```
> multipleTypes = [false, 'Caroline', 28.3]
```

The individual elements are accessed via a numerical index, which—as in many other languages—also starts at 0 and is specified in square brackets. So, if we wanted to access the second element of the array we just created (“Caroline”), we would do so as follows:

```
> multipleTypes[1]
"Caroline"
```

Accordingly, `multipleTypes[0]` would return the first element of the array, **false**.

The elements of arrays can also be empty; more precisely, they can be *left blank* and thus take the value **undefined**:

```
> multipleTypes = [false, , 'Caroline', 28.3]
> multipleTypes
[false, empty, 'Caroline', 28.3]
```

If you view the contents of the array, *empty* is displayed—at least in Google *Chrome*’s JavaScript console—for the array element that has been left empty. However, by displaying this directly, it is easy to convince yourself that the content is indeed **undefined**:

```
> multipleTypes[1]
undefined
```

Arrays are not elementary data types, not *primitives*, but *objects*. Therefore, arrays can be created not only by assigning an array literal as we did above, but also by calling the constructor function of the array type, **Array()**. The array used above could therefore also be constructed in this way:

```
> multipleTypes = new Array(false, undefined, 'Caroline', 28.3)
```

31

The array elements that is left empty must be explicitly assigned the value **undefined** as leaving it truly empty leads to an error message. We have already seen the keyword **new** in ► Sect. 31.3.2; in ► Sects. 31.5.4 and 31.5.5 we will look at its meaning in more detail.

By the way: If you specify only one positive integer as the argument of the constructor **Array()**, then you do not get an array with only one element, but a completely empty array with the specified number of elements will be created.

■ Selecting Multiple Elements from Arrays

Arrays provide a whole range of useful properties and methods as objects. In the following, we will use some of these properties and methods to work more intensively with arrays. Let’s start with the selection of elements from arrays.

We have already seen how to access an array element by specifying the element index. But what if several elements are to be selected at once? Unlike Python, JavaScript does not have an operator to “grab” an entire range of index values at once. However, the **Array** class has the method **slice(from, to)**, which can use to achieve the same effect. In this case, all index values from index value *from* are returned *before* index value *to*, but without this element itself. Consider the following example:

```
> prime = [1,3,5,7,11,13]
> prime.slice(2,4)
[5, 7]
```

So, the elements with the index values 2 and 3 are selected (because the indexing starts at 0, this means the third and fourth element in the array are selected).

The **to** argument can also be omitted. In this case, the entire remainder of the array is returned starting at the index position **from**:

```
> prime.slice(3)
[7, 11, 13]
```

Negative values are also possible for the **from** and **to** arguments of **slice()**. In this case, the selection is made from the back, with the last element of the array carrying the index value -1:

```
> prime.slice(-3,-1)
[7, 11]
```

Please note that also here the elements are selected before the element with the second index.

■ Determining and Changing the Length of an Array

The **length** property can be used to determine the length of arrays:

```
> multipleTypes = new Array(false, undefined, 'Caroline',
  28.3, undefined)
> multipleTypes.length
5
```

This property can also be changed, so we can shorten an array by assigning a new, smaller value to its length:

```
> multipleTypes.length = 3
[false, empty, 'Caroline']
```

If we “extend” it again afterwards, the previously “cut away” elements do not reappear, but the new elements are filled with **undefined** (in Google Chrome’s representation: **empty**):

```
> multipleTypes.length = 5
[false, empty, 'Caroline', empty x 2]
```

An array literal is also an **Array** object, and so we can also access the object properties and methods of arrays for an array literal. Unlike **number** or **string**, for example, the literal does *not* have to be enclosed in round brackets:

```
> [1,3,5,7,11,13].length
6
```

■ Changing Elements in Arrays, Adding Elements to Arrays or Deleting Elements from Arrays

So far, with the exception of changing the array length, we have worked with arrays in a *read-only manner*. Of course, you can also *change* the value of an array element using the selection technique we have already learned about:

```
> primes = [1,3,5,7,11,13]
> primes[2] = 'Gap in prime series'
> primes
[1, 3, 'Gap in prime series', 7, 11, 13]
```

You can *delete* elements from an array using the **splice(*from*, *count*)** method of the **Array** object. *count* elements from (and including) the element at index position *from* are deleted. The **splice()** method returns an array of the deleted elements:

```
> primes.splice(2,3)
['Gap in prime number series', 7, 11]

> primes
[1, 3, 13]
```

Unlike the methods provided for the primitives **number** and **string**, **splice()** modifies the original array for which it is called.

By the way, the argument **count** is optional: If **count** remains unused, **splice** deletes the rest of the array including the index position **from**.

splice() can be used not only to delete elements, but also to insert them. The second argument (**count**) can be followed by any number of further values, which are inserted *after* the element with the index position **from**; the **count** elements specified by the argument **count** are nevertheless deleted beforehand:

```
> primes = [1,3,5,7,11,13]
> primes.splice(1, 2, 'Gap 1', 'Gap 2')
[3, 5]

> primes
[1, "Gap 1", "Gap 2", 7, 11, 13]
```

If you only want to insert, but not delete, set the argument **count** to 0:

```
> primes.splice(3, 0, 'Gap 3')
[]
> primes
[1, 'Gap 1', 'Gap 2', 'Gap 3', 7, 11, 13]
```

splice(), since no elements were deleted from the array, returns an empty array as the function value.

In addition to **splice()**, the **Array** object has other methods that can be used to add elements to or remove elements from the array. The **push()** method appends an element to the back of the array and returns the new length of the array as a function value. **pop()** deletes the last element from the array and returns the value of the deleted element:

```
> primes.push(57)
8

> primes
[1, 'Gap 1', 'Gap 2', 'Gap 3', 7, 11, 13, 57]

> primes.pop()
57

> primes
[1, 'Gap 1', 'Gap 2', 'Gap 3', 7, 11, 13]
```

The **shift()** and **unshift()** methods work very similarly to **push()** and **pop()**, but work with the beginning of the array instead of its end (try it out!).

■ Merge Arrays

Two arrays can be conveniently concatenated using the **concat(*otherArray*)** method. This places the elements of the array *otherArray* after the last element of the array whose **concat()** method is called. However, the array is not changed by calling its

concat() method; instead, the new, merged array is simply returned as a function value and can be caught in a variable:

```
> primes = [1,3,5,7,11,13]
> evenNumbers = [2,4,6,8,10]
> myNumbers = primes.concat(evenNumbers)
> myNumbers
[1, 3, 5, 7, 11, 13, 2, 4, 6, 8, 10]
```

■ Sort Arrays

You can use the **sort()** method of the **Array** object to sort the elements of an array *alphabetically* in ascending order according to their values:

```
> primes = [1,3,5,7,11,13]
> primes.sort()
> primes
[1, 11, 13, 3, 5, 7]
```

You may have noticed that **sort()** treats the elements of the array as strings and therefore places, for example, 3 *after* 13, an order that would not result from numerical sorting. If this behavior is not desired, it can be changed. In fact, the **sort()** method has an optional argument that allows you to specify a function that is passed two values (let's call them **x** and **y**) as arguments and returns a positive value whenever **x** is to come before **y** in the sort order and a negative value in the opposite case. So, by specifying a comparison rule that decides, for any two values, which of the two should appear first in the order and which second, you can fine-tune the behavior of the **sort()** function. To achieve numerical sorting, we could write ourselves a helper function **greater(x,y)** that returns a positive value if **x > y**, and a negative value otherwise. The expression **(x-y)>0** is a logical expression, i.e., an expression that takes the value **true** or **false**, depending on how the comparison of **x** and **y** turns out. With this, our numerical sorting would then look like this:

```
> greater = function(x, y) {
  return (x - y) > 0;
}
> primes.sort(greater)
> primes
[1, 3, 5, 7, 11, 13]
```

We will deal with the definition of functions in more detail in a later chapter.

To sort the values in descending order, we can use the **reverse()** method of the **Array** object. It simply flips the elements of an array around. If we apply this method to the array previously sorted with **sort()**, we get a descending sort:

```
> primes.sort(greater).reverse()
```

The notation with the two dot operators may seem strange, but it is actually very logical: The expression **primes.sort(greater)** returns an **Array** object, which in turn has a **reverse()** function. This is called with the usual dot notation. Of course, you can also break the whole operation into two steps.

Both **sort()** and **reverse()** not only return the result of the respective operation as a function value, but also change the array for which they are called. In this last respect, they resemble **splice()** and differ from **concat()**.

■ Display Arrays as Strings

With **join()** and **toString()**, the **Array** object provides two methods to combine its elements into a string. The original array is not touched by either method. **join()** and the **toString()** method offered by any JavaScript object have basically the same effect, but **join()** is more flexible in that an optional argument can be used to specify the separator character that will be between each array element in the generated string, while **toString()** stubbornly uses the comma as the separator character:

```
> primes = [1,3,5,7,11,13]
> primes.toString()
"1,3,5,7,11,13"

> primes.join()
"1,3,5,7,11,13"

> primes.join('-')
"1-3-5-7-11-13"
```

There is a danger of confusion between the **join()** method and the **concat()** method: Contrary to what the name **join()** might suggest, here it is not *two different arrays* that are joined, but the *different elements* of an array.

By the way: If you “add” two arrays, you will also get a string that includes all elements of both arrays; but be careful, because no separator is inserted between the two arrays, i.e., between the last element of the first array and the first element of the second array (hence the “number” **132** in the example):

```
> primes = [1,3,5,7,11,13]
> evenNumbers = [2,4,6,8,10]
> primes + evenNumbers
"1,3,5,7,11,132,4,6,8,10"
```


■ Decompose Strings to Arrays

Just as you can use `join()` and `toString()` to convert arrays into strings, it is possible to split a string and make the individual parts elements of an array. In the following example, consider the string **friends**, which contains a comma-separated list of names:

```
> friends = 'Thomas,Marc,Sharon,Hazel'
```

With the help of the method `split()`, the string can now be split into the individual names and an array can be “fed” with these. The argument of `split()` is the separator character that separates the individual parts of the string, in our example the comma:

```
> friendsArray = friends.split(',')
> friendsArray
['Thomas', 'Marc', 'Sharon', 'Hazel']
```

■ Repetition: Accessing String Characters in Array Notation

In JavaScript, as we saw in ► Sect. 31.2.2, strings are an elementary data type, a *primitive*, and thus not an object. As you recall, their individual characters can nevertheless be accessed like the elements of an array:

```
> var message = 'Hello World!'
> message[1]
"2"

> message[message.length-1]
"!"
```

With the last command, we access the last character in the string (remember that indexing starts at 0!). However, you cannot change the individual characters of a string with the array notation. While such an attempt will not result in an error message, the attempted change will not take effect in the string. Strings are, in a sense, “read-only arrays”.

? 31.3 [5 min]

Create a string with the value **"Hello World"**. Select the sixth and the eighth character from this string. Then remove these characters from the string.

31.5 Objects

31.5.1 Object Orientation in JavaScript

Object orientation in JavaScript works somewhat differently than in most other object-oriented languages. JavaScript does not inherently recognize the central basic concept of object-oriented programming, *classes*. Instead, JavaScript takes an approach that creates objects based on *prototypes*. “Prototypes” and “classes” may not sound very different at first, since classes in object-oriented programming also have a prototype function for the objects derived from them (these are, in a sense, built according to the “prototype” of the class). In fact, however, the two approaches are quite different.

Two examples: Unlike in typical object-oriented languages with their classes, in JavaScript properties and methods can be *added directly* to individual objects (in normal object-oriented terminology you would say: to the instances of classes) that are not included in the in the prototype definition. Thus, without deriving additional classes from a base class, different object instances of a type (class) can differ from each other in their methods and properties. In other object-oriented languages, all objects of the same type (class) would have the same structure in the form of available methods and properties (the latter, of course, quite possibly assigned different values). Another difference between the prototypical approach of JavaScript and the class-based approach of other object-oriented languages is that in JavaScript the structure of classes is not explicitly described; there is (or rather: was, until ECMAScript version 2015, which, however, does not change anything about the prototypical approach) thus no syntax structure that describes a type (class) abstractly as a collection of methods and properties, as we have seen in Python, for example. Instead, the type definition in JavaScript is completely wrapped up in the type’s constructor and its reference to the prototypes of other objects from which it “inherits” methods (inheriting properties doesn’t really make sense in a prototype approach, unless you want *static* properties in the spirit of object-oriented programming, meaning they have the same value for all object instances).

If the last paragraph seems very technical and rather difficult to skim read, don’t worry, we’ll skirt most of the shoals of JavaScript’s own prototype approach in the following sections. Covering them would significantly exceed the book’s idea of *quickly* learning the *most important* basic concepts of the language to be able to apply them.

Unlike the Python part of the book, we refrain here from dealing with inheritance and thus deriving objects from other objects or their prototypes. You will be able to master most application scenarios well, even without having fully internalized JavaScript’s unusual but very flexible approach to object orientation down to the last detail (especially if you come from a “classical” object-oriented way of thinking).

31.5.2 Creating Objects Directly

The simplest way to create an object in JavaScript is a variable declaration, in which not only a value, but also (one or more often) several properties are assigned to an (object) variable. To demonstrate this, we will work with the example of a product definition that has already been used several times.

An object with these same properties can easily be created in JavaScript with an assignment like the following:

```
var product = {  
  name: 'Garden shovel, stainless steel',  
  price: 10.99  
}
```

As you can see, a variable called **product** is declared here and something is directly assigned to it as part of the declaration. What is assigned are the properties of the product. The curly braces make it clear that **product** is a self-defined object. The properties specified in the curly braces each consist of an identifier, in our example **name** and **price**, and a value that is assigned to the property after the colon. Multiple properties are separated by commas within the object declaration.

The properties of the product can also be assigned variables (whose current value is thus assigned to the property); in particular, the values of properties can in turn be other objects.

Rarely encountered in practice, but syntactically permissible, it is also possible to enclose property identifiers in (single or double) quotes: This allows you to even include spaces in the property identifiers, for example:

```
var product = {  
  'Name of the product': 'Garden shovel, stainless steel',  
  price: 10.99  
}
```

This way you can add identifiers to properties that would otherwise be illegal in JavaScript, such as **#hashtag** (illegal because of first character). You could also use otherwise reserved keywords like **var** as property identifiers. However, this is also rather uncommon because of the expected poor readability and the higher error-proneness of the code.

As with “free-standing” variables, JavaScript itself decides on the data type that the properties must have; in our example, **name** will be of type **string**, **price** of type **number**.

31.5.3 Accessing Properties of Objects

Objects are nothing more than *associative arrays*: Arrays of properties consisting of key-value pairs.

But what about methods? In the sense of object-oriented programming, we understand objects as constructs that include properties (attributes) and methods, i.e., functions, with the help of which the properties can be worked with. The methods, however, cannot be properties at the same time. So how can an object in JavaScript be practically an associative array that *only* has properties? The trick is that in JavaScript, functions are also objects, and they are objects of type **function**. But we saw in the last section that the value of an object property can in turn be an object and thus also a function.

If objects are ultimately a kind of associative array, then their properties (and thus also methods) can be accessed by specifying the key, i.e., the property identifier. For this purpose, it must be specified in quotes and square brackets. After you have executed the object declaration from the last section in the JavaScript console, you can now easily access the **price** property of the **product** object:

```
> product['price']  
10.99
```

The key may of course be a variable itself:

```
> property = 'price'  
> product[property]  
10.99
```

This notation makes clear the character of JavaScript objects as associative arrays.

In practice, however, the access method that is also known from many other object-oriented languages is much more common, namely with the help of the dot operator:

```
> product.price  
10.99
```

This type of access only works if the identifiers of your object properties are valid JavaScript identifiers, i.e., do not contain spaces or start with a special character other than underscore and dollar sign. However, it is always recommended to choose identifiers that comply with the usual rules.

31.5.4 Creating Objects Using the Object Constructor

In ► Sect. 31.5.2, we created an object by assigning a set of key-value pairs, namely the object's properties, to a variable when we declared it. In this section, we will learn a second way to create an object. To do this, we take advantage of the property that all objects in the JavaScript object hierarchy are derived from the basic type **object**. Therefore, we first create a variable of type **object** by calling the constructor function (more about JavaScript constructors in the following section). Please note the capitalization of the constructor function:

```
> var product = new Object()
> typeof product
"object"
```

With this we have created an empty object. If you type **product**. into the console, you can see from the popup window that now opens that our supposedly empty object already contains a whole set of properties and methods, namely those that the type **object** has by default. But we want to give the object our standard **product** properties **name** and **price**. This is now done by simple assignment:

```
> product.name = 'Garden shovel, stainless steel';
> product.price = 10.99;
```

You can easily verify that our **product** object actually has the properties **name** and **price** with the appropriate values:

```
> product.name
"Garden shovel, stainless steel"

> product.price
10.99
```

31.5.5 Creating Objects Using Constructor Functions

The last—and probably most important—way to generate objects is to write a constructor function that creates the object. In the previous section we also used a constructor function, namely the constructor **Object()**. In this section, we will now build ourselves a constructor to create our very own type of object.

We want our **product** object to have two properties, **name** and **price**; these should be directly specifiable when the object is created. We achieve this with a constructor function like the following:

```
function product(price, name) {  
  this.price = price;  
  this.name = name;  
}
```

The constructor is passed two values as arguments, which are then assigned to properties of the newly created object, using the keyword **this**. It refers to the current object in whose context it is used. In our example, this refers to our newly created **product** object. This also completes the definition of our very simple type. We can now create an instance of this type by calling the constructor function we just developed:

```
> gardenShovel = new product(10.99,  
  'Garden shovel, stainless steel')
```

Note the keyword **new**. It ensures that a new object is created. If the constructor function is called without this keyword, no new object is created, but simply **undefined** is returned.

31.5.6 JSON

Even if you don't work with JavaScript, chances are you'll encounter the JSON data format at some point. For example, it is often used in Internet APIs for retrieving information from web services. Such interfaces often return their results to the calling application in JSON format. Next to XML, it is the second most important data exchange format on the Internet. JSON is the abbreviation for *JavaScript Object Notation*, so it's certainly not surprising that we're focusing our attention on this popular data exchange format at the end of our look at objects in JavaScript.

We have already encountered the JSON format in ► Sect. 31.5.2, where we created objects directly as literals. The notation used there with its key-value pairs, which are placed in braces separated by commas, is nothing other than a notation in JSON format.

Consider the following excerpt from a JSON dataset:

```
{  
  'customer01': {  
    'First name': 'Phil',  
    'Last name': 'Philipson',  
    'Address': {  
      'Street': '1 Philly Drive',  
      'ZIP': 19145,  
      'City': 'Philadelphia',  
      'State': 'PA'  
    }  
  }  
}
```

```

    }
  },
  'customer02': {
    'First name': 'Auric',
    'Last name': 'Goldfinger',
    'Address': {
      'Street': '450 Gold Avenue',
      'ZIP': 40121,
      'City': 'Fort Knox'
      'State': 'KY'
    }
  }
}

```

This JSON object (delimited by the outer curly brackets) comprises two (sub) objects, **customer01** and **customer02**, which are characterized by different fields. One of the fields, the address, is itself an object composed of different fields. The field names, i.e., the keys of the key-value pairs, are enclosed in quotes in JSON, which—as already mentioned—is also syntactically permissible in JavaScript.

Using the **JSON.stringify(object)** function, you can convert any object into a JSON string, which is also called *serialization*; conversely, **JSON.parse(string)** allows you to turn a JSON string (returned as the result of a web API call, for example) into a real JavaScript object.

Although the origin of JSON is closely linked to JavaScript, today practically all common programming languages offer functions for working with data in JSON format, which underlines the popularity of JSON as a data exchange format. This popularity certainly stems from the fact that JSON allows hierarchical object structures to be represented with little effort in a syntactically straightforward way that can be read well even without JSON knowledge.

31

? 31.4 [5 min]

Create an array of objects, each containing the name and age (in years) of your immediate family members. Then access and display the name of the second person in this array. Finally, convert this array into a JSON string.

? 31.5 [10 min]

Create two objects, each containing a customer address. Work once with the direct creation of objects by assigning key-value pairs to a variable, once with the general object constructor **Object()**.

31.6 Summary

In this chapter we have dealt with elementary variables and objects in JavaScript.

Be sure to take the following points from this chapter:

- Variables are not mandatory to declare in JavaScript, however it is good practice to do so.

- The main elementary data types are **number** (numbers, both integers and floating-point numbers), **string** (strings) and **boolean** (logical values).
- Strings can be enclosed in single or double quotes; individual characters of a string can be accessed in simple array notation (**string[characterIndex]**), but only in read-only mode.
- For the elementary data types, which are not objects themselves, there are object prototypes with the same name (but with a capital letter, such as **Number**) that provide useful properties and methods; if necessary, JavaScript automatically converts the elementary data types temporarily into objects of the corresponding type in the background, so that you can access the properties and methods as if they were properties or methods of the elementary data types themselves.
- The special value **undefined** is used when a variable does not yet have a defined value (or a function does not return a real return value), whereas **null** is used whenever a conscious decision is made that a variable should remain “empty”.
- JavaScript already implicitly converts where necessary.
- Caution is advised when converting strings: The plus operator is used with strings to link two strings, but undesirable effects can occur if numbers contained in strings are to be added numerically, and are not explicitly converted to the **number** type beforehand.
- An explicit conversion of the elementary data types among each other can be achieved with the help of the constructors of the object types belonging to the elementary types, for example with **Number(string)**.
- Arrays are lists that can hold variables (also objects) of different data types; the individual elements of the array are accessed in the notation **array[elementIndex]**, with indexing starting at 0.
- JavaScript does not recognize any classes in its core definition; object orientation is realized with the help of prototypes, after whose likeness objects can then be created; to the instances of objects further properties and methods can be added, which the prototype, whose construction the object follows, did not possess.
- Objects can be created directly by assigning a comma-separated list of elements/properties in the form of key-value pairs (object notation) to a variable in curly braces, for example: **object = {property1: value1, property2: value2}**. Similarly, an “empty” object can first be created with the object constructor **Object()** by assigning the form **object = new Object()** and then successively creating properties in it, by assigning the form **object.property = value**.
- You can also create objects by calling the constructor of the respective object type, for example **object = MyObject()**.
- You can access the elements/properties of objects using the dot operator in the notation **object.property** or, as with an (associative) field, in an array notation of the form **object["property"]**.
- *JavaScript Object Notation (JSON)* is a common exchange format for data on the Internet. It corresponds exactly to the notation that is also used for the direct creation of objects as a comma-separated list of key-value pairs specified in curly braces. Any JSON document can therefore be transformed into a JavaScript object with **JSON.parse(jsonDocument)** and, conversely, any JavaScript object can be represented as a JSON document with **JSON.stringify(object)**.

31.7 Solutions to the Exercises

■ Exercise 31.1

```
> Infinity + 1
Infinity
> Infinity + 1 == Infinity
true
```

Here we use the special value **Infinity**, with which we can also calculate. By the way, you can see from the second input that infinity plus one equals infinity again (the double equal sign is the comparison operator). So, the comparison results in a true statement, accordingly the value **true** is returned.

■ Exercise 31.2

- (a) **'abc' + 'def' = 'abcdef'**. Strings are concatenated by the plus operator.
- (b) **'98' + '5' = '985'**. The plus operator also appends the two strings together here. The fact that they happen to contain numbers does not matter.
- (c) **'98' + 5 = '985'**. Here the second summand is a real number. However, the plus as *string concatenation operator* has priority in the processing of the expression. Therefore, JavaScript implicitly converts the number **5** into a string in order to be able to concatenate it with the string **'98'**.
- (d) **'98' + '5.3' = '985.3'**. Also, if the strings contain fractional numbers, the plus operator results in string concatenation.
- (e) **'98' * 5 = 490**. The multiplication operator has no meaning for strings. Because an operation with strings is therefore out of the question here, JavaScript implicitly converts the string **"98"** into a number in order to be able to perform a meaningful operation after all.
- (f) **'98' * false = 0**. The constant **false** is internally evaluated with as value **0**. Since multiplication makes no sense for strings, JavaScript tries an operation with numbers and converts the string **"98"** into a number for this purpose.
- (g) **'98' * 'false' = NaN**. Here even JavaScript capitulates. The multiplication is obviously an operation with numbers as operands, but JavaScript gets served two strings here. The result is *not a number*. In no way are both strings converted into numerical values, although that would happen with **"98" * "5"** (try it out!).

As you can see, it is not easy to predict JavaScript's implicit conversions without knowing the full set of rules in detail. It is therefore best not to rely on implicit conversions, but to make sure that explicit conversions are performed wherever they might be needed.

■ Exercise 31.3

Strings behave like arrays in *read access*, accordingly we can select out the characters in array notation:

```
> message = 'Hello World'
> message[5]
" "

> message[7]
"o"
```

With regard to write access, however, strings do not behave like arrays. Using the array method **splice()** to delete the elements is therefore out of the question. One way to achieve the desired effect is to “slice” the string by using **slice()** to cut the corresponding parts so that the characters at index positions **5** and **7** fall out. Note that **slice()** always selects up to *before* the specified second index:

```
message = message.slice(0,5) + message.slice(6,7) +
    message.slice(8, message.length)
```

■ Exercise 31.4

We create an array whose elements are objects that we store as key-value pairs in the usual object notation:

```
> family = [{name: 'Mark', age: 28}, {name: 'Cathy',
age: 54}, {name: 'Ben', age: 57}]
```

We get the name of the second person by first selecting the person object from the array using its index (**family[1]**) and then accessing its **name** property, which we do in the usual dot notation:

```
> family[1].name
"Cathy"
```

The **stringify()** method of the global JSON object helps with the conversion to a JSON string.

```
> JSON.stringify(family)
"[{"name":"Mark","age":28},{\"name\":\"Cathy\",\"age\":54},
{\"name\":\"Ben\",\"age\":57}]"
```

■ Exercise 31.5

Direct generation in object notation as a comma-separated list of key-value pairs:

```
> var customer1 = { customerNumber: 14527,  
  name: 'Peterson', firstName: 'Carl',  
  street: '54 Wellington Drive', zip : '02113',  
  city: 'Boston, MA' }  
> var customer2 = { customerNumber: 19321,  
  name: 'Hamilton', firstName: 'Beth',  
  street: '21 Eagle's Square', zip : '10012',  
  city: 'New York, NY' }
```

Generation by first letting the general **Object()** constructor create an “empty” object and then equipping it with our properties:

```
> var customer1 = new Object()  
> customer1.customerNumber = 14527  
> customer1.name = 'Peterson'  
> customer1.firstName = 'Carl'  
> customer1.street = '54 Wellington Drive'  
> customer1.zip = '02113'  
> customer1.city = 'Boston, MA'
```

The procedure for **customer2** is analogous.



User Interfaces: How Do I Input and Output Data?

Contents

- 32.1 Overview of Input and Output in JavaScript – 453**
- 32.2 Output Via the Console – 454**
- 32.3 Inputs and Outputs Via Dialog Boxes – 457**
- 32.4 Output to an HTML Document/Web Page – 458**
 - 32.4.1 Writing HTML Code into the Web Page – 458
 - 32.4.2 The Document Object Model (DOM) – 460
 - 32.4.3 Selecting DOM Nodes by Their Properties – 461
 - 32.4.4 Selecting DOM Nodes via the Hierarchical Structure of the Document – 463
 - 32.4.5 Changing HTML Elements – 466
 - 32.4.6 Adding and Deleting HTML Elements – 468
- 32.5 Input with Forms – 470**
 - 32.5.1 Forms in HTML – 470
 - 32.5.2 Accessing Forms from JavaScript – 472
- 32.6 Example: Simple Calculator – 476**
 - 32.6.1 The Web Interface – 477
 - 32.6.2 The CSS Design Instructions – 479
 - 32.6.3 The JavaScript Code – 482

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1007/978-3-658-42912-6_32.

32.7 Example: Color Picker – 484

32.7.1 The Web Interface – 484

32.7.2 The JavaScript Code – 486

32.8 Summary – 488

32.9 Solutions to the Exercises – 490

Overview

JavaScript is the primary language used on the web, which explains why data input and output largely involve interactions with the webpage that hosts the JavaScript code.

JavaScript offers many possibilities to receive information from the user of the web page and to modify the web page to output information. The interaction with the surrounding web page is made possible by the fact that JavaScript allows access to its individual elements via the *Document Object Model (DOM)* of the web page. But before we turn to using the DOM, let's take a quick look at output with the JavaScript console, which we've already used a lot, and at working with dialog boxes.

In this chapter you will learn the following:

- how to output objects in the console
- how to conveniently build output strings using variables with template literals and string substitutions
- how to display information via dialog boxes and request decisions from the user
- what the *Document Object Model (DOM)* is, and how it is structured
- how to select and modify individual elements of a web page on the basis of the DOM
- how to use forms to accept input from the user

32.1 Overview of Input and Output in JavaScript

The simplest way to output data with JavaScript is to output to the console. We made extensive use of this option in the last chapter, mainly because the console provides interactivity that is useful when trying out new language concepts. One enters an instruction and is immediately shown the result.

When writing real JavaScript applications, however, you typically won't want to work with the console, which is a developer tool and therefore usually invisible to your website viewer. Instead, you want your JavaScript application to interact with the website and output information there for the user to see.

In this chapter, after taking a quick look at the console output we're already familiar with, we'll look at how you can output (and input) data via *dialog boxes*. After that, however, we'll turn to the core area of input and output, working with the web page that the script is embedded in. In doing so, we'll see how individual elements of the website can be modified from within your JavaScript program via what's called the *Document Object Model (DOM)*. In particular, we'll look at *forms*, which are particularly useful for website interactivity because they allow users to enter text and other information directly. Working with forms is an important application area of JavaScript for a reason.

Unlike other programming languages, we won't get into working with files in JavaScript because, for security reasons, JavaScript usually doesn't have access to the local computer's file system.

We will conclude the chapter with two small sample applications, a simple calculator and a color picker, that can be used to conveniently generate the hexadecimal color codes commonly used in HTML.

32.2 Output Via the Console

In the last chapter, we already worked frequently with the **log()** method of the **console** object to output data to the console quickly and easily. In this section, we'll take a closer look at it, and in particular at how **console.log()** can be used to produce output that is composed of several different parts. The approaches presented here for **console.log()** are transferable and can be used in many places where strings are used.

■ Printing a List of Multiple Objects

If you want to output multiple values/objects in a row, simply pass them to **console.log()** as a comma-separated list:

```
> console.log('Today is: ', Date(), '. A random number between
0 and 10 is: ',
Math.round(Math.random()*10,0))
```

```
Today is: Mon Oct 03 2022 13:24:53 GMT+0200 (Central European
Summer Time). A random number between 0 and 10 is: 4
```

console.log() simply outputs the objects passed as arguments stubbornly one after the other. The outputs of the different objects are separated by a space.

32 ■ Printing Multiple Objects as a Concatenated String

If you want to avoid this whitespace, you must first assemble the objects into a string, controlling the occurrence of whitespace as desired, and then output the complete string:

```
> output = 'Today is: ' + Date().toString() + '. A random
number between 0 and 10 is: ' +
Math.round(Math.random()*10,0).toString()
```

```
> console.log(output)
```

```
Today is: Mon Oct 03 2022 13:30:19 GMT+0200 (Central European
Summer Time). A random number between 0 and 10 is: 9
```

■ Working with Template Literals

Another method to achieve the same is to work with a *template literal*. Template literals are strings that contain placeholders. Unlike conventional strings, they are enclosed in *back ticks* (```). Within a template literal, placeholders can be inserted to represent the value of variables or other expressions. Wildcards are characterized by starting with a dollar sign (\$) and enclosing the expression they represent in curly braces, as in the following example:

```
> randomNumber = Math.round(Math.random()*10,0)
> output = `A random number between 0 and 10 is:
    ${ randomNumber}.`
> console.log(output)
A random number between 0 and 10 is: 6.
```

This way, you don't have to painstakingly assemble your string, making sure that the quotes are set correctly for each substring and that the substrings themselves are all connected with plus operators. It is sufficient to simply write a long string in which you write everything you want to represent in variables or other expressions as placeholders.

Important: The values of the variables are fixed at the time when the template literal is created. Subsequent changes to the values of the variables are then no longer reflected in the template literal. The following example illustrates this:

```
> value = 2
> output = `Current value: ${value}`
> value = 3
> console.log(output)
Current value: 2
```

By the way, a very practical feature of template literals is that, unlike conventional strings, they can extend over several lines, as this example shows:

```
> twoLines = `First line
Second line`

> console.log(twoLines)
First line
Second line
```


In a normal string, we would have had to work with the escape sequence `\n` to do this:

```
> console.log('First line\nSecond line')
First line
Second line
```

■ Working with String Substitutions

Another way to generate a composite output is to use string substitutions, which are supported by **console.log()** and other JavaScript functions. This also works with wildcards. A simple example illustrates how this works:

```
> pi = 3.14159
> console.log('The value of the number pi is: %f', pi)
3.14159
```

The placeholder here is composed of the percent sign and a formatting instruction; **f** instructs the output to display the number as a floating-point number. What the placeholder is finally filled with depends on the further argument of the **console.log()** function. In our example, the first argument after the string is our variable **pi**, so this is used for the first placeholder found in the string. If the string contained further placeholders, their substitutions would be added as further arguments after **pi**.

Similarly, a value can be output as a string with **%s**, or as an integer with **%d** and **%i**:

```
> console.log('The value of the number pi is: %', pi)
3
```

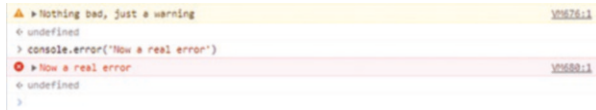
32

■ Producing Warnings and Errors

So far, we have always output to the console using **console.log()**. However, you can also output warning and error messages, which are then highlighted and provided with appropriate icons:

```
> console.warn('Nothing bad, just a warning')
> console.error('Now a real error')
```

You can see the result of these outputs in ■ Fig. 32.1.



■ Fig. 32.1 Self-generated warning and error messages

32.3 Inputs and Outputs Via Dialog Boxes

JavaScript provides the ability to cause the browser to display messages as small dialog boxes. With the help of the **alert(message)** function, you display your message prominently to the user:

```
> alert('An important Message')
```

A dialog box opens as shown in ■ Fig. 32.2.

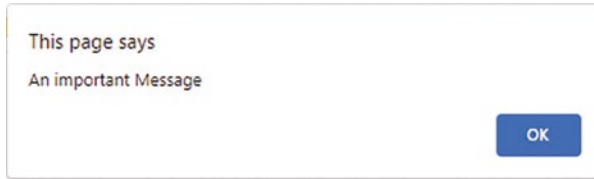
If you want to have the user confirm an action, you use the **confirm(message)** function, which displays a message and offers the user the “Okay” and “Cancel” buttons in a dialog like the one shown for the example in ■ Fig. 32.3. **confirm()** returns **true** if the user selected **Okay** and **false** if the user selected **Cancel**.

```
> action = confirm('Do you really want to continue learning
JavaScript?')
> console.log(action)
True
```

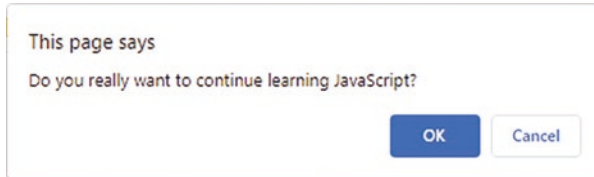
Another way to accept input from the user is to use the **prompt(message)** function. It creates a dialog box in which the user can make any input, which **prompt()** returns as a string (even if the user enters a number!)

The following example, in which we also use the template literals introduced in the previous section, should look familiar. It is the conversion of a temperature value from degrees Celsius to Kelvin, which has already been used several times:

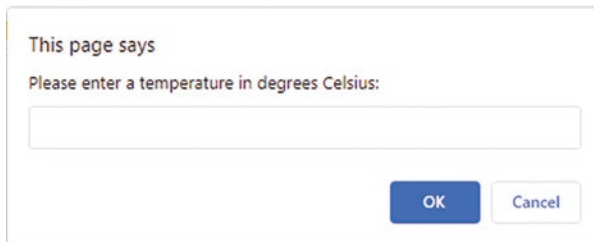
```
celsius = prompt('Please enter a temperature in degrees
Celsius: ');
console.log(`${celsius} degrees Celsius are ${Number(celsius) +
273.15} Kelvin.`);
```



■ Fig. 32.2 `alert()` box in Google Chrome



■ Fig. 32.3 Confirmation dialog with `confirm()` in Google Chrome



■ Fig. 32.4 `prompt()` input dialog in Microsoft Edge

The input dialog that `prompt()` generates is shown in ■ Fig. 32.4.

You can also try this example without calling the `Number()` constructor. What happens? Can you explain the result? If not, go back a few pages to ► Sect. 31.3.1.

32.4 Output to an HTML Document/Web Page

32.4.1 Writing HTML Code into the Web Page

Let's now turn to the most important form of output, modifying the web page in which the JavaScript program is embedded.

The simplest way to modify a web page from JavaScript is to use the `document.write(html)` method. It simply writes the string passed as argument *html* into the web page at the point where the script is embedded in the web page. The string, which may contain text as well as HTML instructions (tags), is inserted into the

source code of the HTML page as if it had originally been written there by the designer of the page.

To illustrate this, let's assume a (very minimalist) web page whose source code looks like this:

```
<!DOCTYPE html>

<html>

  <head>
    <title>Test Page</title>
    <noscript>Please enable JavaScript!</noscript>
  </head>
  <body>
    <h1>Our test page</h1>
    <p id="output"></p>
    <script src="script.js"></script>
  </body>

</html>
```

The body of this web page contains nothing but a header (**<h1>...</h1>**), an empty paragraph (**<p>...</p>**), and the reference to the script.

The JavaScript program **script.js** embedded in the web page looks like this:

```
var random = Math.round(Math.random()*100, 0);

document.write(`
```

As you can see, we use the template literals familiar from ► Sect. 32.2 here to achieve a simple representation of the HTML string to be written to the web page. Instead, of course, we could have written **document.write('<p>A random number between 0 and 100: ', random, '</p>')**; which is a little more cluttered.

Now, each time you refresh the display of the web page in the browser, the JavaScript code is executed, drawing a new random number each time. The web page that is shown in the browser after our script has been written into it using **document.write()** then looks like this:

```

<!DOCTYPE html>

<html>

  <head>
    <title>Test Page</title>
    <noscript>Please enable JavaScript!</noscript>
  </head>
  <body>
    <h1>Our test page</h1>
    <p id="output"></p>
    <p>A random number between 0 and 100: 62.</p>
  </body>

</html>

```

Our script modifies the web page by adding an HTML element. This practically results in a “new” web page, which is then displayed in the browser. You can view the source code of the page in the browser.

Now, of course, you won’t always want to output something new at the current point in the script, but you may want to change existing elements of the web page. For example, we might want to change the headline of the page. But our script is “too far down” in the web page, so we can’t get to the headline. So, there should be a way to change *any* element of the web page from anywhere. This way does exist. It is via the *Document Object Model (DOM)* of the web page, which we deal with in the next section.

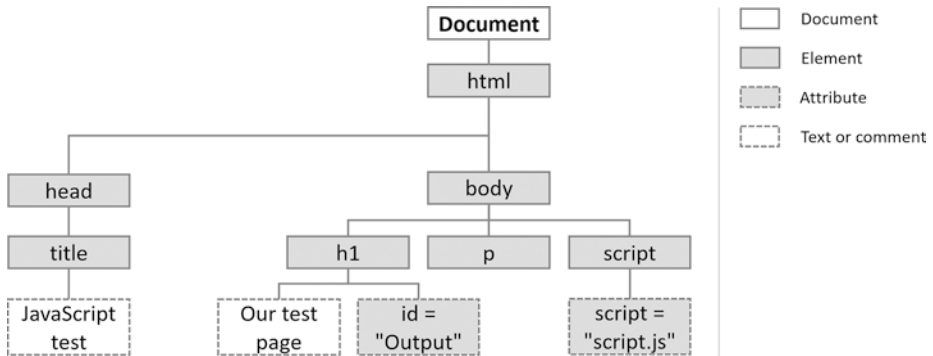
32 32.4.2 The Document Object Model (DOM)

In the HTML repetition in ► Sect. 29.1.1, we saw that the web browser reads the HTML file, and internally converts it into a representation called the *Document Object Model* or *DOM*. This is a hierarchical representation of the document structure. The nodes in the structure are either:

- the document itself (highest node),
- the individual HTML elements such as **title**, **body**, **h1** or **p**,
- any text associated with the elements (the **title** and **h1** elements of our example website from the previous section have directly associated text), or
- the attributes of the HTML elements like the **src** attribute of the **script** element.

Graphing the hierarchical structure of these elements results in a representation of the Document Object Model of the Web page as shown in ■ Fig. 32.5.

The individual nodes of the document object model are represented in JavaScript by corresponding objects. With the help of these objects, we can edit the



■ Fig. 32.5 The Document Object Model (DOM) of our sample website

nodes and in this way change, for example, the text that is displayed in the **h1** heading of the web page. To do this, we just must get hold of the right element. This is not easy, because there could be several occurrences of each element type (for example **h1**) in the document. So, the trick is to “target” the one element we want to edit. This is exactly what we will deal with in the following sections.

By the way, the highest node in the Document Object Model, which represents the entire document, is represented in JavaScript by an object we already worked with in the previous section: **document**. As you recall, we had used its **write()** method to write HTML code to the web page at the point where the JavaScript program is embedded.

32.4.3 Selecting DOM Nodes by Their Properties

■ Selecting an HTML Element by Its ID

The simplest way to capture an HTML element (i.e., a node of the Web page’s Document Object Model) is to address it by its **id** attribute. In our example HTML document from ► Sect. 32.4.1, you can see a paragraph element (**p**) with the following code, which we had not used before:

```
<p id="output"></p>
```

The element does not carry any text or other HTML elements (there isn’t anything between **>** and **<**), but it has an **id** attribute that we can use to access it. The **getElementById(id)** function of the **document** object is used for this purpose. The statement

```
var outputField = document.getElementById('output');
```

creates an object **outputField** that corresponds to the **p** element of our web page and through which we can edit the element on the web page. You can quickly see that **outputField** is a real object with properties and methods if you enter **outputField** (i.e., object identifier followed by the dot operator) and the popup window with the properties and methods of the object opens.

The properties and methods offered depend, of course, on the type of object. In our example, we are dealing with an **HTMLParagraph** object. Analogously, there is a whole range of other object types for the various HTML element types. Each of these object types may bring their own specific properties and methods relevant to the particular type of HTML element. What they all have in common, however, is that they are derived from the object type **HTMLElement** and therefore share certain properties and methods.

All HTML elements can have an **id** attribute, the value of which (as is usual with HTML attributes) is written in quotation marks. When working with **id** attributes, it is only necessary to ensure that each ID only occurs once in the document, so that it can be used to uniquely identify the respective element.

■ Selecting an HTML Element by Its Type

HTML elements can be grabbed not only by their unique ID, but also by their type. The method **document.getElementsByTagName(type)** is used for this purpose. Note, however, that an HTML document can have more than one element of each type. Accordingly, **document.getElementsByTagName()** also returns an *array* with all elements found. If you take a closer look, you will see that the method, unlike **getElementById()**, is a plural—it has an s in its name—not without reason!

To illustrate this, let's rebuild our script from ► Sect. 32.4.1 so that *all* **p** elements are grabbed:

```
var random = Math.round(Math.random()*100, 0);
document.write(`<p>A random number between 0 and 100:
  ${random}</p>`);

var pElements = document.getElementsByTagName('p');
console.log(pElements.length);
```

If you open the JavaScript console, you will see that the length of the **pElements** array is 2. But why two elements? Doesn't our HTML document contain only a single **p** element? Where does the second element come from? This second element is the one we create by using the **document.write()** statement in our own script.

We can now work with these elements. For example, we can use the **innerText** property to display the contents of the text element attached directly to the HTML element created by our script in the Document Object Model. This element is the second in the array, that is, the one with index 1:

```
> pElements[1].innerText  
"A random number between 0 and 100: 62."
```

■ Selecting an HTML Element by Its CSS Class

In a similar way to the element type, HTML elements can also be grabbed by the value of their **class** attribute. As you will recall from the HTML repetition in ► Sect. 29.1.1, HTML elements can be grouped together using their **class** attribute, for which specific formatting and presentation settings can then be made in a Cascading Style Sheets (CSS) file. In this way, formatting and presentation instructions can be defined that are not applied to all elements of a type (for example, to all **p** elements), but only to some of them.

Using the **document.getElementsByClassName(class)** method, you can get an array of all HTML elements whose class attribute corresponds to **class**. Our simple example from above does not work with CSS, so none of the HTML elements have the **class** attribute.

32.4.4 Selecting DOM Nodes via the Hierarchical Structure of the Document

The hierarchical structure of the Document Object Model can be used to capture other related nodes starting from one node. For this purpose, all node objects offer a number of predefined properties.

■ Child Elements of a Node

The following example shows how, starting from the **body** element of the HTML document, we can grab all DOM nodes that are directly at the level *below*:

```
var bodyElements = document.getElementsByTagName('body');  
var bodyChildren = bodyElements[0].childNodes;  
  
for(var i = 0; i < bodyChildren.length - 1; i++) {  
    console.log('Node no.', i)  
    console.log('Node name:', bodyChildren[i].nodeName);  
    console.log('Node type:', bodyChildren[i].nodeType, '\n');  
}
```

To do this, we first select the body element; more precisely, we grab *all* body elements using the **document** method **getElementsByTagName()**. Of course, there is only one of them in our web page. Nevertheless, **getElementsByTagName()** always returns an array. We access its first element in the next line. We make use of its **childNodes** property. **childNodes** is one of the read-only arrays that every DOM

node automatically has, and contains the “child nodes”, those nodes that are hierarchically directly below it, i.e., in our case, whose *parent* element is **body**.

In a **for** loop (which we discuss in detail in ► Sect. 35.1), we go through all the child elements and output two of their properties to the JavaScript console, their name and their type. The output to the console would then look like this for our example page:

```
Node no. 0
Node name: #text
Node type: 3
Node no. 1
Node name: H1
Node type: 1
Node no. 2
Node name: #text
Node type: 3
Node no. 3
Node name: P
Node type: 1
Node no. 4
Node name: #text
Node type: 3
Node no. 5
Node name: SCRIPT
Node type: 1
```

The name of the node corresponds to the identifier of the HTML element; for text nodes we find **#text** as name. As you can see, there are several text nodes attached to **body**. In our sample web page, there are only other HTML elements in the body, but there could be text before and after each of them. These texts only are empty in our sample web page. In fact, they contain tab indentations to better emphasize the hierarchical structure of the HTML source code. These tab characters are represented by the text nodes seen in the output above. For the sake of clarity, we have omitted their representation in our Document Object Model in ■ Fig. 32.5.

Text is also contained in the **h1** heading of the document (“Our test page”), but it is in a text node that is hierarchically *one level lower*: It is not attached to the **body** element of the web page, but to the **h1** element and is therefore not captured by our traversal of the child nodes of the **body** element.

■ Querying Attributes

The same applies to the *attributes* of the **p** element and the **script** element. They are hierarchically attached to the **p** and **script** elements and are therefore not direct children of the **body** element (rather its grandchildren, so to speak).

However, there is a peculiarity with the attributes: they are not contained in the array **childNodes** as own nodes. You can easily check this by typing **bodyChil-**

dren[5].childNodes into the console (the 5th child node of **body** is the **script** element). **childNodes** is empty! First of all, this is understandable insofar as there are no other HTML elements or text nodes hanging hierarchically below the **script** element. But the attribute **src** (the name of the script file) is attached to the **script** element, and this attribute is also a node in the Document Object Model of our web page. Nevertheless, this node is not included in **childNodes**. Attributes are mapped differently in the form of the array object **attributes**. **bodyChildren[5].attributes[0]** represents the first attribute of our **script** element, **src**. The name of the attribute and its value can be accessed with the properties **nodeName** and **nodeValue**, in our example with **bodyChildren[5].attributes[0].nodeName** (which would then return **"src"**).

There is a second way to access the attributes of an HTML element. The attributes are also properties of the element object. Because **bodyChildren[5]** is our **script** element, we can access the *value* of its **src** attribute directly with **bodyChildren[5].src** (the name is already in the attribute identifier). As in many cases, the value of the **src** property of the **bodyChildren[5]** object is a simple string.

■ Identifying the Type of the Node

Let's turn again to the output above. Two more things stand out: First, that the nodes are mapped into the array in the order in which they appear in the document; this is handy if you want to blueprint the child nodes of **body** from top to bottom. Second, that there appear to be two node types, 1 and 3. Nodes of type 1 are the HTML elements, nodes of type 3 are the text nodes. Attributes have node type 2; thus, the **src** attribute of our **script** element also has a **nodeType** property:

```
bodyChildren[5].attributes[0].nodeType
```

■ Finding the Parent Element of a Node

Besides **childNodes**, there is another important object for navigating within the DOM structure, and that is a property of each DOM node: **parentNode**. **parentNode** is the counterpart of **childNodes**, it specifies the node to which the current node (the one whose **parentNode** property we are querying) is hierarchically immediately subordinate. Unlike **childNodes**, **parentNode** is not an array, but a single node, because each child node has exactly one parent node under which it hangs.

? 32.1 [5 min]

Suppose you have an object **elem** in JavaScript that represents an HTML element of a web page. How can you access the sibling elements of **elem** (including **elem** itself), that is, all HTML elements that are at the same hierarchical level as **elem** in the HTML document?

32.4.5 Changing HTML Elements

■ Injecting HTML Code Directly into an Object

All HTML element objects have the **innerHTML** object property in JavaScript. It is a string that contains all the HTML code that hangs hierarchically below the respective HTML element.

Let's look at the example of our web page from ► Sect. 32.4.1. Its body element has, as we can easily check in the console, the following **innerHTML** property:

```
> document.body.innerHTML
"
  <h1>Our Test Page</h1>
  <p id="output"></p>
  <script src="script.js"></script>
  <p>A random number between 0 and 100: 31.</p>
"
```

Of course, we can also edit this property and thus “inject” HTML code into an object, so to speak. Suppose we didn't want to output our random number in a new paragraph element (**p**) that the script would simply insert at the point on the website where it runs, but instead we wanted to write the number into the existing paragraph element with the ID **output**. To do this, our script **script.js** would simply need to grab the paragraph element by its ID and then “inject” it with the output containing the random number into that element. With this, our script **script.js** could then look like this:

```
rndNum = Math.round(Math.random()*100, 0);
pOutput = document.getElementById("output");
pOutput.innerHTML = 'A random number between 0 und 100: '
  + rndNum + '.';
```

However, we might want to highlight our output in boldface and embed it in a **strong** element for that purpose. To do this, we would simply need to modify the last line of our script as follows:

```
pOutput.innerHTML =
  '<strong>A random number between 0 und 100: '
  + rndNum + '</strong>';
```

If you now look at the source code of the page displayed in the browser, or simply use **document.body.innerHTML** to view the HTML content of the **body** element in the console, you will find the following in it:

```
"  
  <h1>Our Test Page</h1>  
  <p id="output"><strong>A random number between 0 und 100:  
  33.</strong></p>  
  <script src="script.js"></script>  
"
```

So, as you can see, we successfully “injected” a snippet of HTML code inside the **p** element output.

■ Changing Attributes of Elements

Besides using the **b** element, there is another way to display text in bold, and that is with the help of the CSS property **font-weight**. If you set this to **bold**, the text is also displayed in boldface.

The CSS properties of an HTML element can either be described via a separate CSS file (a path we will take in one of the two examples with which we conclude this chapter) or set directly using the **style** attribute of an HTML element. Thus, if we wanted to display the entire contents of our paragraph element **output** in bold-face type, we would have to modify the opening HTML tag as follows:

```
<p id="output" style="font-face: bold;">
```

The name of a property and its value are always separated by a colon in CSS. Strictly speaking, we would not have needed the final semicolon. However, it is possible to place several CSS property assignments in the **style** attribute, which must then be separated from each other with semicolons. So, it doesn’t hurt to have a semicolon at the end of the **style** attribute.

But how can we now set the **style** attribute with the help of our JavaScript code? The approach should not come as a surprise to you after the previous section. After all, we already know that the HTML element object (which we bound to the **pOutput** variable in the script above using **getElementById**) has a matching property for each standard attribute. And, of course, we can assign those. It’s important to note that **style** is an object with numerous properties that reflect CSS properties. The names of the properties are the same as the CSS properties, but the CSS-typical hyphen is omitted and replaced by capitalization. Thus, **font-weight** becomes **fontWeight**:

```
pOutput.style.fontWeight = 'bold';
```

If you now—so that we can also observe the effect—reset the “injection” of the HTML code to

```
pOutput.innerHTML = 'A random number between 0 und 100: '
+ rndNum + '.';
```

and reload the web page in the browser, you will notice that the operation was successful and the output is displayed in boldface. Now, the **style** attribute is a special case in that the **style** property of the HTML element object cannot simply be assigned a value like **"font-face: bold;"** in JavaScript, but instead must be worked with the individual properties of the **style** object.

However, many attributes have a corresponding property in the HTML element object in JavaScript that can be assigned directly. The **align** attribute, for example, that determines the text direction, could be assigned with the statement

```
pOutput.align = 'right';
```

so that the text within the paragraph is right-aligned (try it out!). Unlike **style**, the corresponding property of the HTML element object (in our case the **align** property of **pOutput**) is not an object, but a simple string that can easily be assigned. The statement therefore results in the code of the **p** element now looking like this:

```
<p id="output" align="right">
```

? 32.2 [10 min]

32

Create a simple web page with a **p** element (paragraph) containing a text. From within a JavaScript program, change the font background color (CSS property **background-color**) so that the text is highlighted in light yellow (color components red: 255, green: 255, blue: 204).

32.4.6 Adding and Deleting HTML Elements

■ Adding HTML Elements

In the last section, we used the **innerHTML** property of HTML element objects to “inject” HTML code directly into the elements, and in this way we were also able to create new HTML objects as child elements of the edited parent elements by simply writing their full HTML code inside the parent element. In this section, we’ll look at another method of creating HTML elements by generating corresponding HTML element objects in JavaScript and then “hooking” them into the document at the desired location.

New HTML element objects can be easily created in JavaScript using the **createElement()** function of the **document** object. As argument, this function expects the identifier of the HTML element type of which an instance is to be created.

To create a new subheading (HTML element type **h2**), for example, the following statement is sufficient:

```
var heading2 = document.createElement('h2');
```

Now we can edit the properties of the new element as desired, for example, using the **align** attribute to make the heading text right-aligned:

```
Headline2.align = 'right';
```

Of course, the heading also needs a text. We could set it with the already known **innerHTML** property of our element object, or we could use the **innerText** property. **innerText** usually represents all the text that is attached to the element itself or its children in text nodes; this becomes clear if you take a look at the **innerText** property of the **body** element of our web page. However, we can also use **innerText** to add text to our newly created element (which has no child elements):

```
heading2.innerText = 'Another exciting section';
```

After we have sufficiently configured our new HTML element object, we still need to add it to the web page where we want it. The easiest way to do this is to grab the intended parent element and then add the new child element to it using the element object method **appendChild(newChild)**:

```
var bodyElem = document.getElementsByTagName('body')[0];  
bodyElem.appendChild(heading2);
```

Alternatively, if you don't want to simply append the new element to the existing child element at the back, you can control its position more precisely by placing the child element with the element object method **insertBefore(newChildElement, successorChildElement)**. This way, we could insert our new heading before the paragraph element **output**, which we represent in our program by the object **pOutput**:

```
bodyElem.insertBefore(heading2, pOutput);
```

? 32.3 [5 min]

Develop a piece of JavaScript code that you can run in the JavaScript console that adds the bold-formatted text “This is the end.” to the HTML body of a web page (at the bottom of the web page).

Try it out: Take a web page like ► wikipedia.com, open the developer tools in your browser and run your code. You will see that your HTML element has been added to the web page!

■ Deleting HTML Elements

Deleting HTML elements from the web page can be conveniently accomplished with the **remove()** method that all HTML node objects in JavaScript inherently provide. So, for example, to delete the paragraph element **output**, all we have to do is call **remove()** on the **pOutput** object:

```
pOutput.remove()
```

The element immediately disappears from the web page. Just like when adding and changing objects, the browser regenerates the display without the user or us as developers having to do anything.

32.5 Input with Forms

In the following, we will look at how JavaScript can be used to validate or otherwise process input from HTML forms. The next section first provides a brief overview of how forms work in HTML. If you are already familiar with this, you can—without missing anything—go directly to ► Sect. 32.5.2.

32.5.1 Forms in HTML

With the exception of dialog boxes, we have so far focused on how to output data in web pages, but not on how the user can enter data. This is exactly the purpose of HTML forms: They are used to accept data from the user. The data received is often sent to the web server that provides the page and processed there. This often involves the use of the PHP programming language, which was developed precisely for this purpose, but sometimes also involves server-side JavaScript programs. In such client-server situations, JavaScript usually has the task of validating the user's input on the client side before it is sent to the web server, for example, to check for any incorrect entries, stop the sending of the data (if necessary) and inform the user of the incorrect entries.

However, the data does not necessarily have to be sent to a web server. In fact, the form input can also be used as input for a JavaScript application; this application would then not be a validation mechanism, but in a sense the final recipient of

the data. The data would be entered for the sole purpose of being processed by the JavaScript program. The two examples with which this chapter concludes, a calculator and a color picker, fall squarely into the category of interaction between forms and JavaScript.

However, before we get into these applications, let's first look at how forms are constructed in HTML. Forms are always created by the HTML element **form**. This creates an initially empty form. Within the **form**, there can be any number of different **input** elements that represent different input options. The type of input represented by each element is controlled by the **input** attribute **type**. Consider the following simple (login) form as an example:

```
<form action="http://www.mysupernicewebsite.com/login.php"
method="POST">

  Username: <br>
  <input type="entry" value="" name="username"><br>

  Password: <br>
  <input type="password" value="" name="password"><br>

  <input type="submit" value="Login">

</form>
```

In this example, we create a form with a total of three input elements:

- an element of type **entry** for the username; this is a simple text input field
- an element of the type **password**, which is ultimately a special variant of the type **entry**, where the entered characters are masked by asterisks
- an element of type **submit**, a special button that sends the form content to the server

Besides these, there are a number of other types of input elements, for example **button** (“normal” buttons, the **submit** type has a very special function as a button), **radio** (radio buttons), **checkbox** which can then in turn be queried in the (checkboxes), **range** (sliders) and **textarea** (for multi-line text input). There are also input elements that allow the selection of dates (**date**), colors (**color**) or files to be uploaded (**file**), plus a range of other elements that support other input modes.

All elements have a **value** attribute that contains their current value; in the case of the text input fields, this is the text currently in the input field, and in the case of the **submit** button, it is the button label. This attribute is important because it can be used to determine the user's input, which is to be processed. Besides **value**, the input elements can also have a **name** attribute (as do all HTML elements). This is helpful if you want to access the input values on the server side. If, on the other

hand, we are only working with JavaScript on the client side, we can also access the elements as usual via an **id** attribute, which has been omitted in the above example for the sake of simplicity. The **name** is also used to group those elements that logically need to be evaluated together; this is especially true for the individual selection options in a group of radio buttons, where only one option can be selected at a time (we will take a closer look at this in an example below).

The behavior of the input elements can be fine-tuned with a number of other attributes; for example, the **required** attribute prevents the form from being submitted if the field in question is not filled in, and the **readonly** attribute prevents the user from making changes to the current value of the input element. Both attributes are of type **boolean**, so their possible values are **true** and **false**. Instead of, say, **required="true"**, however, you often see simply **required** in practice. The mere existence of the attribute is already evaluated as **true** and would be sufficient in this case to make the input element in question a mandatory specification.

In addition to these standard attributes, the various input elements can have other type-specific attributes. For example, the **range** input element, which represents a slider, has the attributes **min** and **max**, which describe the limits of the range within which a value can be selected with the help of the slider (which can then in turn be queried in the **value** attribute); checkboxes have a **boolean** attribute with **checked**, which specifies whether the checkbox should currently be checked or not.

So far, we have not discussed the attributes of the **form** element itself. The attribute **action** determines which address should be called (usually a PHP script) to pass the data to the server when the user triggers the submission of the form via the **submit** button. The **method** determines which mode should be used to transfer the data via the *Hypertext Transfer Protocol* (HTTP). The default value of this attribute is **GET**, but **POST** is usually used when sensitive data such as passwords is to be transferred.

The attributes **action** and **method** of the **form** element, as well as the **submit button**, with which the data can be sent, are however only needed if the entered data is to be transferred to a web server. The color picker example at the end of the chapter does not need any buttons at all, the calculator example uses buttons but no **submit** button. Both applications process the data directly in a client-side JavaScript program and can therefore do without any precautions for sending the data.

32.5.2 Accessing Forms from JavaScript

From within our JavaScript programs, we regularly want to work with the data that the user has entered into a form. To do this, we need to access the form elements from JavaScript to get to the data they contain, but we also need to find a way to link this access (and the subsequent processing of the data) to an action by the user. After all, in the area of web interfaces, we are also operating in an *event-driven* environment in which the user triggers events (for example, by clicking on a button) and our JavaScript application reacts to them.

To learn about the event control procedure and how to access the form data, let's look at our well-used example, the temperature conversion between degrees Celsius and Kelvin.

This application could have an interface where the user enters a temperature and decides whether to convert that temperature to degrees Celsius (they had entered a Kelvin temperature) or to Kelvin (they had entered a Celsius temperature). A click on a button starts the conversion and outputs the result.

The HTML code of such an interface looks like this:

```
<!DOCTYPE html>
<html>

  <head>
    <title>Temperature Conversion</title>
    <noscript>Please activate JavaScript!</noscript>
  </head>

  <body>
    <script src="kelvincelsius.js"></script>

    <h1>Temperature conversion Kelvin <=> degrees Celsius </h1>

    <form>
      <p>Temperature for conversion: <input id="temp"
        type="text" value="" size="5">
        <span id="unitLabel"> Kelvin</span></p>

      Convert to:<br>
      <p><input type="radio" name="direction" checked
        onchange="change('Kelvin') ">degrees Celsius</p>

      <p><input type="radio" name="direction"
        onchange="change('Celsius') ">Kelvin</p>

    <p></p>

    <input type="button" value="Convert"
      onclick="convert()">
    </form>

  </body>

</html>
```

Temperature conversion Kelvin <=> degrees Celsius

Temperature for conversion: Kelvin

Convert to:

☒ degrees Celsius

☐ Kelvin

■ Fig. 32.6 Kelvin-Celsius conversion with HTML form

This is also shown in ■ Fig. 32.6

The form includes a text input with the ID **temp** for *temperature*, two radio buttons (**direction**) to define the conversion direction and a button that triggers the conversion.

You will also see an HTML element that we haven't encountered before, namely **span**. **span** doesn't have any special function, but it helps us to give a text display its own ID so that we can address it from within our JavaScript program. On our **span** element **unitLabel**, we represent the unit in which the user enters the temperature. Every time he changes the selection regarding the conversion direction, this unit display must also change.

We have combined the two radio buttons via their **name** attribute: This means that they belong to one group, so only one of them can be selected at a time. Please note that the grouping of the radio buttons is realized via the **name** attribute and not via the **id** attribute. The difference is that the ID is a *unique* identifier. So there must not be two elements with the same ID. Our two radio buttons do not have an ID at all, because we can also address them by name from within our JavaScript program. The button that starts the conversion even has neither an ID nor a name. Neither is necessary, since the button triggers an action and thus triggers our JavaScript program, but we do not need to access it from within the program.

The **p** and **br** elements have a purely design function (**br** creates a simple line break), they help us to make the form a bit more visually appealing. So, as you can see, not only **input** elements can be placed in a **form** element, but you can use the whole HTML element variety (including tables and images) to build an attractive form.

However, we have not yet considered a very important part of the form: The **onchange** and **onclick** attributes of the **direction** radio button and the **Convert** button. The value of each of these attributes is a JavaScript function (an *event handler*), the name of the attribute is related to an event, namely to the event on whose occurrence the respective JavaScript function is to be called. For example, if the user clicks on our button, the **click** event is triggered. When the **click** event occurs, the browser automatically checks whether the **onclick** attribute is set and, if so, executes the JavaScript function specified there. In this way, we link our interface to the JavaScript program.

In our example, this looks like this:

```
function convert() {
    var temp = Number(document.getElementById('temp').value);
    var direction = document.getElementsByName('direction');

    if(direction[0].checked == true) {
        document.write('<p>${temp} Kelvin are
            ${temp - 273.15} degrees Celsius.<p>`');
    }
    else {
        document.write('<p>${temp} degrees Celsius are
            ${temp + 273.15} Kelvin.<p>`');
    }
}

function change(unit) {
    var unitLabel = document.getElementById('unitLabel');
    unitLabel.innerHTML = unit;
}
```

The whole program consists of only two functions, namely the two event handlers that are linked to the radio buttons and the main button of our application. The function **change()** is stored as an event handler with the **onchange** attributes of our radio buttons and is therefore always called when the **change** event occurs. This happens whenever the user clicks on one of the radio buttons (by the way, we could just as well have attached our event handler to the **click** event). When the event occurs, the function is called with one argument, which is the unit to be displayed on our **span** element **unitLabel** as the unit for the user input. In the HTML code you can see very clearly that the event handler is immediately given the desired unit as a parameter when it is called: **onchange="change('Kelvin')"**.

Inside the event handler **change()** we first select the **span** element with **document.getElementById()** and then replace the HTML code inside it; in our example this is just plain text without any further HTML coding anyway.

But now to our other event handler, **convert()**, which is called whenever the user clicks the “Convert” button. It has a very simple structure. First, we get the temperature by querying the **value** attribute of the **temp** input field, which we select by its ID. Note that we have to convert the value to a **number** variable, because we want to calculate with the temperature value. The form itself always saves the entered value as a **string**; there are ways to configure form input fields so that they only allow numeric input from the start, which we have not done here for simplicity’s sake. Next, we select the radio buttons by name. To do this, we use the **getElementsByName()** function. Pay attention to the plural **s** in **Elements**! Since the name—unlike the ID—is not necessarily unique, it can happen that you get several elements when selecting by name. And this is exactly the case in our example. The return value of the call to **getElementsByName()** is an *array* of elements, in our

example the two radio buttons. In the next step, we use the **checked** attribute of the radio buttons to check whether our first radio button (index 0!) is checked. Don't worry too much about the If-Else construct at this point, we will deal with this kind of program branching in detail in ► Sect. 34.1. If our first radio button is checked, it means that the user of our program wants a conversion from Kelvin to degrees Celsius. We then output this to the web page with **document.write()** using a template literal (if you are no longer familiar with template literals, scroll back a few pages to ► Sect. 32.2).

As you can see, our JavaScript program consists entirely of event handlers that lie dormant until they are triggered by the browser because an event has occurred to which they are linked. We will deal with events in more detail in ► Sect. 34.3, when we talk about program flow control. At this point, it is sufficient to understand the basic mechanism by which we can “wire” our JavaScript code to the interface controls.

? 32.4 [5 min]

Modify the Kelvin-Celsius conversion example so that the output is not done with **document.write()**, but to an HTML element of type **span** that must be built into the web page interface for this purpose.

? 32.5 [30 min]

Develop a simple application in which one can use a slider to change the font size of some sample text displayed on the application's web page. The font size can be set in HTML/CSS via the **fontsize** CSS style option and is specified in pixels (**px**); a valid font size setting would, for example, be **'18px'**. If you think you need some more “inspiration” before tackling this task, first deal with the examples in the following two sections.

32

32.6 Example: Simple Calculator

In this and the next section, we will develop two little example applications. First, we turn to a simple calculator. The calculator should be able to handle the four basic arithmetic operations and allow copying the result of the calculation to the clipboard. The input of numbers and operators shall be done either by buttons or by direct input via the keyboard.

Our application consists of three files:

- the HTML file **calculator.html**, which builds the web interface
- the Cascading-Style-Sheet-(CSS)-file **calculator.css**, which helps us to define the design of the buttons and the display, as well as
- the JavaScript program **calculator.js**, which provides the functionality for the interface

32.6.1 The Web Interface

Let's start with the HTML file `calculator.html`:

```

1  <!DOCTYPE html>
2  <html>
3
4  <head>
5    <title>Calculator</title>
6    <link rel="stylesheet" type="text/css" href="calculator.css">
7    <noscript>Please activate JavaScript!</noscript>
8  </head>
9
10 <body bgcolor="#282923">
11   <script src="calculator.js"></script>
12
13   <form>
14     <input id="display" type="text" value="0"
15       class="inputOutput">
16     <p></p>
17     <input type="button" value="C" class="normalButton
18       functionButton" onclick="clearDisplay()">
19     <input type="button" value="Copy" style="width:104px"
20       class="normalButton functionButton" onclick="copy()">
21     <input type="button" value="/" class="normalButton
22       functionButton" onclick="key('/')">
23     <p></p>
24     <input type="button" value="7" class="normalButton"
25       onclick="key('7')">
26     <input type="button" value="8" class="normalButton"
27       onclick="key('8')">
28     <input type="button" value="9" class="normalButton"
29       onclick="key('9')">
30     <input type="button" value="*" class="normalButton
31       functionButton" onclick="key('*')">
32     <p></p>
33     <input type="button" value="4" class="normalButton"
34       onclick="key('4')">
35     <input type="button" value="5" class="normalButton"
36       onclick="key('5')">
37     <input type="button" value="6" class="normalButton"
38       onclick="key('6')">
39     <input type="button" value="-" class="normalButton
40       functionButton" onclick="key('-')">
41     <p></p>
42     <input type="button" value="1" class="normalButton"
43       onclick="key('1')">
44     <input type="button" value="2" class="normalButton"

```

```

        onclick="key('2') ">
32    <input type="button" value="3" class="normalButton"
        onclick="key('3') ">
33    <input type="button" value="+" class="normalButton
        functionButton" onclick="key('+') ">
34    <p></p>
35    <input type="button" value="0" class="normalButton"
        style="width:104px" onclick="key('0') ">
36    <input type="button" value="." class="normalButton"
        onclick="key('.') ">
37    <input type="button" value="=" class="normalButton
        functionButton" onclick="calculate() ">
38    </form>
39    </body>
40
41    </html>

```

■ Lines 1–8

The usual HTML header, in which we first include the CSS file **calculator.css** and use the **noscript** element to make a provision for the case where the user has JavaScript disabled in his browser. The CSS file is integrated with the **link** element.

■ Line 10

We set the background of the web page by the **bgcolor** attribute to a dark shade, so that our calculator also looks stylish. After all, the eye does the math!

■ Line 11

We include the script **calculator.js**, which contains the actual functionality of the page. It is technically executed at this early point when the web page is loaded, but as we will see below, it only consists of functions that are called as event handlers, in other words, it is event-driven by the individual buttons. As long as these functions are not explicitly called, nothing at all happens on the application side when the script is executed. We could just as well have included the script in the **body** segment of our website. This would not affect the functionality of the application.

■ Lines 13 and 38

The rest of the **body** segment of our page is an HTML form that contains all the controls of the calculator.

■ Line 14

Here we define the display of our calculator. We give it the id **"display"** so that we can address it later from our JavaScript program. Its type is **text**, so it is an input field, after all, the user should also be able to enter numbers and operators

via the keyboard; in our calculator, he can type directly into the display. The initial value should be 0 as long as the user has not entered anything else. In addition, we give our display a class information with the help of the attribute **class**. For this class we defined **inputOutput**, there are special design instructions in the CSS file. So, we don't have to define the design at this point via attributes directly in the HTML code, (especially the CSS attribute **style**) but we outsource these settings to the separate CSS file, making our code clearer and easier to maintain. This way, if we wanted to change the design of the calculator display, we could simply include another CSS file via the **link** element in the header of the page, which also contains design statements for the class **inputOutput**, and the display of the display would change without us having to adjust the actual web page (the HTML document).

■ Lines 16–37

This is where the actual buttons come in. Through the **onclick** attribute, we assign a JavaScript function to each of the buttons, which is triggered whenever the user clicks on the button. This event handler function is either the function **key(character)**, which is passed as an argument to the character (digit or operator) that is on the key button, or the special functions **calculate()** to trigger the calculation when the user clicks on the equals sign, or **clear()** to clear the display, or **copy()** to copy the current display content to the clipboard. Initially, all buttons have **normalButton** as their **class** attribute, except for the special function buttons, such as, the operators and the copy, clear, and equal sign buttons. The function buttons belong not only to the **normalButton** class, but also to the **functionButton** class. The **functionButton** class ensures that these buttons get an orange coloring, while the buttons of the **normalButton** class get the default coloring (usually a shade of gray). We'll take a closer look at this later in connection with the CSS instructions.

As you may have noticed, the buttons themselves have not been given an **id**. Strictly speaking, this is a bit messy, but it's not necessary for us, since we don't need to address the buttons from our JavaScript code. It's the other way around: the buttons address our code by calling the corresponding event handler function when they are clicked by the user.

You can see the entire interface in ■ Fig. 32.7.

32.6.2 The CSS Design Instructions

In order to separate the basic structure of the interface described in the HTML file from the detailed design of the individual elements, we move the latter in our example to a separate CSS file **calculator.css**.

The CSS file defines design statements for three classes of objects, **normalButton** (basically all buttons), **functionButton** (the function buttons) and **inputOutput**



■ Fig. 32.7 The interface of the calculator application

(the display of our calculator). The design statements are presented per class in a CSS block enclosed by curly braces. The block is preceded by the *selector*, which specifies the HTML elements of the web page to which the design statements are to be applied. The preceding dot means “All objects whose **class** attribute *contains* the specified class”. In ► Sect. 32.6.1, we saw that function buttons belong to two classes, **normalButton**, which is the class for all buttons, and the special class **functionButton**. The CSS selector **.functionButton** therefore causes the appropriate design statements to be applied to these buttons.

```

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 {

```

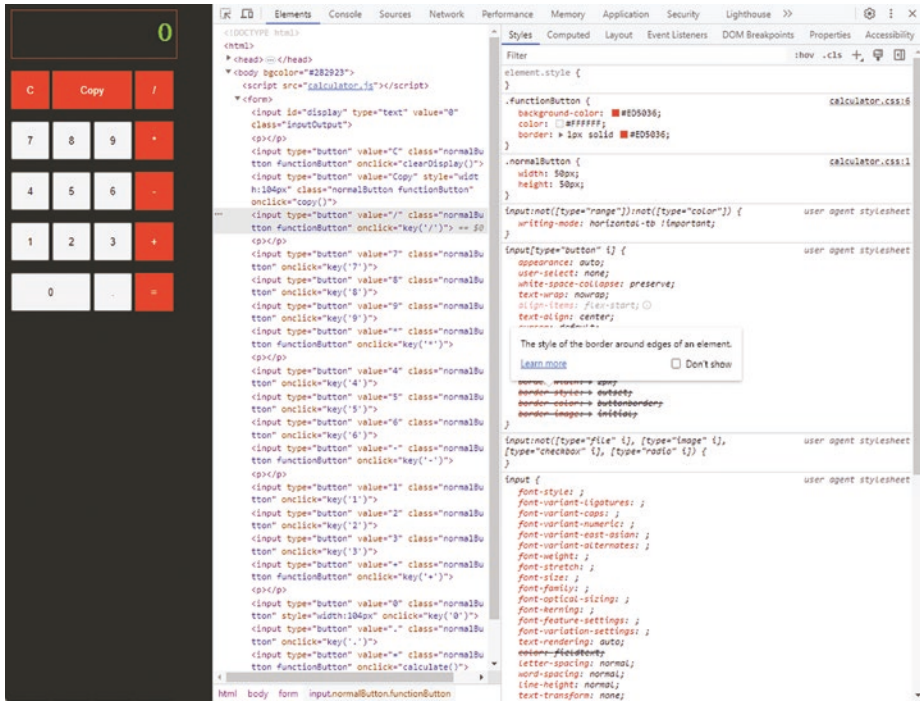
```
13 width:208px;
14 height:60px;
15 background-color: #282923;
16 color: #66FF33;
17 border: 1px solid #ED5036;
18 padding-right: 5px;
19 font-family: "Lucida Console";
20 font-size:32px;
21 font-weight: bold;
22 text-align: right;
23 }
```

■ Lines 2–3

For the class **normalButton**, and thus initially for all buttons, we define height and width in pixels (delete these design statements once from the CSS file and reload the page in the browser. What happens?)

■ Lines 7–9

For the class **functionButton**, we additionally define special background and foreground colors and the design of the button border (here, a one-pixel-wide solid line in the same color as the button background). Those buttons that belong only to the **normalButton** class (i.e., primarily the number buttons) naturally have a color scheme as well, but we haven't explicitly defined it; therefore, default values are used, which normally result in these buttons being gray. For the **functionButton** class, we override these default values with our own color specifications. Once you have loaded the page in the browser, open the Developer Tools and click on "Elements" (in browsers other than Google *Chrome*, the corresponding tab may be called something else). There you'll find a function button (in Google *Chrome*, at the very top left) that allows you to switch to a special Elements Inspection mode. In this mode, you can select an element on the web page by clicking on it and get more information about it in the Developer Tools. In the Element Inspector (■ Fig. 32.8), you see the selected element in the HTML source code of the page on the left and the CSS design specifications for the element on the right. The design defaults read from the bottom to the top. In our example (the function button with the division operator was selected), you see a whole series of CSS design properties that are preset with default values, first for the class **input** (CSS block at the very bottom), then, above, a block with additional properties specifically for **input** elements of type **button**. Then, above that, the two CSS blocks for the buttons of the classes we defined, **normalButton** and **functionButton**. Some properties are crossed out. This means that these properties are overridden by a more specific CSS block. For example, the **color** and **background** properties in the CSS block for input elements of type **button** are crossed out because they are defined differently in the special CSS block for buttons of class **functionButton** (and the selected button is such a button). This way you can



■ Fig. 32.8 Element inspector in Google Chrome

quickly see which (values for the) CSS design properties your element has and where they come from.

We assign a width directly to the buttons for copying the display content and the number 0 in the HTML code of the web interface using your **style** attribute. The **style** attribute contains CSS code that applies only to the element in question. This takes precedence over all other statements in the CSS stylesheet file, as you can see by examining the CSS statement hierarchy of one of the elements in the Element inspector.

■ Lines 13–22

This is where the design for the calculator display is set; including the background, font, color and size, and the text indentation from the edge of the element (*padding*).

32.6.3 The JavaScript Code

The JavaScript code of our application consists of the four event handler functions that we trigger from the HTML interface when the different buttons are clicked:

```
1  function key(character) {
2      var display = document.getElementById('display');
3      display.value = display.value + character;
4  }
5
6  function clearDisplay() {
7      var display = document.getElementById('display');
8      display.value = '0';
9  }
10
11 function copy() {
12     var display = document.getElementById('display');
13     display.select();
14     document.execCommand('copy');
15 }
16
17 function calculate() {
18     var display = document.getElementById('display');
19     display.value = Number(eval(display.value)).toFixed(6);
20 }
```

■ Lines 1–9

The functions **key(character)** and **clearDisplay()** are event handler functions that are called when the user clicks one of the corresponding buttons. As you remember, we always call the function **key(character)** from the HTML code of the web page with the character as argument (no matter if it is a number or an operator) that is assigned to the pressed key. By using this “trick” we only need one function for all buttons, instead of a special event handler for each button.

Both functions change the display on the screen. To do this, we first always “grab” the display element of our web page form using the **getElementById()** method of the **document** object. Then we change the property **value** of the display element, i.e. the text displayed on the input element; with **key()** we simply add the label of the pressed button to the current value (line 3).

■ Lines 11–15

To copy the current display content to the clipboard, we first select the existing text using the **select()** method of the **display input** element and then call the browser’s copy command.

■ Lines 17–20

When the user wants to perform the entered calculation and clicks the button with the equal sign, the value of the display is updated. Here we use the function **eval(expression)**, which evaluates the expression passed as a string, that is in our case: calculates it. We then convert the result into a number using **Number()**, which

we then format into a string representation with six decimal places using its `toFixed()` method.

32.7 Example: Color Picker

In this example we develop a small application that allows you to design colors according to the red-green-blue (RGB) scheme in a user-friendly way and to convert them into the HTML-typical hexadecimal encoding. Such applications, even much more sophisticated ones, can be found *en masse* on the Internet.

32.7.1 The Web Interface

The interface of our application is very simple. It consists of three sliders that allow you to change the color components of red, green, and blue, and a field that displays the resulting color as a hexadecimal code of the form `#RRGGBB`. The color currently selected via the sliders is used as the background color of the web page, so that the user can get a good impression of the color he has created.

```

1  <!DOCTYPE html>
2  <html>
3
4  <head>
5    <title>Color Picker</title>
6    <noscript>Please activate JavaScript!</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="input" value="#000000" readonly>
16        <p>Red:</p>
17        <input id="colorRedRange" type="range" value="255"
18          min="0" max="255"
19          oninput="adjustColor()">
20        <input id="colorRedOutput" type="input" value="255"
21          readonly>
22        <p></p>
23        <p>Green</p>
24        <input id="colorGreenRange" type="range" value="255"
25          min="0" max="255"
26          oninput=" adjustColor ()">

```

```

21     <input id="colorGreenOutput" type="input" value="255"
      readonly>
22     <p></p>
23     <p>Blue</p>
24     <input id="colorBlueRange" type="range" value="255"
      min="0" max="255"
      oninput=" adjustColor () ">
25     <input id="colorBlueOutput" type="input" value="255"
      readonly>
26 </form>
27
28 </div>
29
30 <script src="colorpicker.js"></script>
31
32 </body>
33
34 </html>

```

Let's look at the interface in detail:

■ Line 9

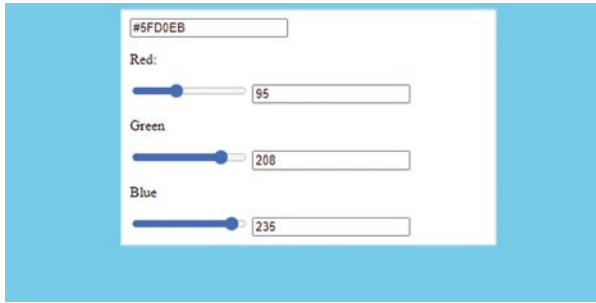
This time we give the **body** element of the page an **id** because we want to color the background of the page in the selected hue. To do this, we need to adjust the **bgcolor** attribute of the **body** element from within our JavaScript code.

■ Line 11

A **div** element in HTML simply marks a contiguous area on the page, ultimately a box that is initially invisible. We put all our controls in such a box. We can then color the **div** element white using its CSS attribute **style** (**#FFFFFF** corresponds to color portions for red, green, and blue of 255 each) and center it on the page (this is done with the value **0 auto** for the **margin** property). In addition, we equip the box with a margin of 10 pixels wide using the **padding** property, so that the controls have some free space to the edge of the box and fix the width of the box to 400 pixels.

■ Lines 13–26

The form inside the **div** box consists of a text input field in which we display the color code converted to hexadecimal notation (line 14). By default, the color should be white as long as the user has not selected a different color via the sliders. We also want the input field to be read-only, so we add the attribute **readonly**, which is a



■ Fig. 32.9 The Color Picker application interface

boolean attribute and therefore does not require an explicit value assignment for a true value (but we could have written `readonly="true"`).

Then we add three sliders (`colorColornameRange`), one for each color part *Colorname* of our RGB value. The sliders are of type **range** and are adjustable between 0 (**min**) and 255 (**max**). Each time the user moves them (event **oninput**), the event handler `adjustColor()` is fired, which, as we will see below, adjusts the hexadecimal color code in the **hexColor input** field, resets the background color of the page, and finally displays the new color portion in a (readonly) input field (`color-ColornameOutput`) next to the slider. By using the empty paragraph elements `<p></p>` we provide line breaks and thus better readability.

You can see the entire interface in ■ Fig. 32.9.

32.7.2 The JavaScript Code

The JavaScript file `colorpicker.js`, which we include in our web page in line 30, consists only of the event handler `colorAdjust()`, the function that is called every time the user moves one of the sliders.

The code in detail:

```

1  function adjustColor() {
2
3      var bodyElem = document.getElementById('bodyElem');
4      var hexColor = document.getElementById('hexColor');
5
6      var colorRedRange = document.getElementById('colorRedRange');
7      var colorGreenRange =
          document.getElementById('colorGreenRange');
```

```

8     var colorBlueRange =
        document.getElementById('colorBlueRange');
9
10    var colorRedOutput =
        document.getElementById('colorRedOutput');
11    var colorGreenOutput =
        document.getElementById('colorGreenOutput');
12    var colorBlueOutput =
        document.getElementById('colorBlueOutput');
13
14    var hexRed = Number(colorRedRange.value).toString(16);
15    var hexGreen = Number(colorGreenRange.value).toString(16);
16    var hexBlue = Number(colorBlueRange.value).toString(16);
17
18    if(hexRed.length == 1) hexRed = '0' + hexRed;
19    if(hexGreen.length == 1) hexGreen = '0' + hexGreen;
20    if(hexBlue.length == 1) hexBlue = '0' + hexBlue;
21
22    var hex = '#' + hexRed + hexGreen + hexBlue;
23    hexColor.value = hex.toUpperCase();
24
25    bodyElem.bgColor = hex;
26
27    colorRedOutput.value = colorRedRange.value;
28    colorGreenOutput.value = colorGreenRange.value;
29    colorBlueOutput.value = colorBlueRange.value;
30 }

```

■ Lines 3–12

Here we first create variables for the different elements of the interface that we need to address.

■ Lines 14–16

Next, we take the current values that the sliders are set to (using their **value** attribute) and convert the value to a hexadecimal string. As you recall from ► Sect. 31.3.2, the **toString()** method has an argument that specifies the base of the number system into which the number is to be converted for representation as a string, which in our case is 16, since we want to achieve hexadecimal representation.

■ Lines 18–20

With the variables **hexRed**, **hexGreen** and **hexBlue** we already have everything together that we need to display the hexadecimal color value in the format **#RRGGBB**. But there is a small complication: The value of the color component

variable could be a single digit (if the decimal value of the respective color component is smaller than 16). In this case we must prepend a leading 0, because hexadecimal color codes of the form **#RRGGBB** expect *exactly two* value digits per color component. Therefore, we check the length of the previously generated color part strings and add a 0 if necessary.

■ Lines 22–23

Now we can compose the hexadecimal value and display it in our (readonly) input element **hexColor**.

■ Line 25

We also assign the hexadecimal RGB value we just determined to the **bgcolor** attribute of the **body** element. So, as soon as the user activates one of the color proportion sliders, not only the display of the hexadecimal color value, **hexColor**, changes, but also the background color of the web page.

■ Lines 27–29

Finally, we display the color percentage values as decimal numbers in the (readonly) input fields provided next to the respective color percentage sliders.

32.8 Summary

In this chapter, we looked at how to input and output data using JavaScript. In particular, we have looked at the JavaScript console and how JavaScript applications can interact with a web page.

Be sure to take the following points from this chapter:

- The **console.log()** method can be used to output objects in the JavaScript console.
- Template literals allow you to include variables in strings in the form of placeholders that are replaced with the current value of the variable at the time the literal is created; the variable is written as **\$ {variable}**, and the entire literal itself is enclosed in backticks (**`**).
- Alternatives to the template literal are string substitution and concatenating strings and objects that can be output as strings using the plus operator.
- With **console.warn()** and **console.error()** you can output warning or error messages in the console.
- Interaction with the user can also be done via dialog boxes; in particular with **alert()** (display of a message), **confirm()** (dialog with “Okay” and “Cancel” as options) and **prompt()** (text input in a popup dialog).
- JavaScript is great for editing the HTML elements of web pages, making the web page dynamic through the JavaScript application.

- The easiest way to do this is to use **document.write()** to write HTML code to the current location where the script is running in the web page.
- The components of HTML pages, especially the HTML elements themselves, their attributes, and the text inside them, can be represented hierarchically as nodes of the *Document Object Model* (DOM).
- JavaScript allows elements of the DOM to be selected and edited in JavaScript as HTML element objects, and changes to these objects are immediately reflected in the rendering of the web page.
- DOM nodes can be identified mainly by their **id** attribute (with **document.getElementById()**), by their type (with **document.getElementsByTagName()**; note plural, an array is returned here because there may be multiple elements that match the criterion!), or their CSS class (with **document.getElementsByClassName()**; again, the return is an array).
- Element objects/arrays of element objects are returned that represent the HTML elements of the web page and whose properties are the attributes of those HTML elements.
- Also, starting from an element object, the DOM structure can be exploited to select hierarchically connected objects with properties like **childNodes** and **parentNode**.
- The **innerHTML** and **innerText** properties of an element object represent the HTML code contained in an HTML element (that is, the code of the hierarchically subordinate elements in the DOM) and the text contained in it, respectively.
- HTML elements can be created on the web page with **document.createElement(type)** and added below an element by calling its **appendChild(new_element)** method; the **insertBefore(new_element, before_child)** method of an HTML element adds **new_element** as a child element before another child element (**before_child**); **remove(element)** removes the HTML element object **element** from the Web page.
- In practice, the most important form of interaction with the user on web pages is the use of *forms*; they are created in HTML with the **form** element, so their components, namely the controls such as text input or buttons, are located between **<form>** and **</form>**.
- The elements of the forms are mainly of the HTML type **input** and are further differentiated by their **type** attribute; for example, **"text"** is the type for a text input field, **"button"** is the type for a button, and **"slider"** is the type for a slider.
- User actions with these controls (for example, clicking a button) can be linked via events to event handler functions in the JavaScript code, which are automatically called whenever the event associated with the action is triggered. The calls to the event handlers are given to the HTML elements as attribute values, where the attribute name is composed of **on** and the event, for example, **onclick** for the **click** event.

32.9 Solutions to the Exercises

■ Exercise 32.1

The sibling elements of **elem** (including **elem** itself) are nothing but the children of the parent element of **elem**. Thus, **elem.parentElement.children** returns an **HTMLCollection**, a collection of objects that correspond to the HTML elements. You can work with this object of type **HTMLCollection** practically like with an array.

■ Exercise 32.2

Your **p** element could look like this in the HTML source code of the web page:

```
<p id="paragraph1">Here's a note.</p>
```

You can now change the background color in your JavaScript program:

```
var pElem = document.getElementById('paragraph1');

pElem.style.backgroundColor = '#FFFFCC';
```

The **background-color** property is convenient to change because our element object **pElem** mirrors the **style** attribute in an object property of the same name. When doing so, please note that the CSS **background-color** property becomes the **backgroundColor** property of our element object's **style** property in JavaScript (that is, the hyphen is dropped, and any additional initial letters after the initial letter of the first component of the property name become uppercase). Also, of course, you need to convert the decimal RGB color components to hexadecimal numbers to get a valid color code of the form **'#RRGGBB'**. In doing so, 255 (red and green components) becomes **FF** and 204 (blue component) becomes **CC**.

■ Exercise 32.3

The code might look like this:

```
var bodyElem = document.getElementsByTagName('body')[0];
var pElem = document.createElement('p');

pElem.style.fontWeight = 'bold'
pElem.innerText = 'This is the end';
bodyElem.appendChild(pElem);
```

First, we grab the **body** element of the web page via its element tag (note: **getElementsByTagName()** returns an array of HTML nodes, since there may well be several copies of an element type in the web page—at least for elements other than **body**). Then we create a new **p** element, format it using its CSS properties, give it the **"This is the end"** text and add it to the **body** element of the web page as a new child element, i.e. append it to the existing children at the end.

By the way: If you include this script in a web page, as we usually do, i.e., via the **script** attribute of the body of the web page, then **"This is the end"** appears at the *beginning* of the web page! The reason is simple: the script is loaded and executed at the beginning of the body, at a time when there are no other elements in the body of the web page. The element that we add with the above code is thus the first and is logically displayed at the top of the page.

■ Exercise 32.4

First, the (still “empty”) **span** element must be inserted into the HTML code:

```
<span id="output"></span>
```

You only have to change the code of the function **convert()** slightly by first selecting the **span** element and then assigning the output to the **innerHTML** property:

```
function convert() {
    var temp = Number(document.getElementById('temp').value);
    var direction = document.getElementsByName('direction');
    var outputSpan = document.getElementById('output');

    if(direction[0].checked == true) {

        outputSpan.innerHTML = `<p>${temp} Kelvin are
        ${temp - 273.15} degrees Celsius.<p>`;
    }
    else {
        outputSpan.innerHTML = `<p>${temp} degrees Celsius are
        ${temp + 273.15} Kelvin.<p>`;
    }
}

function change(unit) {

    var unitLabel = document.getElementById('unitLabel');
    unitLabel.innerHTML = unit;
}
```

■ Exercise 32.5

The website could look like this:

```
<!DOCTYPE html>
<html>

  <head>
    <title>Font Sizes</title>
    <noscript>Please activate JavaScript!</noscript>
  </head>

  <body>
    <script src="fontsizecontroller.js"></script>

    <form>
      <p>Font size:
        <input id="controller" type="range" min="1" max="150"
          value="20" onchange="changeFontSize()">
      </p>
    </form>

    <span id="sampletext" style="font-size: 20px;">
      Some text in size 20
    </span>
  </body>

</html>
```

32

The corresponding JavaScript program in **fontsizecontroller.js** would then look like this:

```
function changeFontSize() {
  var size =
    Number(document.getElementById('controller').value);
  var text = document.getElementById('sampletext');

  text.style.fontSize = size + 'px';
  text.innerHTML = 'Some text in size ' + String(size);}
```



Functions & Methods: How Do I Work with Program Functions to Work with Data and Trigger Actions?

Contents

- 33.1 Working with Functions – 494**
 - 33.1.1 Definition of Functions – 494
 - 33.1.2 Return Values – 499
 - 33.1.3 Arguments and Parameters of Functions – 500
 - 33.1.4 Scope of Variables in Functions – 504
- 33.2 Working with Modules/Libraries – 506**
 - 33.2.1 Developing and Using Your Own Modules – 506
 - 33.2.2 Finding and Integrating External Modules/Libraries – 508
- 33.3 Frameworks – 509**
- 33.4 Summary – 510**
- 33.5 Solutions to the Exercises – 511**

Overview

Moving forward, we will examine functions, which are considered the backbone of programming in JavaScript, similar to their role in various other coding languages. After all, not only do you constantly work with predefined functions that JavaScript offers out-of-the-box or that you get from extension libraries, but you also regularly write your own functions; in particular, the event handlers that take a central role in event-driven JavaScript applications. So, it's no wonder that an intensive study of functions is a core part of our tour through JavaScript.

In this chapter you will learn:

- how to define and call functions
- how functions process arguments and return results
- that functions are also objects, and what consequences this has
- how the scopes of variables are cut in JavaScript—especially with regard to functions—and how this affects the accessibility of variables
- how to extend the available functions beyond the standard language with external libraries, and how to find suitable extension libraries
- what frameworks are and how they differ from libraries

33.1 Working with Functions

33.1.1 Definition of Functions

■ Defining Functions

Functions are defined in JavaScript with the keyword **function**. Any arguments are placed after the function identifier in round brackets, which are necessary even if the function is not passed any arguments at all. The program code that is executed when the function is called follows as a code block in curly braces. A simple function that just prints “Hello World” to the console would look like this:

```
function hello() {
    console.log('Hello World! ');
}
```

It could now be called from the program (or the console):

```
hello();
```

If you forget the round brackets when calling the function in the console, which can happen quickly if the function takes no arguments, you will be shown the source code of the function.

■ Functions as Objects

Functions in JavaScript are objects themselves, of type **function**, as can be easily verified:

```
> typeof(hello)
"function"
```

Because they are objects, they can also be assigned to other variable objects:

```
> greeting = hello
> greetings()
Hello World!
```

If we had written **greeting = hello()** here, we would have assigned the function *value* of **hello()** to a variable **greeting**, because **hello()** is nothing more than a call to the function of the same name. (As we will see below, this function value would be **undefined** because the function does not explicitly return a value).

As objects, they also have a number of methods and properties; the **toString()** function, for example, returns the source text of the function as a string:

```
> greeting.toString()
"function hello() {
  console.log('Hello World!');
}"
```

The round brackets must not be used after the function name (**greeting**) here, because we do not want to *call* the function and get its return value, but only access the function object's methods and properties.

The fact that functions are objects also becomes clear in other places where we can work with them, just as with any other object. In ► Sect. 31.5.2 we saw how objects can be created using the **var** keyword. This is exactly what we can do with function objects:

```
var hello = function() {
  console.log('Hello World!');
}
```

Here, we initialize a variable **hello** with a function expression, which is introduced with the keyword **function**. Subsequently, **hello** is a callable object and can be executed with **hello()**.

Because **hello** is now a real object, we can also add properties to it—which may seem a bit strange

```
> hello.counter = 5
> typeof(hello)
"function"

> hello.counter
5
```

The object remains of type **function**, but now has an additional property, **counter**. When we created objects using constructor functions in ► Sect. 31.5.5, we did something very similar (scroll back again to the constructor function of the **Product** object type). The constructor function ultimately wrote properties of the current object, using the keyword **this**.

Because functions are simply objects, they can also be used as properties in other objects, giving those objects callable *methods*. Suppose we wanted to develop an object that would hold a date, broken down into its constituent parts, and have a **display()** method that would output the date in a responsive format. We could define such an object as follows:

```
var shortdate = {
  day: 0,
  month: 0,
  year: 0,
  display: function() {
    console.log(this.year + '/' + this.month + '/' +
      this.day)
  }
}
```

33

Note the keyword **this**. We already encountered it in ► Sect. 31.5.5, and it establishes a reference to the current context in which (as here) a property or method is called. So, with **this.day** we access the **day** property in the current context, and that is the context of the object definition. If you omit the **this** keyword, JavaScript doesn't understand what **day** is supposed to be, because no variable of that name exists inside the **display()** function. After this declaration, we can work with the object by entering the date components and then calling the **display()** function:

```
shortdate.day = 14;
shortdate.month = 12;
shortdate.year = 2025;
shortdate.display();
```

This method call gives us the output **2025/12/14**.

So, as you can see, we can very easily populate an object with callable methods, because the methods are ultimately just properties of the objects, and they are properties of type **function**. These differ from other properties of the object only in that they are callable.

■ Advanced Topics I: Functions in Functions

A special feature of JavaScript is that functions can also be defined within functions. A simple (though admittedly not very meaningful in terms of content) example:

```
function helloWorld() {  
  function hello() {  
    console.log('Hello');  
  }  
  function world() {  
    console.log('World!');  
  }  
  hello();  
  world();  
}
```

Here we define within the function **helloWorld()** two further functions **hello()** and **world()**, which respectively cause an output in the console. Then both functions are called. In this way, two new lines **Hello** and **World!** are created in the console.

Alternatively, we could have created the (“sub”) functions by object assignments:

```
function helloWorld() {  
  hello = function() {  
    console.log('Hello');  
  }  
  world = function() {  
    console.log('World!');  
  }  
  hello();  
  world();  
}
```

Defining functions within functions may seem like a syntactical gimmick at first glance, but it has a practical use in this second variant when working with code modules.

By the way, what would have happened if we had written **this.hello = function...** and **this.world = function...** in the definitions of the (“sub”) functions? Then we would have developed a *constructor function* for a **helloWorld** object, which gives

two properties to the object, namely the two functions. Then we could have created objects of this type and accessed their two (function) properties (i.e., methods):

```
hi = new helloWorld();
hi.hello();
```

■ Advanced Topics II: Anonymous Functions

Functions normally have a name under which they can be called. However, there is also the possibility to define functions that do not have their own name, or *anonymous* functions.

A simple example is this:

```
(function() {
    console.log('Hello World')
}) ()
```

You will first see round brackets containing a function definition, but without a function identifier. The parenthesis expression returns a function (a function object) that we *immediately call again*. This can be seen by the pair of parentheses at the end. They are the usual parentheses that are also used when calling a “normal” function with a function identifier but with no arguments. This way, the function does not need any name and can still be called, but in our example only in the direct context of its definition, because otherwise we lack the “handle” to touch it with.

For “syntax gourmets”, there is a way to create such a “handle”, which we have used before. We assign the anonymous function to an object. To do this, let’s look again at an example from above:

```
var hello = function() {
    console.log('Hello World!');
}
```

Here we create a variable named **hello**, but the function we assign to it has no identifier. If you enter the name of the *variable* in the console, the source code of the function is displayed; however, no function identifier is visible in it:

```
> hello
f () {
    console.log('Hello World!');
}
```

Although it may seem a bit confusing, we have created an object called **hello** that represents a function, but the function itself is anonymous, so it has no name. However, if we define the function by using a function identifier, we again get a function object; this time, however, the function has a name:

```
> var hello = function hello() {  
    console.log('Hello World!');  
}  
> hello  
f hello() {  
    console.log('Hello World!');  
}
```

So, we don't necessarily have to call anonymous functions directly after their definition, we can also catch them in an object. Just like the “functions in functions”, anonymous functions are also useful when it comes to constructing entire code modules. In “everyday programming” it is functions with function identifiers that are mostly used.

33.1.2 Return Values

Functions can return objects using the **return** statement. The following function generates and returns a random number between 0 and 10:

```
function getRandomNumber() {  
    var randomNumber;  
    randomNumber = Math.round(Math.random()*10,0);  
  
    return randomNumber;  
}
```

The **return** statement can also be written like a function instead of using the keyword **return**, in our case as **return(randomNumber)**.

Functions that do not have a **return** statement do their job (in our case printing something to the console) but return **undefined**:

```
> function hello() {  
    console.log('Hello World!');  
}  
> res = hello()  
Hello World  
  
> res  
undefined
```

33.1.3 Arguments and Parameters of Functions

■ Foundations of Arguments and Parameters

The functions of mathematics are assignment rules that assign a value to one or more arguments. So far, we have only considered functions that have managed entirely without arguments. As a rule, however, functions become useful precisely because you can call them with arguments and thereby control their behavior or pass data to them for processing in this way.

Terminologically, we have so far been less selective than is often seen with JavaScript. Here, it is not uncommon to distinguish between *parameters* and *arguments*. Parameters are the abstract values listed in the function *definition* that a function accepts, while arguments are the concrete values that are actually passed to the function when it is *called*.

In JavaScript, parameters are specified in the function definition without a type. This means that the programmer must take care of any necessary type checks himself.

The previously used example of converting temperature from Kelvin to degrees Celsius would then look like this using function arguments (but without checking the type of the parameter):

```
function kelvinToCelsius(kelvin) {
    return kelvin - 273.15;
}
```

kelvin is a *parameter* of the function. If we call the function later, for example, with the Kelvin temperature 54, then 54 is the *argument* for the parameter **kelvin**. Multiple parameters would be separated by commas in the function definition.

■ Changing Arguments Within the Function

In JavaScript, arguments are always passed as *values* if the data types of the arguments are elementary data types, such as **number**, **string**, or **boolean**; thus, if you pass a variable to a function and the function changes the argument passed to it, the original variable does not change. In a sense, the function works with a copy of the value passed to it, not the variable itself. So, the passing is done by *value*.

However, if you pass a more complex object to the variable and make a change to it, the object passed to the function does change. The passing is then done by *reference*.

The following example illustrates the difference:

```
function Product(price, name) {
    this.price = price;
    this.name = name;
}
```

```
var chair = new Product(24.99, 'Garden chair');
var purchased = false;

function setPrice(article, price) {
    article.price = price;
}

function purchase(purchaseStatus) {
    purchaseStatus = true;
}
```

In the example, we again use the object type **Product**, whose constructor function we first call to create a **Product** object called **chair** and initialize some properties, including the price, which is set to **24.99**. We also create a **boolean** variable **purchased**, which indicates whether something has already been sold and is initialized to **false**. Following are the definitions of two functions: **setPrice(article, price)**, which changes the price of a product passed to it as its first argument, and **purchase(purchaseStatus)**, which takes a **boolean** variable as a sales indicator and sets it to **true** (or not, as we'll see). Let's try both functions in the console:

```
> setPrice(chair, 50.89)
> purchase(purchased)
> chair.price
50.89

> purchased
false
```

As you can see, the price of the chair changes, but the status indicator **purchased** keeps its old value. **purchased** is a *primitive value*, it cannot be changed as an argument of the function (**purchaseStatus**); the object **chair**, on the other hand, can very well be changed in the code of the function if it is passed to the function as argument **article**.

■ Calling Functions with Arguments

Function arguments can also be passed with their parameter name when calling the function; for example, we could call our Kelvin to Celsius conversion as **kelvinToCelsius(kelvin=54)**. By passing the arguments with their parameter name makes the assignment of the arguments to the parameters clear, this procedure also allows the arguments to be passed in a different order than the parameter sequence specified in the function definition.

JavaScript is very flexible about the number of parameters: if *too many* arguments are given to the function, the “excess” arguments are simply ignored. The

question of what happens in the case of *too few* arguments leads us to the topic of default values and optional parameters.

■ Working with Default Values and Optional Parameters

Parameters can be given a default value, as in most other programming languages, which is always used if no argument, i.e., no concrete value, is given for this parameter when the function is called.

Unlike many other programming languages, however, it is not this default value that makes a parameter optional in the first place. Instead, *all* parameters are optional in JavaScript. If parameters do not have a default value and yet are not specified in the function call, they are automatically assigned the value **undefined**. In our example, a call to the temperature conversion function from above as **kelvinToCelsius()** would be perfectly valid, but would result in the return value **NaN** (*not a number*), because in this case JavaScript assumes the value **undefined** for the parameter **kelvin**, and this cannot be used for arithmetic calculations.

■ Using Functions as Arguments of Functions

The arguments of functions can be of any object type. They can also be function objects themselves. This should not really surprise you after what we have discussed before. However, we want to illustrate this fact here with an example to show that this is a very useful feature of JavaScript.

Consider the following two functions, each of which concatenates a student's first name and last name in different ways:

```
function lastFirst(firstname, lastName) {
    return lastName + ', ' + firstName;
}

function firstLast(firstName, lastName) {
    return firstName + ' ' + lastName;
}
```

33

We could now develop a function **showMark()** that, in addition to showing the student's name, also takes care of showing an exam grade:

```
function showMark(firstName, lastName, mark, displayName) {
    console.log(displayName(firstName, lastName) + ': ' + mark);
}
```

We could now call **showMark()** as follows:

```
> showNote('Beth', 'Hathaway', 'A+', lastFirst);  
Hathaway, Bath: A+  
  
> showNote('Beth', 'Hathaway', 'A+', firstLast);  
Beth Hathaway: 1.0
```

Thus, **showMark()** uses the **displayName** function passed to it as an argument. This makes **showMark()** extremely flexible. Because as long as the passed function takes first and last names in that order, it can be used to display names in various ways.

■ Working with a Variable Number of Arguments

JavaScript provides an easy way to access the arguments of a function without using the name of the parameters. This is especially helpful when you don't know in advance exactly how many arguments the user will be passing. In the following simple example, we have a function that can be passed several strings as arguments, and that combines those strings as a news ticker with **+++** as the separator:

```
function newsTicker() {  
    var ticker = Array.from(arguments).join(' +++ ');  
    console.log(ticker)  
}
```

It can now be called like this, for example:

```
> newsTicker('Eagles at 49ers 35:38', 'Chiefs at Bengals  
23:20')  
Eagles at 49ers 35:38 +++ Chiefs at Bengals 23:20
```

Instead of two match results, we could have returned *any other number* of pairings. Our **newsTicker()** function would be able to cope with this, even though at first glance it receives no arguments at all! However, since JavaScript allows the function to be passed more arguments than it has parameters in its definition, we can just feed our news ticker as many football results as we want to our heart's content.

Arguments behaves like an array in some respects. For example, you can access the individual arguments passed to the function with **arguments[0]**, **arguments[1]**, and so on. However, **arguments** is not a true array. Other than the **length** property, which returns the number of arguments passed, **arguments** has none of the usual array properties and methods. For our purposes, we must first convert it to the **Array** type, which then provides us with the **join()** function to concatenate the individual elements. Here we use the **from()** function of the **Array object**, which converts an array-like object (like **arguments**) into a real array.

? 33.1 [5 min]

Define an object **Product** with product name and price as properties and develop a method for this object that applies a price discount to the product, which the caller of the method can specify as a parameter. If he does not specify a price discount, it should be assumed that the price is to be reduced by 20%.

? 33.2 [5 min]

Write a function that can be passed a **Product** object like in previous task and then applies the discount given as a parameter. The function should not be a method of the object. In what two ways can the “return” of the modified product object be done, and why?

? 33.3 [5 min]

Develop a function that takes an unspecified number of arguments and returns them as an alphabetically sorted array.

? 33.4 [5 min]

Write a function that does an output to the console. Assign the resulting function object to another variable. Then call the function using this other variable.

33.1.4 Scope of Variables in Functions

Especially in connection with functions, the range of validity of variables plays an important role. Variables declared with **var** *within* a function are *local* variables that exist only in the code block of that function. They cannot be accessed outside the function.

However, if a variable is accessed within a function *without* **var**, it is only created as a new local variable if there is no global variable of the same name, i.e., declared outside the function (“one level higher”, so to speak).

Consider the following simple example:

```
var factor1 = 3, factor2 = 5;

function multiply() {
    var factor2 = 7;

    result = factor1 * factor2;
    factor1 = 11;
    factor3 = 200;

    return result;
}

console.log(multiply());
```

```
console.log(factor1);  
console.log(factor2);  
console.log(factor3);  
This little program makes four outputs:
```

```
21  
11  
5  
200
```

Within the function **multiply()** a *local* variable **factor2** is created with **var**, whose value (7) differs from the value of the global variable with the same name, which was declared and initialized outside the function (5).

After calling the function, we output the value of the variable **factor2** to the console. In doing so, we automatically access the *global* variable **factor2**, because the local variable of the same name ceased to exist at the end of the **multiply()** function. However, it is the local variable with the value 7 that is used to calculate the multiplication within the function. In a sense, it shields the global variable **factor2**; as a result, it is not visible. When the variable **factor2** is accessed in the statement **result = factor1 * factor2**, the local variable is therefore automatically used. Only if no local variable exists, a global variable is searched for, and in the case of **factor1**, one is found. A new value is then assigned to this variable. Since the keyword **var** is not used, the assignment is made to the global variable. This change of the variable value is therefore also visible outside the function, as the second output in the console shows. If the assignment had been prefaced with **var**, instead of assigning a value to a global variable, we would have created a *new local* variable with the identifier **factor1**. The assignment of the value 11 would then have gone to this local variable, leaving the global variable **factor1** unaffected, shielded in its visibility by the local variable, and retaining its value after the **multiply()** function had been executed.

This is exactly what happens, as you can see in the third output, with the variable **factor2**, which is only apparently assigned a new value within the function; this new value goes to the new *local* variable **factor2**, so that the global variable is not changed in its value by the assignment.

Output number four is interesting. It accesses the variable **factor3**, which is used for the first time in our **multiply()** function. We already said that variables accessed within a function without the **var** keyword are global variables. And that's exactly how it is with **factor3**: by assigning **factor3 = 200**, we create a global variable, even though the assignment happens *inside* the function. And because **factor3** is a global variable, we can access the value stored in it outside the function without any problems.

By the way, the arguments that are passed to functions are always local variables. If a global variable of the same name exists, it is practically masked by the

function, it is invisible. If you access the variable with its identifier, you work with the local variable of the same name, i.e., the argument of the function.

? 33.5 [10 min]

What values do the variables **x**, **y**, **y1**, **y2**, and **z** have after executing the following program, and why?

```
x = 5;
z = 3;

function allOthers(x, y) {
    y1 = x;
    x = null;
    var y2 = y;
    z = 1;
}

allOthers(6,2);
```

33.2 Working with Modules/Libraries

33.2.1 Developing and Using Your Own Modules

Using modules means including code that is sourced out. Outsourcing code makes sense especially if you want to use the code in different programs. For example, if you have developed a practical function and want to use it not only in the program for which you originally designed it, but also in other programs, the easiest thing to do is to outsource this function to its own module and then include this module in all programs that are to access the function.

The easiest way to include another JavaScript file is to include it in the web page using the **script** element. Let's take a look at exactly that with an example.

In ► Sect. 33.1.2 we developed a function **getRandomNumber()** which returns a random number between 0 and 10. Let's assume that because this function is convenient and we want to use it in different scripts, we want to put it in its own module. To do this, we first create a new JavaScript file **mymodule.js**, in which we place the function. In addition, we define a variable called **fixedNumber** in our module. This will make our file **mymodule.js** look like this:

```
function getRandomNumber() {
    var randomNumber;
    randomNumber = Math.round(Math.random()*10, 0);
```

```
    return randomNumber;
}

fixedNumber = 4;
```

The function **getRandomNumber()** and the variable **fixedNumber** are accessed from another JavaScript program called **moduleapplication.js**:

```
document.write('A random number: ', getRandomNumber());
document.write('<p></p>');
document.write('A fixed number: ', fixedNumber);
```

This script in turn is embedded in a simple web page:

```
<!DOCTYPE html>

<html>
  <head>
    <title>Script with its own module</title>
    <noscript>Please enable JavaScript!</noscript>
  </head>
  <body>
    <script src="mymodule.js"></script>
    <script src="moduleapplication.js"></script>
  </body>
</html>
```

In order for the function and variable we defined in **mymodule.js** to be available in our actual program, which is in the **moduleapplication.js** file, we need to include the module in the web page as well, *before* **moduleapplication.js**. Since the scripts are processed in order, if we included the module later, its contents would not be known at the time we access it from **moduleapplication.js**. So, pay attention to the order!

Now, when we open the web page, we get an output like the following:

```
A random number: 2
A fixed number: 4
```

In exactly the same way, you could now include the module **mymodule.js** in other projects and access the function **getRandomNumber()** there without having to repeat the code of the function in your new project in the main code file.

JavaScript’s ability to work with modules has expanded more and more over time. While in the beginning this ability was limited to including source several files in the web page (as we just did), with newer language standards true modularization became possible. This, along with other things, allows more precise control over which objects are exported by modules and can then be imported again for use by other modules (and thus also by the JavaScript applications themselves). When importing, various options are also available to deal with naming conflicts that can arise when modules provide objects (for example, functions) whose identifiers are identical to the identifiers of objects already present in the importing code.

However, these considerations go well beyond the scope of an introduction to JavaScript development. For us it is sufficient to know that we can modularize our code by splitting it into individual script files and then embedding them into our HTML page.

33.2.2 Finding and Integrating External Modules/Libraries

Unlike for some other programming languages (such as Python with the *Python Package Index*, see ► Sect. 23.3.3), there is no central or quasi-official platform for JavaScript where you can find useful code to use when developing your own applications. Nevertheless, there are of course some “hotspots” of the very active JavaScript community; first and foremost *GitHub*, which we already had a look at in ► Sect. 13.2 and which is a platform for exchange and collaboration among developers based on the versioning tool *git* developed by Linux inventor *Linus Torvalds*. Here you will find a plethora of *repositories*, or code archives, with countless useful functions and classes.

Find out about the licensing situation from the **LICENSE** file in the repository before using third-party code! Most developers who make their work available on *GitHub* use one of the well-known standard licenses such as *GNU General Public License* or *Creative Commons* or *MIT*, each of which usually comes in various variants and versions. Conveniently, *GitHub* always provides you with a summary of these standard licenses, showing what you are allowed to do with the code and what you are not. So, you don’t have to read long legal texts on a regular basis at all to understand how you’re allowed to make use of other developers’ prior work. *GitHub* has already done that work for you.

A *GitHub repository* usually contains a large number of files. Don’t be surprised if the supposed descriptions of these files (in the middle column of a repository code view) seem strange to you. This column does not contain a description of the files at all, but comments that explain the last change to the file. In practice, the most important files are regularly located in the repository’s **dist** directory; they are the *distributable* files, i.e., those code files that are intended for distribution and production use. If you want to read the code, it is recommended to look at the **src** directory. This is because the code in **dist** is usually stripped of unnecessary characters (for example, spaces and comments) in order to keep the file size as small as

possible and to improve the performance of the web page that uses this code. Sometimes the code has been obfuscated (scroll back a few pages to ► Sect. 29.1.2, where we discussed obfuscation). You can, of course, also work with the code in the `\src` directory.

The easiest way to make the code usable for you is to download a ZIP file via the “Close or download” button and then unzip it on your computer. It contains the entire repository. While having the code available is necessary, it is also important to understand how to use it. The **README** file of the respective project and possibly other files in the `\doc` folder regularly provide information on this.

Besides *GitHub*, there are of course numerous other sources from which you can obtain JavaScript modules, such as ► [javascript.com](https://www.javascript.com). Also, a targeted internet search in which you follow the pattern “How can I...” quite often brings you to answers that point to modules that you can download from somewhere. Alternatively, you can ask ChatGPT something like “Which JavaScript modules are available to *implement a running news ticker on a website?*” (Replace the text in italics with your own topic of interest).

By the way, one of the most popular JavaScript libraries is *jQuery*, which especially simplifies working with the browser’s object model (i.e., selecting and modifying HTML elements, for example).

33.3 Frameworks

Frameworks play a big role in professional application development with JavaScript today. Frameworks differ from normal program libraries/modules in that they provide a wireframe for the application that calls the code written by the developer whenever that is necessary. When using libraries, it is the developer’s code that calls the library when it is needed. Therefore, in the context of frameworks, one speaks of an *inversion of control*. Frameworks play a major role in practice because they allow the programmer to concentrate on the core functionality of his application and to hand off other, more standard tasks such as handling logins and session management or retrieving data from databases and displaying it in template-based pages to the framework.

Some well-known JavaScript frameworks are *Angular*, *React*, and *Vue*. Looking at their structure and usage is far beyond the scope of this book. However, a prerequisite for using frameworks is knowledge of JavaScript basics, and these are exactly what we cover in this part of the book. So, after studying this part, you’ll have a solid foundation on which to venture into working with frameworks as well, should you so desire. In fact, for most purposes in the “private user area” (hobby programmer is not a nice phrase!) the use of standard JavaScript (perhaps enhanced by some libraries from *GitHub*) without the use of frameworks is sufficient, which although they are extremely powerful, also bring a certain “overhead” of work and enforce a structure that is not necessary for smaller projects.

33.4 Summary

In this chapter we have looked at functions and seen how functions are defined and used. In addition, we have looked at extending the scope of functions with external libraries and explored the question of what frameworks are and how they differ from “conventional” libraries.

Be sure to take the following points from this chapter:

- Functions are usually defined with the keyword **function**.
- You can take parameters (whose concrete values are called arguments when called) that can be given a default value.
- JavaScript is very flexible about passing arguments when calling functions; for example, a function can be called with more or fewer arguments than it has parameters; accordingly, practically every parameter to a function is optional.
- Arguments can be passed both as position arguments, that is, based on their position in the order of the parameters list, and as keyword arguments, that is, specifying the identifier of their parameter.
- Functions can return values with the **return** statement (or the **return()** function). Functions that do not explicitly return a value return **undefined**.
- In JavaScript, functions are objects of the type **function**.
- They can therefore also be assigned to variables. In particular, methods can be defined for objects in this way; the method of a JavaScript object is ultimately nothing more than an attribute of the object, namely a (callable) attribute of type **function**.
- Because functions are objects, they can be passed as arguments to other functions.
- Variables defined within a function with keyword **var** are local variables that cease to exist as soon as the execution of the function is finished; accordingly, they are not visible from the main program. Function arguments are also considered local variables. Any global variables with the same name are “shielded” from access by these local variables.
- If a variable is created within a function without using the keyword **var**, a global variable is created that is also visible outside the function body. If a global variable of this name already exists, this variable is accessed.
- Beyond JavaScript’s standard feature set, you can work with extension libraries; while there is no official source for these, as there is for many programming languages, there are platforms, such as ► javascripting.com, where developers provide a large set of libraries for a variety of purposes.
- Frameworks play a major role in practical (at least professional) JavaScript development because they lead to a significant reduction in workload when developing complex applications and allow the developer to focus on the essentials. They provide a wireframe in which the developer can embed his code. The framework organizes the flow of the application, and the developer’s code is invoked by the framework where necessary. Because here—unlike “normally”—the control of the application does not lie with the developer, but with the framework, one also speaks of an “inversion of control”.

33.5 Solutions to the Exercises

■ Exercise 33.1

```
Product = {name: 'Garden chair', price: 24.99 };

Product.discount = function(discountPercent = 20) {
    this.price = this.price * (1-discountPercent/100);
}
```

First, we create the object with its attributes **name** and **price**. We then add the function **discount()** to the object; more precisely, we assign a function object to the property **discount** that is to have the specified code. From now on, the function object can be called under the identifier **discount**, namely as the method **discount()**.

Similarly, we could of course have created our **Product** object with a constructor function, as we saw in ► Sect. 33.1.3. We could then have included the definition of the **discount** property in the constructor and simply assigned the function object to **this.discount** instead of **product.discount**.

■ Exercise 33.2

```
function discount(prod, discountPercent = 20) {
    prod.price = prod.price * (1-discountPercent/100);
    // return prod;
}
```

This function is not a method of the **Product** object. It simply takes a **Product** object and adjusts its price. This is possible because in JavaScript, objects are passed as function arguments *by reference*, which means that we have direct access to the passed object with the **prod** parameter. The case would be different if the argument was a *primitive*, such as a number or a string. In this case, the argument value would be passed *by value*; changes that we would make to these arguments would accordingly have no effect on the variable passed.

Instead of using the arguments passing *by reference*, we could also return the modified **Product** object to the caller using a **return** statement (commented out in the solution suggestion above). By the way, this possibility would also exist if the argument was a primitive and passed *by value*.

■ Exercise 33.3

The function could look like this:

```
function arrayCreateAndSort() {
    return Array.from(arguments).sort();
}
```

Here we use the **arguments** object, which contains the argument values passed for each function. Although it is not a real array itself, it can be converted into an array with the function **Array.from()** and then sorted.

After that, we could call the function like this, for example:

```
> arrayCreateAndSort('Hello', ' ', ' ', 'I', 'will', 'be', 'a',
'sorted', 'array')
[" ", "array", "be", "hello", "I", "one", "sorted", "to",
"will"]
```

■ Exercise 33.4

```
var myFunction = function() {
    console.log('This is the function that is actually
called.');
```

```
}
var myFunction2 = myFunction;

myFunction2();
```

Here we first assign a function expression to the variable **myFunction**; the variable **myFunction** is thus an object of the type **function**. Like any other object, we can now assign this object to another variable, in our example to the variable **myFunction2**. With this variable we can call the function. As with every function call, the round brackets must be specified, even if the function does not take any parameters.

■ Exercise 33.5

Our program results in the following values of the variables:

- **x = 5:** **x** is initialized as a global variable with the value 5. Within the function, the argument with the same name is then set to zero. However, the argument of the function are *local variables*. So, if we access **x** within the function, we do not work with the global variable **x**, but with the local variable of the same name, namely the argument **x** with which the function was called (and which ceases to exist at the end of the function body). Therefore, the value of the global variable remains unchanged.

- **y** does not exist: The variable **y** is also a *local variable* within the function. It ceases to exist after the function is fully executed. Since no global variable of the same name exists, we can no longer access a variable with the identifier **y** after the execution of our function is complete.
- **y1 = 6**: The variable **y1** is created within the function, but without the keyword **var**. Therefore, when **y1 = x** is assigned, a new *global* variable is created that is still available after the function has finished.
- **y2** does not exist: Similar to **y1**, **y2** is also created within the function by assignment, but using the keyword **var**. Thus, the assignment does not create a global variable, as was the case with **y1**, but a *local* variable.
- **z = 1**: **z** is a global variable that has the value 3 when the function is called. Within the function, **z** is then set to the value 1. Since the assignment is made without the keyword **var** (which would have created a new local variable with this identifier), we are therefore manipulating the *global* variable here.



Conditional Statements & Event Handling: How Do I Control the Program Flow and Make the Program React to User Actions and Other Events?

Contents

- 34.1 If-Else Constructs – 516
- 34.2 Switch-Case Constructs – 519
- 34.3 Events – 520
- 34.4 Summary – 524
- 34.5 Solutions to the Exercises – 525

Overview

Previously, we have encountered two ways to guide the course of program execution: through linear decision-making with If-Else mechanisms and by responding to actions with event handlers. We will proceed to investigate these two methods in greater detail.

In this chapter you will learn:

- how to use conditions with the help of If-Else constructs to branch into different parts of the program code
- how to formulate conditions and link them with each other using logical operators
- how to use the conditional operator? instead of an If-Else constructs, and when that makes sense
- how to use Switch-Case constructs to efficiently check many similarly structured conditions and branch accordingly
- how to use event handlers to process events
- what the main types of events are and how to evaluate information about an event that has occurred

34.1 If-Else Constructs

■ Formulation with the Help of the Keywords **if** and **else**

In the example of the Kelvin-Celsius conversion from ► Sect. 32.5.2, the function **convert()**, which was event-driven when the user clicked on the corresponding button, checked whether degrees Celsius should be converted to Kelvin, or vice versa—namely, the user could specify this with the help of two radio buttons. The code of the function looked like this:

```
function convert() {
    var temp = Number(document.getElementById('temp').value);
    var direction = document.getElementsByName('direction');

    if(direction[0].checked == true) {
        document.write('<p>${temp} Kelvin are
        ${temp - 273.15} degrees Celsius.<p>');
    }
    else {
        document.write('<p>${temp} degrees Celsius are
        ${temp + 273.15} Kelvin.<p>');
    }
}
```

Here you can clearly see the branching within the code of the function: **if direction[0].checked == true** (i.e. radio button number 1 is checked) *then* a conversion from Kelvin to Celsius takes place, otherwise (**else**) Celsius is converted to Kelvin.

The general structure of the If-Else construct looks like this:

```
if(condition) {  
  // Instructions  
}  
else {  
  // Instructions  
}
```

condition is an expression that can be evaluated to **true** or **false**. Often the expressions are *comparisons* like in the example. Note that in JavaScript (as in many other programming languages—Python is no exception) the equality operator is written as **==** and the inequality operator as **!=** (i.e. “not equal”).

In JavaScript, a comparison with **true** or **false** can also be omitted. So instead of writing **if(direction[0].checked == true)** as in the example above, **if(direction[0].checked)** would have sufficed, because **direction[0].checked** is an expression that can be evaluated to **true** or **false**; the comparison with **true** is therefore performed by default, even if we don’t write it explicitly. After all, other comparisons are also expressions that can evaluate to **true** or **false**. For example, instead of **if(x>5)** you could write **if((x>5) == true)**; **(x>5)** is an expression that can be **true** or **false**, depending on the value **x** takes.

By the way, the special values **null**, **undefined**, and **NaN** are always evaluated as **false** in conditions. Some functions may return one of these values, so if your expression to be checked consists of a call to such a function, you should consider whether your program really branches the way you want it to in case of such a return. Strings are always considered **true** (even if the string is **'false'** or **'0'**, so there is no implicit conversion here), numbers are always considered **true** unless they are the **0**, which is evaluated as **false**.

Please note that the condition—unlike in Python, for example—must always be enclosed in *round brackets*. Within the expression to be checked, additional parentheses can be used. This is especially advisable if you use complicated expressions with many operators and you do not know exactly in which order the operators are processed. To ensure a certain order then, it does not hurt to use enough parentheses. Better one bracket too many than one too few!

This also applies when you are working with compound conditions, i.e., conditions that are composed of several sub conditions. The sub conditions are then linked with each other using the logical operators **&&** (logical *AND*) and **||** (logical *OR*). For example, if you wanted to check whether the variable **age** is between 18 and 68, the appropriate condition would be: **if(age >= 18 && age <=68)**. If you also want your condition to take into account the sex of the person in question and always be true if the person is a woman or the person’s age is between 18 and 68, you would formulate the condition as follows: **if((age >= 18 && age <=68) || sex == 'f')**. Note the parentheses here, which we use to ensure that **(age >= 18 && age <=68)** is evaluated first as a sub condition. These parentheses would not have been necessary here at all, because the logical *AND* and *OR operators* are simply pro-

cessed from left to right in the order they appear in the code, and therefore there would never have been any danger of `age <= 68 || sex == 'f'` being evaluated as a sub condition; nevertheless, the notation with the parentheses makes it clear what belongs together here and thus increases the readability of the program code.

The third logical operator missing is the logical *NOT*, which is written as `!` in JavaScript and flips the truth of a statement. In contrast to the logical *AND* and the logical *OR*, the logical *NOT* is a *unary* operator, i.e., it requires only one operand (namely the expression whose truth value is flipped). The logical *AND* and the logical *OR* are binary operators that link two operands (here: logical expressions) with each other. With the help of the `!`-operator, we could write the condition `if((age >= 18 && age <=68) || sex == 'f')` as `if((age >= 18 && age <=68) || !(sex == 'm'))`. Here `!(sex == 'm')` means that the statement `sex == 'm'` should *not be* true. Conversely, `sex` must have the value `'f'` (at least if we assume two sexes as usual). By the way, the parenthesis around the expression to be negated cannot be omitted here, because otherwise the `!` operator would only refer to `sex`; `sex`, however, always has the logical value `true` as a string, which would be twisted to `false` by `!`. Thus, in case of a male, the expression `!sex == 'm'` is reduced to `false == 'm'`, a logical statement, which itself is false, because `"m"` as a string has the truth value `true`.

After the `if` condition and after the `else` keyword, which introduces the alternate branch that is executed whenever the `if` condition evaluates to `false`, there follows a block of code in curly braces. If the block contains only a single statement, the curly braces can be omitted (so you could have done that in our opening example).

The entire `else` branch can also be omitted. The minimal form of the If-else construct thus contains only an `if` branch. If the condition to whose execution it is linked does not apply, nothing happens. The program continues to run normally, starting with the first statement after the code block of the `if` branch.

■ Formulation with the Help of the Condition Operator?

A special kind of branching is the conditional operator `?`. It is a *ternary* operator, i.e., an operator that, unlike the *unary* operator `!` (the logical *NOT*) or the binary operators `&&` (the logical *AND*) and `||` (the logical *OR*), processes *three* operands, hence the name *ternary*. It checks a condition and returns either the first or the second as the result of two expressions, depending on whether the condition was met or not.

Consider the following example, which we will run in the console for simplicity:

```
> name = 'Anderson'
> sex = 'm'
> console.log("Hello", (sex == 'f') ? 'woman' : 'man', name)
```

Here, depending on whether the person is female or male, the salutation is adjusted accordingly. The condition `(sex == 'f')` is checked; if this is true, the operator returns the expression after the question mark, otherwise the expression after the colon. The general form of the condition operator is thus *condition ? returnIf :*

returnElse. This concise formulation is advantageous because it provides a more compact way of branching that is easier to integrate with other statements.

? 34.1 [3 min]

How can we easily show that strings in conditions always evaluate to **false**?

? 34.2 [5 min]

Reword the salutation example so that it no longer uses the conditional operator`?:`, but instead uses a traditional If-Else construct.

? 34.3 [5 min]

Rewrite the function `convert()` for converting temperatures between Kelvin and degrees Celsius so that Kelvin temperatures below 0 K and Celsius temperatures below absolute zero of $-273.15\text{ }^{\circ}\text{C}$ ($= 0\text{ K}$) are acknowledged with an error message.

34.2 Switch-Case Constructs

Sometimes you want to check many similar conditions at once; then a nested If-Else construct is possible, but it quickly becomes very confusing. This is why JavaScript, like many other languages, has a Switch-Case construct. For example, suppose you wanted to determine the number of days for a given month. With nested If-Else constructs, the readability and maintainability of the program code would suffer significantly. It is simpler with **switch**:

```
month = 'December';

switch(month) {
  case 'January', 'March', 'May', 'July', 'August',
       'October', 'December':
    days = 31;
    break;
  case 'April', 'June', 'September', 'November':
    days = 30;
    break;
  case 'February':
    days = 28;
    break;
  default:
    days = -1;
    break;
}
```

The **switch** statement is passed the expression to be checked in parentheses. The individual cases are then introduced with **case**. Several cases can also be checked at once, as we do here in the example. After the colon for each checked case, you see

the statements that are to be executed when that case occurs; in our example, only the variable **days** is set to the corresponding number of days. The statement block for each case ends with the keyword **break**.

During execution, the interpreter determines the value of the expression to be checked and jumps directly to the corresponding statement block. After the statement block has been processed, program execution continues after the **switch** code block which is enclosed in curly brackets. If none of the cases applies, the **default** block is executed, but this is optional and can therefore be omitted. If the **default** is omitted and none of the cases apply, execution continues directly after the Switch-Case construct.

34.3 Events

Besides the If-Else and Switch-case constructs, *events* are an important way to control the program flow; in fact, in JavaScript they are the most important form of flow control of all.

■ Insert Event Handlers Directly into the HTML Code

We have already worked with events when we attached event handlers to form elements such as buttons in ► Chap. 32, for example, to be able to react to a user click. In this section, we'll take a closer look at working with events.

The following HTML code is an excerpt from our color picker application from ► Sect. 32.7:

```
<input id="colorRedRange" type="range" value="255"
min="0" max="255" oninput="adjustColor()">
```

Here, for a **range input** element, i.e. a slider, we had set the property **oninput** to the event handler **adjustColor()**. Whenever the user moves the slider, this function we developed is called and can react to the user input.

The event covered here by an event handler is called **input**; by convention, the corresponding property of the HTML element is named **oninput**, so the event name is always prefixed with **on**. We have seen that elsewhere, for example, with **onclick**. The value assigned to the property is our event handler, or more precisely, the *call to the event handler*—easily recognizable by the round brackets. Instead of this call, we could have directly entered more JavaScript code here, for example **oninput = "alert('Change!'); console.log('Change!')"**. However, this approach is only recommended for very short code segments, and even then it's not really recommended, because it makes maintaining the code more difficult. So, normally at this point you will see the call to an event handler, as in our example.

■ Assign Event Handlers via HTML Element Properties in JavaScript Code

Next, let's look at the following simple example. First, the HTML document:

```
<!DOCTYPE html>
<html>
  <body>
    <form>
      <input id="myinput" type="text">
    </form>
    <script src="eventtest.js"></script>
  </body>
</html>
```

Here we just create a text input field with the ID **myinput**. You may have noticed that this time we have included the script at the very end of the body in the HTML document. The reason is that, as you will see in a moment, we access elements of the web page directly in the script. If we had included the script at the beginning, we would have missed out on these access attempts, because the elements of the pages do not yet exist at this early stage. Accordingly, the script would run into an error (try it out and watch the error messages in the JavaScript console).

The program logic is in the JavaScript file **eventtest.js**:

```
function mouseclick(e) {
  console.log('It was clicked. ');
  console.log('X: ', e.x, '\nY: ', e.y);
}

var inpField = document.getElementById('myinput');
inpField.onclick = mouseclick;
```

In this JavaScript code, first the input field of the web page is selected below. Then, the **onclick** property of this element is assigned the **mouseclick()** function defined above. More precisely, the **mouseclick** function *object* is assigned to the property. Note that there is no call to the function here, hence no round brackets after the function name.

The **onclick** property of our **inpField** object is an example of how JavaScript objects representing HTML elements have event handler properties whose names are formed using the same logic as for the HTML elements themselves, i.e., the **onevent** schema with **event** being the specific event that triggers the event handler.

The event we process here is the **click** event. The processing is done by our event handler **mouseclick()**. As you can see, **mouseclick()** takes a parameter, an *event object*, that describes the event in more detail. Depending on which event is processed, the event object is composed differently. In the case of the **click** event, the object has, among other things, the properties **x** and **y**, which indicate exactly where on the page the click occurred. We take advantage of this and output this informa-

tion in the console. If you want to know what properties the event object has for a particular event, just display it in the console with **console.log(e)**.

When we had “wired” the event handlers directly in the HTML code with the HTML elements, the event handlers were called without an event object, for example with **oninput="adjustColors()"**. We could not have passed an event object here at all. This was not a problem, because in these examples we had no need to access the properties of the event. However, if we had wanted to do that, we could have accessed the standard **event** object within our event handler, which would have provided us with the information. Strictly speaking, we would not need our parameter **e** at all and could always work with the standard object **event** instead. The fact that we can call functions that should be passed an event object without one, and even define the function without this parameter at all, is thanks to JavaScript’s flexible handling of function parameters, which we have already dealt with in ► Sect. 33.1.3.

The **click** event is not the only event related to mouse clicks. With **mousedown** and **mouseup**, two events are available that are always triggered when a mouse button is pressed or released. The event objects of these two events have the **buttons** property, which indicates which mouse button was used—with 1 for left mouse button and 2 for right mouse button. The **click** event is triggered after **mousedown** and **mouseup**. In the case of a double-click, the **detail** property of the event object of the second **click** event with the value 2 indicates that it was a double-click. Independently of this, the event **dblclick** is also triggered in this case. If you are not at all interested in the clicks, but rather in the mouse movements, you should take a closer look at the event **mousemove**. Since it is triggered by even the smallest mouse movement, it is better not to attach extensive code to this event.

■ Add Event Handlers with **addEventListener()**

The third way to attach an event handler to an event is to call the **addEventListener()** method of the JavaScript object that represents the HTML element in question. This is exactly what we do in the following example, where we process input in our input field using the **keypress** event. The JavaScript code of our **eventtest.js** file thus looks like this:

```
function showKeyPress(e) {
    if(e.key != 'a') inpField.value = inpField.value + e.key;
    e.preventDefault();
}

var inpField = document.getElementById('myinput');
inpField.addEventListener('keypress', showKeyPress);
```

The Event Handler is installed in this example with the **addEventListener()** method (event listener is a synonym for event handler). The method is called with the name of the event and the event handler object as argument.

Our event handler **showKeyPress()** causes the entered character to be displayed in the input field, but only if it is not an "a". So, in a sense, the a's are filtered out (try it out!). To do this, we make use of the **key** property of the event object, which contains the input character. Now, it is in the nature of an input field that the characters entered are also displayed. This standard behavior of input fields is provided by the browser. But if we want to filter the entered characters, we have to prevent this default behavior somehow. This is exactly what we do by calling the **preventDefault()** method of our event object. It prevents the browser from performing the default behavior normally associated with such an event. So, this way you could also, for example, suppress the default behavior of opening a context menu when you click the right mouse button.

Analogous to **click** and the two "detail events" **mousedown** and **mouseup** for mouse clicks, there are also special events for keystrokes with the names **keydown** and **keyup**. Unlike **keypress**, which is triggered only when displayable characters are entered, **keydown** and **keyup** are triggered whenever *any* key is pressed at all. Display the event object of **keydown** or **keyup** once with **console.log()**. You will see that special **boolean** properties are available with **ctrlKey**, **shiftKey**, and **altKey** to indicate whether any of the special keys were pressed. If a special key was pressed in combination with a character, e.g. <CTRL> + <S>, then **key** contains the character; if *only* the special key was pressed, however, **key** contains a string such as **control**, **shift**, or **alt**.

For the events, get an overview of the information provided and a sense of when (and how often) the events are triggered by outputting the event objects to the console.

By the way, with **addEventListener()** you could attach several event handlers for the same event to the same object. With the method **removeEventListener()**, which has the same parameters as **addEventListener()**, you can then "disconnect" an event handler again.

■ Other Events

JavaScript knows numerous other events in addition to those mentioned here. Not only HTML elements can be carriers of events. The document (standard object **document**) and the browser window (standard object **window**) also have events; for example, the **resize** event is triggered when the browser window is resized, and the **beforeunload** event is triggered when the web page is exited (navigation to another URL). If you want to know which events an object supports, enter the object identifier followed by **.on** in the JavaScript console, and the pop-up list of object properties will take you directly to the available events (whose property names all start with **on**).

34.4 Summary

In this chapter, we looked at how to branch in program flow depending on conditions and events.

Be sure to take the following points from this chapter:

- If-Else constructs have the general form **if(condition) { codeIf } else { codeOtherwise }**, where the **else** branch is optional.
- The condition is an expression that can be evaluated as **true** or **false**; the special values **null**, **undefined** and **NaN** are always considered **false**, strings are always considered **true**; the same is **true** for numbers except **0**.
- A condition can be composed of several sub-conditions that are linked with the logical operators **&&** (logical *AND*), **||** (logical *OR*) and **!** (logical *NOT*) operators.
- In addition to the usual numeric comparison operators **>**, **>=**, **<** and **<=**, the operators **==** and **!=** (unequal) are used to formulate conditions.
- The condition operator **?** allows efficient formulations of branches in the form **condition ? returnIf : returnOtherwise** and is useful when a value is to be decided on the basis of a condition.
- Switch-Case constructs of the form **switch(expression) { case expression1: statement1; ...; break; ... case expressionN: statementN; ... ; break; default: statementDefault; ... ; break; }** are well suited when an expression (for example, the value of a variable) is to be tested for many similar conditions. A deeply nested and therefore confusing If-Else construct can be avoided in this way. The **default** branch is optional.
- Event handlers are functions that are automatically called whenever an event occurs.
- Event handlers can be set directly in the HTML code of the Web page for an HTML element with a property assignment of the form **onevent = "eventHandler()**", or they can be installed dynamically in the JavaScript code with the **addEventListener(eventName, eventHandlerObject)** method of an HTML element object.
- The most important events are mouse and keyboard events, of which there are different types, which are triggered by certain constellations of actions (such as one double-click = two clicks) or partial actions (such as releasing a mouse button or pressing down a keyboard key).
- Event handlers are automatically passed an event object whose properties describe the event in more detail (for example, exactly which key was pressed on the keyboard, or at which position on the screen a mouse click occurred); even if your event handler does not provide an argument at all, you can always access the default **event** argument in its code.

34.5 Solutions to the Exercises

■ Exercise 34.1

A simple way to show that strings always evaluate to **true** is to run a condition like the following in the JavaScript console:

```
> if('A string') console.log('Is true')
Is true
```

Interestingly, a condition of the form **if('A string' == true)** would not have resulted in an output. This is because the string itself does not have the value **true**, of course (it is just a string and has the value stored in the string). However, if it actually has *to be* evaluated as **true** or **false**, because a *logical expression* is expected at the corresponding position, it is always considered **true**.

■ Exercise 34.2

One solution might look like this:

```
if(sex == 'm') salutation = 'Mr'
else salutation = 'Ms';

console.log('Hello', salutation, name);
```

Because only one statement follows in each of the code blocks of the **if** and **else** branch, the curly braces can be omitted here. The same applies to the previous tasks, but note, however, the omission of the braces leads to a more confusing code.

■ Exercise 34.3

One solution might look like this:

```
function convert() {
  var temp = Number(document.getElementById('temp').value);
  var direction = document.getElementsByName('direction');

  if(direction[0].checked == true) {
    if(temp >= 0) {
      document.write('<p>${temp} Kelvin are  
${temp - 273.15} degrees Celsius.<p>');
    }
    else {
      alert('The Kelvin temperature needs to be greater  
than or equal to zero.');
```

```
else {  
    if(temp >= -273.15) {  
        document.write('<p>${temp} degrees Celsius are  
        ${temp + 273.15} Kelvin.<p>');  
    }  
    else {  
        alert('The Celsius temperature must be greater  
        or equal to 273.15.');    }  
}  
}
```



Loops: How Do I Repeat Program Instructions Efficiently?

Contents

- 35.1 Counting Loops (for and for-of) – 528**
 - 35.1.1 for-Loops with Numerical Run Variables – 528
 - 35.1.2 for-Loop with Object Run Variable (for...of) – 534
- 35.2 Conditional Loops (while and do-while) – 536**
- 35.3 Summary – 537**
- 35.4 Solutions to the Exercises – 538**

Supplementary Information The online version contains supplementary material available at https://doi.org/10.1007/978-3-658-42912-6_35.

© The Author(s), under exclusive license to Springer Fachmedien Wiesbaden GmbH, part of Springer Nature 2024

J. L. Zuckarelli, *Learn coding with Python and JavaScript*, https://doi.org/10.1007/978-3-658-42912-6_35

Overview

Like most other programming languages, JavaScript provides looping constructs that allow similar statements to be repeated. In practice, such loops are popular and frequently used, as they allow for a clear and elegant formulation of repetitions. In this chapter, we will take a closer look at the main types of loops in JavaScript and see them in action in a larger example.

In this chapter you will learn:

- how to use a counting **for** loop with a numeric run variable to execute a statement or block of statements for a specified number of repetitions
- how to easily loop through a set of objects (such as the contents of an array) with a **for-of** loop
- how to develop head-controlled (**while**) and foot-controlled (**do-while**) conditional loops whose execution depends on a freely definable run condition

35.1 Counting Loops (for and for-of)

35.1.1 for-Loops with Numerical Run Variables

■ Structure and Function of the for Loop

The normal **for** loop to repeat a block of code comes in the following form in JavaScript:

```
for(initialization; check; increment) {
    // Code block that is repeated
}
```

Suppose we have an array of names that we want to loop through with a **for** loop:

```
> friends = ['Peter', 'Sophie', 'Helen', 'Mike', 'Ben']
```

35

As usual with counting loops, we need a (numerical) run variable, let's call it **i**, which is first initialized with a start value. Since, as we know by now, indexing of arrays in JavaScript starts at 0, it is recommended to set the run variable **i** to this value at the beginning: **i = 0**. Next, we need to set a check condition in the loop header that must be met for the loop to continue. In our example, where we want to loop through the array from 0 to the last element, this condition must be: **i <= friends.length-1** (note that the last element has index **length-1**, because we are already starting the count at 0!). Finally, we have to tell the loop how much the run variable should be incremented in each step. In our example, the run variable is incremented by one. Thus, a loop that runs through the elements of the array and outputs them to the console would look like this:


```
for(i = 0; i <= friends.length-1; i = i+1) {  
  console.log('Friend no. ', i+1, ': ', friends[i])  
}
```

The loop now starts with a value of 0 for the run variable and first checks whether the run condition is met. Because 0 is smaller than the length of the array reduced by one (namely four), the loop begins to execute the code in the loop body. It runs through this code in the first pass with 0 as the value of the run variable. The actual code executed is thus:

```
console.log('Friend no. ', 1, ': ', friends[0])
```

After the end of the loop body is reached, the execution jumps back to the loop header and increments the run variable according to the increment statement, in our example by one. Then the run condition is checked again and, if it is fulfilled, the code in the loop body is executed one more time. The loop continues these rounds until the run condition is no longer met after the next increment of the run variable. In our example if **i** were to be equal to 5 the run variable is now larger than the length of the array reduced by one, and the code block in the loop body is not executed again. Instead, the execution of the program continues after the **for** loop. The run variable retains its old value and is not incremented again.

Instead of using the increment instruction **i = i+1**, we can use the increment operator **++** (which is unary because it only works with one operand) and simply write **i++**. The values of the run variables can also be decremented (so we could have run through the array from back to front). Then, of course, the start value to which the run variable is initialized must also be adjusted, otherwise the loop would not run at all. The best way to decrease the run variable by 1, is to use the decrement operator **--**.

By the way: All loops in JavaScript can be exited with the **break** statement and sent to the next run with the **continue** statement. This is true not only for the **for** loop, but also **for-of**, **while**, and **do-while** loops, which we will look at in the following sections.

■ A Practical Example

The following example shows a **for** loop, more precisely two nested loops, in practical use. The goal this time is to develop a simple spreadsheet that can be used to enter numbers in the cells of a spreadsheet and then calculate the row and column totals. In the first step, the user enters the size of the spreadsheet.

The spreadsheet that is to be generated according to the user specifications can be seen in ■ Fig. 35.1.

Spreadsheet Document

				0
3	4			7
	5			5
				0
				0
3	9	0	0	

Calculate

■ Fig. 35.1 Spreadsheet with 5 rows and 4 columns

To enter the sheet size, we provide the user with the following simple web interface:

```
<!DOCTYPE html>
<html>

  <head>
    <title>Spreadsheet</title>
    <noscript>Please activate JavaScript!</noscript>
  </head>

  <body>
    <script src="spreadsheet.js"></script>

    <h1>Set Table Size</h1>
    <form>
      Rows:<br>
      <input id="rows" type="text" value="0"><p></p>
      Columns:<br>
      <input id="columns" type="text" value="0"><p></p>
      <input type="button" value="Create table"
        onclick="createTable()" >
    </form>
  </body>

</html>
```

When you click the “Create Table” button, the JavaScript function **createTable()** from the **spreadsheet.js** file is called. This function looks like this:

```
1  function createTable() {
2    var num_rows = Number(document.getElementById('rows').
   value);
3    var num_columns =
   Number(document.getElementById('columns').value);
4    var i,f;
```

35.1 • Counting Loops (for and for-of)

```

5
6     document.write('<H1>Spreadsheet document</H1>');
7     document.write('<form><table>');
8
9     // Write cells
10    for(i = 1; i <= num_rows; i++) {
11        document.write('<tr>');
12        for(f = 1; f <= num_columns; f++) {
13            document.write('<td><input id="R', i, 'C', f, '" \
14                           type="text" value=""></td>');
15        }
16        // Add cell for sum column
17        document.write('<td><input id="SUM_R', i, '"
18                       type="text" \
19                           value="" readonly="true" style="background-color: \
20                           #d1d1d1; "></td>');
21        document.write('</tr>');
22    }
23    // Add sum row
24    document.write('<tr>');
25    for(f = 1; f <= num_columns; f++) {
26        document.write('<td><input id="SUM_C', f, '"
27                       type="text" \
28                           value="" readonly="true" style="background-color: \
29                           #d1d1d1;"></td>');
30    }
31    document.write('</tr>');
32    document.write('</table>');
33
34    document.write('<input type="hidden" id="nrows" \
35                   value="', num_rows, '">');
36    document.write('<input type="hidden" id="ncolumns" \
37                   value="', num_columns, '">');
38
39    document.write("&<p></p>")
40    document.write('<input type="button" value="Calculate" \
41                   onclick="calculate()">');
42    document.write("&</form>");
43 }

```

This JavaScript code creates the table for our spreadsheet. Simple tables have the following form in HTML:

```
<table>
  <tr><td>Row 1, column 1</td><td>Row 1, column 2</td></tr>
  <tr><td>Row 2, column 1</td><td>Row 2, column 2</td></tr>
</table>
```

The individual table rows are represented by **tr** elements (*table row*), the cells they contain by **td** elements (*table data*).

The outer **for** loop with run variable **i**, whose loop head can be found in line 10, creates the rows of the table. In the body of this loop is another **for** loop, this one with the loop header in line 12. This “inner loop” with run variable **f**, writes data for the current row, (i.e., the row indicated by run variable **i** in the “outer” **for** loop), one cell *for each column*. In this way, the two nested **for** loops completely “run through” the rectangular table schema.

In lines 16/17 (this code is *not* in the inner loop!) another cell is written for the current line as part of a “Totals” column. Similarly, outside the two loops (lines 24–28), another loop is used to write a “Totals” row.

Note that we give our value cells IDs of the form **R_xC_y**, where **x** is the row and **y** is the column in which the cell is located. The summary rows or columns have IDs of the form **SUM_R_x** or **SUM_C_y**. This type of systematic ID composition will allow us to easily access the individual cells.

The form elements in lines 29/30 also help us here: They are of type **hidden** and are ultimately nothing more than a hidden repository for information. We store the number of rows and columns here to be able to access them later when summing.

The summation is handled by the **calculate()** function, which the user can trigger via the button we create in line 33. Alternatively, you can attach the function as an event handler to the **change** or **input** events of the individual cell input fields; to do this, you only need to supply the **onchange** or **oninput** property of the respective input element with a reference to the **calculate()** functions in line 13 (try it out!)

The code of the **calculate()** function looks like this:

```
1  function calculate() {
2    var num_rows =
      Number(document.getElementById('nrows').value);
3    var num_columns =
      Number(document.getElementById('ncolumns').value);
4    var i, f, sum, sum_column;
5
6    // Calculate row sums
7    for(i = 1; i <= num_rows; i++) {
8      sum_cell = document.getElementById('SUM_R' + i);
9      sum = 0;
10     for(f = 1; f <= num_columns; f++) {
11       sum = sum +
```

35.1 • Counting Loops (for and for-of)

```

        Number(document.getElementById('R' + i + 'C'
            + f).value);
12    }
13    sum_cell.value = sum;
14 }
15
16 // Calculate column sums
17 for(f = 1; f <= num_columns; f++) {
18     sum_cell = document.getElementById('SUM_C' + f);
19     sum = 0;
20     for(i = 1; i <= num_rows; i++) {
21         sum = sum +
            Number(document.getElementById('R' + i + 'C'
                + f).value);
22     }
23     sum_cell.value = sum;
24 }
25 }

```

calculate() first queries the row and column counts from our two **hidden** form elements (lines 2 and 3). Then we calculate the row totals (lines 7–14) and column totals (lines 17–24). We take advantage of the fact that the IDs of the value cells have the form **RxCy** and the cells of the sum rows and columns have IDs of the form **SUM_Rx** and **SUM_Cy**, respectively.

In the case of row totals, for example, we go through all rows with the help of a **for** loop (line 7) and first select the respective cell of the “Totals” column (line 8). Afterwards, we only have to go through the individual value columns (row 10), add the contained numbers (line 11) and write the sum into the corresponding sum cell of this table row (row 13).

? 35.1 [10 min]

What do two **for** loops look like that output our array **friends** to the console, where

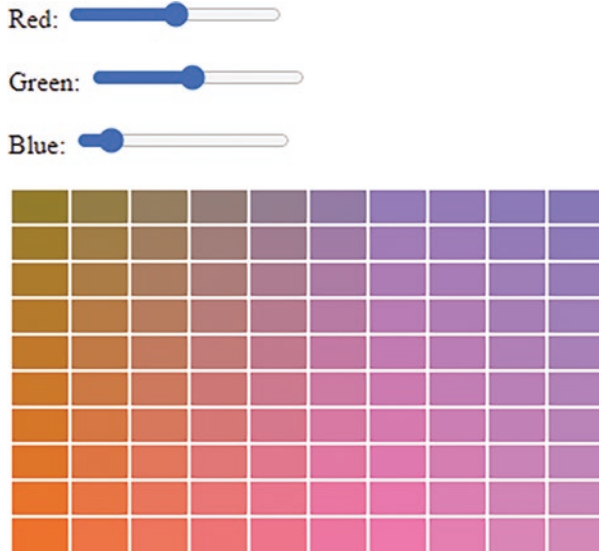
- a loop displays only every second entry
- the other loop displays each entry, but proceeds from back to front?

? 35.2 [60 min]

Develop an application in which the user can first specify a color based on its red, green and blue components using sliders. Based on this color, a shading table is then to be expanded in such a way that each of the table cells shows a different color. In the horizontal the red and in the vertical the blue parts are to be increased in a total of 10 steps up to 255 (the maximum according to the RGB scheme). The green portion remains fixed at the value set by the user.

The shading table should update automatically when the user moves one of the sliders.

Your finished application might then look like  Fig. 35.2.
Some tips:



■ Fig. 35.2 Shading table in the exercise application

- Use the **toString()** method of the **Number** object to convert a decimal number to a hexadecimal number, which you need to display as an RGB color value in HTML-standard format **#RRGGBB**. As argument, **toString()** takes the base of the number system to convert to, which is **16** for hexadecimal numbers.
- Remember that with the format **#RRGGBB** there must always be two digits per color part! Numbers smaller than 16 lead to *one-digit* hexadecimal numbers. These must then be preceded by a 0.
- The color components you calculate must be integers. To be on the safe side, round them with the **Math.floor(number)** function. This function returns the next smaller integer to the number passed as argument.

35.1.2 for-Loop with Object Run Variable (for...of)

35

The second form of **for-loops** known to JavaScript does not count up or down a numeric run variable but runs through a set of objects; the content of the run variable is then the object to which the respective loop pass applies.

With this, the example from the previous section can then be written like this:

```
> friends = ['Peter', 'Sophie', 'Helen', 'Mike', 'Mohamed']
> i = 0;
```

```
for(myfriend of friends) {  
  i++;  
  console.log('Friend no. ', i, ':', myfriend)  
}
```

We need to increment the variable **i** ourselves here. It is not the run variable of the loop (that is **myFriend**) but serves us here only to generate a consecutive number for our console output.

This will give you the following output:

```
Friend no. 1 : Peter  
Friend no. 2 : Sophie  
Friend no. 3 : Hellen  
Friend no. 4 : Mike  
Friend no. 5 : Fatih
```

Thus, the general form of the **for-of** loop is:

```
for(runVariable of iterableObject) {  
  // Code block that is repeated  
}
```

The *iterable object* we are iterating through here is an array. Other objects can also be iterable; for example, an object representing an address could be made so that its properties (such as the street name, house number, and zip code) are iterable and could therefore be traversed using a **for-of** loop. What this requires, however, is beyond entry-level JavaScript and thus beyond our scope here.

By the way, strings, which can also be accessed in array notation (► Sect. 31.4), are also iterable objects and can be traversed with **for-of** loops:

```
for(character of 'Hello World!') {  
  console.log(character);  
}
```

Note that when passing through the **for-of** loop, the run variable is not simply a *copy* of the element of our iterable object to which the current loop pass applies. It is, in effect, *the element itself*. Changes you make to the run variable in its **for-of** loop therefore affect the object being iterated through!

35.2 Conditional Loops (while and do-while)

With the **while** as well as the **do-while** loop, JavaScript has two conditional loops. The **while** loop is *head-controlled*, i.e., the condition is checked at the beginning of the loop and the loop is therefore not run at all if the condition is not fulfilled right from the start. **do-while** is a *foot-controlled* loop. It runs at least once in any case; at the end of the first run. The condition is checked to determine whether the loop should run a second time, and for each subsequent loop.

The two loops generally have the following structure:

```
while(condition) {
    // Code block that is repeated
}
```

and

```
do {
    // Code block that is repeated
}
while(condition)
```

The output of our **friends** array from the previous section would be achieved with a **while** loop, for example:

```
> friends = ['Peter', 'Sophie', 'Helen', 'Mike', 'Mohamed']
> i = 0;
  while(i <= friends.length - 1) {
    console.log('Friend no. ', i+1, ': ', friends[i]);
    i = i + 1;
  }
```

A formulation with **do-while** might look like this instead:

```
i = 0;
do {
    console.log('Friend no. ', i+1, ': ', friends[i]);
    i = i + 1;
}
while(i <= friends.length - 1)
```

do-while loops should only be used if you can safely assume that the loop's run condition will be satisfied on the first pass; here, that's not a problem because we know we don't have an empty array in front of us.

In our example, we have used **while** and **do-while** loops to solve a task for which one would normally use a **for** loop, since the number of runs can easily be determined with the help of the **length** property of the array. Conditional loops are normally used when the execution depends on a general condition and the number of passes cannot necessarily be determined in advance. However, our example nicely illustrates a principle that we encountered earlier: Any problem that can be solved with a counting loop can also be solved with a conditional loop, because when in doubt, you can also check as a condition the value of a numeric run variable that you manually increment, just as we did in our example. In this sense, the *counting* loop is a special form of the *conditional loop*, namely one whose condition checks a run variable, which the counting loop kindly also takes care of initializing and incrementing.

? 35.3 [10 min]

Write two more versions of the **while** loop from this section that outputs our array **friends** to the console, where

- Once the “run variable” **i** is initialized with 1 (instead of 0)
- The less-than operator is used in the condition (instead of the less-than-equal operator).

? 35.4 [30 min]

Develop a function **countCells()** that optionally counts the rows or the columns of the table that our spreadsheet application creates from ► Sect. 35.1.1. Use a **while** loop to step through the IDs of the cells. Take advantage of the fact that the **document.getElementById()** function returns **null** if an element with the specified ID could not be found.

The function you develop here could be used by our **calculate()** function to get the number of rows and columns, if we didn’t “cache” it as information using **hidden** elements directly in the page source of the HTML document.

35.3 Summary

In this chapter, we saw how to repeat code in JavaScript using counting (**for**, **for-of**) and conditional (**while**, **do-while**) loops.

Be sure to take the following points from this chapter:

- In JavaScript, counting **for loops** have the form **for(initialization; check; increment) { statements }**. Here, a numeric run variable initialized to a value at the beginning is changed before each loop pass according to an increment statement and checked whether the new value satisfies a check condition. If this is the case, the following block of statements is run through, but if this is not the case, program execution continues after the loop and the run variable retains its value from before the last increment.
- The run variable can also be decreased, i.e., the run variable becomes smaller with each loop pass.

- Increases and decreases by one can be implemented with the increment operator **++** and the decrement operator **--** in the form **variable++** or **variable--**.
- A **for-of** loop of the form **for(runVariable of iterableObject) { statements }** can be used to iterate through objects whose elements are iterable, i.e. they can be put into any kind of order by JavaScript, such as arrays. In this case, the run variable is not a numeric value, but the element of the iterable object to which the current loop pass applies. Changes to the run variable affect the iterable object whose elements are being iterated through.
- Conditional loops can be either head-driven in the form **while(condition) { statements }** or footer-driven with **do { statements } while(condition)**. Loops of the latter type are run at least once because the condition is not checked until the end.

35.4 Solutions to the Exercises

■ Exercise 35.1

A loop that displays only every second entry from the array might look like this:

```
for(i = 0; i <= friends.length-1; i=i+2) {
    console.log('Friend no. ', i+1, ':', friends[i]);
}
```

A loop that proceeds from back to front when displaying the array entries might look like this:

```
for(i = friends.length-1; i >= 0; i--) {
    console.log('Friend no. ', i+1, ':', friends[i]);
}
```

■ Exercise 35.2

The interface of our shading table application could be designed in HTML code like this:

```
<!DOCTYPE html>
<html>

    <head>
        <title>Color Table</title>
        <noscript>Please activate JavaScript!</noscript>
    </head>
```

```

<body>

  <script src="colortable.js"></script>

  <form>
    <p>Red: <input id="red" type="range" min=0
      max=255 value="0" onchange="colorsNew()"><p>
    <p>Green: <input id="green" type="range" min=0
      max=255 value="0" onchange="colorsNew()"><p>
    <p>Blue: <input id="blue" type="range" min=0
      max=255 value="0" onchange="colorsNew()"><p>
  </form>

  <table id="colortable">

  </table>

</body>

</html>

```

For this purpose, the JavaScript file **colortable.js**:

```

function colorsNew() {
  var colorRed = Number(document.getElementById('red').value);
  var colorGreen = Number(document.getElementById('green').
    value);
  var colorBlue = Number(document.getElementById('blue').
    value);
  var tab = document.getElementById('colortable');
  var i,f, colorValue, stepRed, stepBlue;

  stepRed = Math.floor((255-colorRed)/10);
  stepBlue = Math.floor((255-colorBlue)/10);
  tableHTML = "";

  for(i = 1; i <= 10; i++){
    tableHTML = tableHTML + '<tr>';

    for(f = 1; f <= 10; f++) {
      colorRedNew = (colorRed + i*stepRed).toString(16);
      colorGreenNew = colorGreen.toString(16);
      colorBlueNew = (colorBlue + f*stepBlue).toString(16);

```

```

        if(colorRedNew.length == 1)
            colorRedNew = '0' + colorRedNew;
        if(colorGreenNew.length == 1)
            colorGreenNew = '0' + colorGreenNew;
        if(colorBlueNew.length == 1)
            colorBlueNew = '0' + colorBlueNew;

        colorVal = "#" + colorRedNew + colorGreenNew + color-
            BlueNew;
        tableHTML = tableHTML + '<td style="background-color: '
            +
                colorValue + '; color: ' + colorValue + '">xxxx</
            td>';
    }
    tableHTML = tableHTML + '</tr>';
}
tab.innerHTML = tableHTML;
}

```

■ Exercise 35.3

- (a) A **while** loop that goes through the array and starts at 1: $i = 1$;

```

while(i <= friends.length) {
    console.log('Friend no. ', i, ':', friends[i-1]);
    i = i + 1;
}

```

- (b) A **while** loop that traverses the array and uses the less operator in the run condition:

```

i = 0;
while(i < friends.length) {
    console.log('Friend no. ', i+1, ':', friends[i]);
    i = i + 1;
}

```

■ Exercise 35.4

The function **countCells()** could look like this:

```
function countCells(columnRow = 'column') {  
    var num = 0;  
    var id;  
  
    do {  
        num = num + 1;  
        if(columnRow == 'column') {  
            id = 'R1C' + num;  
        }  
        else {  
            id = 'R' + num + 'C1';  
        }  
    }  
    while(document.getElementById(id) != null)  
  
    return num - 1;  
}
```

We work here with an argument **columnRow**, which has the default value "**column**" and specifies whether the number of columns or rows is to be returned. The number is then determined by composing cell IDs in a **do-while** loop (we assume here that at least one column or row exists, so that we can also test the run condition of the loop at the end) and attempting to select these cells with **getElementById()**. If this fails because the cell does not exist, **getElementById()** returns **null** and the loop ends. The number of rows/columns **num** incremented in the loop is then the return value of the function.



Debugging & Error-Handling: How Do I Search for and Fix Errors in a Structured Way?

Contents

- 36.1 Error Handling at Runtime – 544
- 36.2 Troubleshooting During Development – 546
- 36.3 Summary – 550

Overview

To wrap up our introduction to JavaScript, we turn to the unloved but important topic of debugging and error handling. Overall, JavaScript is a bit more robust than other programming languages when it comes to runtime errors. So, where your program would abort with an error message in other languages, JavaScript just keeps running. However, this only makes the handling of runtime errors *seem* easier; special, unusual situations must still be anticipated and handled by you as a programmer.

For debugging purposes throughout the development process, the developer tools available in modern browsers include several useful features that you should take full advantage of.

In this chapter you will learn:

- how JavaScript handles runtime errors and what this means for you as a programmer
- how you can catch exceptions
- how to use browser debugging tools to find bugs in your programs, especially how to set breakpoints, watch variables, and run your program in single step mode

36.1 Error Handling at Runtime

Runtime errors where the program terminates are rarer in JavaScript than in many other programming languages. The reason is that JavaScript handles some situations that would have thrown an exception in the other languages differently and seeks a less dramatic resolution. For example, if you take the square root of a negative number with **Math.sqrt(-1)**, the return value is simply **NaN**—*not a number*. If you divide a number by 0, you get **Infinity** (or **-Infinity** if the dividend was negative). If you want to convert a string that is not a number into a number (for example with **Number("abc")**), this conversion again results in **NaN**. In all these cases, many other programming languages would have failed the service and thrown an exception. Not so in JavaScript. JavaScript stubbornly continues to run and only signals through the results of the performed operations that something didn't quite go as planned.

At first glance, this way of dealing with errors is a good thing for you as a programmer, as it reduces the risk of your program going completely off the rails with some strange exception. However, the fact that no exceptions are thrown does not, of course, mean that your program will do what it is supposed to; after all, whether your JavaScript code will still produce useful results after the user converts a supposed numeric input that was not actually a number at all and therefore produces a **NaN** value is questionable for now. As a developer, you must therefore ensure, through suitable checks, that your program can really cope with all conceivable constellations—even and especially if these constellations do *not* lead to exceptions.

However, exceptions can of course also occur in JavaScript. For example, if you call the **toLowerCase()** method (that converts a string to lowercase) for a numeric variable, you will get an exception, just as you will if you try to access an HTML element in a script (such as our spreadsheet example from the previous chapter) with an ID that is not actually assigned to any element of the web page. However, errors of this kind are often not caused by circumstances that only occur at runtime but can be detected and corrected during development—at least with sufficient testing. In the fight against such errors, the debugging features of the developer tools in the browser are particularly important supporters. We will deal with them in the next section.

Even though exceptions are less important in JavaScript than in many other languages, JavaScript also supports a Try-Catch construct. It has the following syntax and thus, if you remember our considerations from ► Sect. 16.2, has a very “classic” structure:

```
try {  
    // Code that is "tried"  
}  
catch(err) {  
    // Code for error handling  
}  
finally {  
    // Code that is always executed in any case  
}
```

catch takes as its argument an error object from which you can extract a variety of information, most notably the error name (in the **name** property) and an error message (in the **message** property). If you are particularly interested in which line of your code the exception was thrown, you can query the **line** property of the **err** object (or whatever you want to call the argument of **catch()** in your script).

By the way, you can also create exceptions yourself with the help of the **throw** statement. For example, you could use an exception to test whether the dividend is 0 when dividing two numbers:

```
var a = 10, b = 0;  
  
try {  
    if(b == 0) throw new  
        Error('Division by 0 is impossible!');  
}  
catch(myError) {  
    console.log('An error occurred: ', myError.message);  
}
```


Of course, if you don't catch your own exception, you'll get a program abort with a suitably dramatic error message in the browser's JavaScript console, as befits a proper exception!

36.2 Troubleshooting During Development

For debugging during development, modern browsers regularly provide some tools that we already learned about in ► Sect. 29.2.1. These include breakpoints, variable watching, and step-by-step execution.

Even though we regularly refer to the situation in Google *Chrome* in this section you will find the same or very similar tools in virtually all other modern browsers as well. Even the functionality and way of working are often extremely similar.

To make the following considerations a little more vivid, consider the following example of a web page in which the user can simply enter a number and click a button:

```
<!DOCTYPE html>

<html>

  <head>
    <title>Script with errors</title>
    <noscript>Please activate JavaScript!</noscript>
  </head>

  <body>
    <script src="witherror.js"></script>
    <form>
      <p>Number: <input type="text" id="number"></p>
      <p><span id="result"></span>
      <p></p>
      <input type="button" value="Calculate"
onclick="calculate()" ">
    </form>
  </body>

</html>
```

For the output of a calculation result we have created a **span** element with the ID **result**. In the web page we included the script **witherror.js**, which also contains the **calculate()** function triggered when the button is clicked. It looks like this:

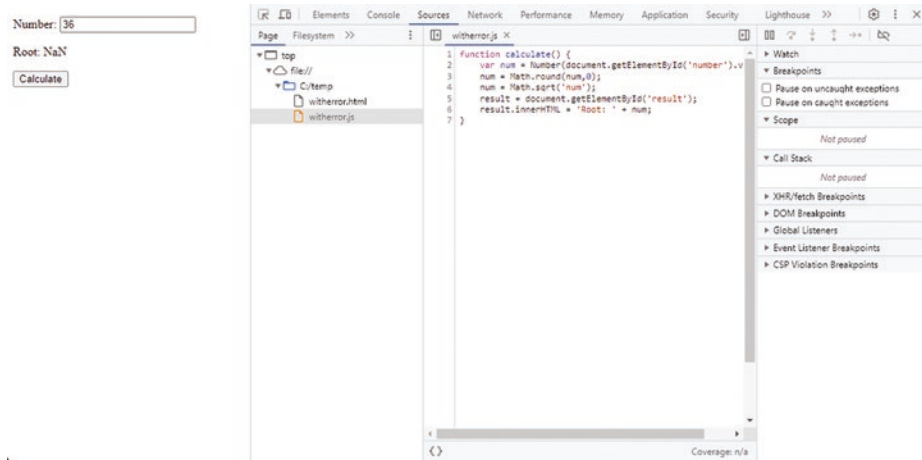
```
function calculate() {  
    var num =\  
        Number(document.getElementById('number').value);  
    num = Math.round(num);  
    num = Math.sqrt('num');  
    var result = document.getElementById('result');  
    result.innerHTML = 'Root: ' + num;  
}
```

As you can see, we do nothing here but read in the number entered by the user, round it to an integer, and take the square root from it. We output the result on our **span** element. So far, so good.

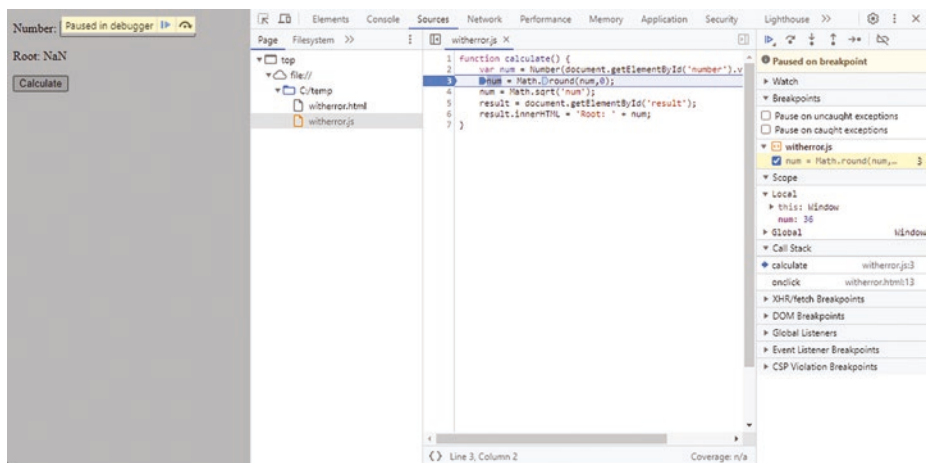
If you now open the web page in the browser, enter a number and click the button, you get the output **NaN** displayed on the **span** element **result** of the web page. So, the result of the calculation is apparently not a number. To get to the bottom of the problem, you could of course either study the program code, and perhaps you have already noticed the error (which is, of course, completely intentional); or you could simply have the variable **num**, which we work with throughout the script, displayed once after each operation by including appropriate output statements like **console.log(num)** in the program code. Of course, either way will get you there, even if you didn't have debugging tools available in the browser. Temporarily inserting output statements into program code is probably the most popular debugging method of all because it is both simple and effective. However, especially if the section of code where you suspect the error is very long, you may need a lot of debugging output, or you may need to move an output incrementally farther and farther through the section of code until you find the interesting place where your problem actually is. This is tedious. It is in such cases that it is worth using the debugging tools provided by the browser, and that is what we will use now.

If you click on the “Sources” tab in the developer tools and then select your JavaScript file, you will see its contents in the middle of the developer tools area. You can see this in ■ Fig. 36.1.

In the right part of the developer tools area, you will see several expandable and collapsible sub-areas, including the “Breakpoints” subarea. If you click on the line number in front of the code display, a *breakpoint* is set at this point. If you now execute the JavaScript code, in our example by clicking on the button on the web page, the JavaScript code is processed until the breakpoint is reached. Then a situation as shown in ■ Fig. 36.2 appears. The execution of the program has now stopped before line 3 is executed. In the “Scope” sub-area, you can see variables present in the current scope and their values, in our case the variable **num** with a value of 36. So, obviously everything is still fine here. We could move breakpoint further below in the code and let the program execute down to this next breakpoint. ■ Figure 36.3 shows the state of the program at another breakpoint, namely in line 6. Here we see from the “Scope” subarea on the right that **num** has meanwhile assumed the value **NaN**.



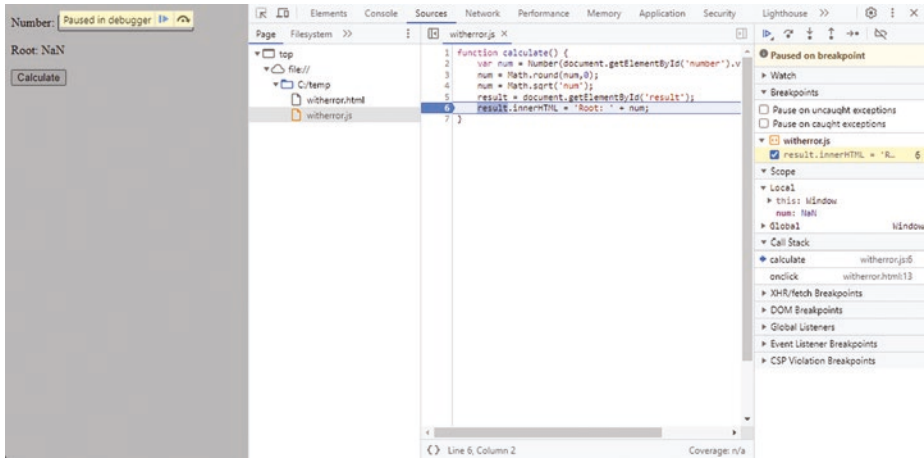
■ Fig. 36.1 JavaScript debugging area in Google Chrome



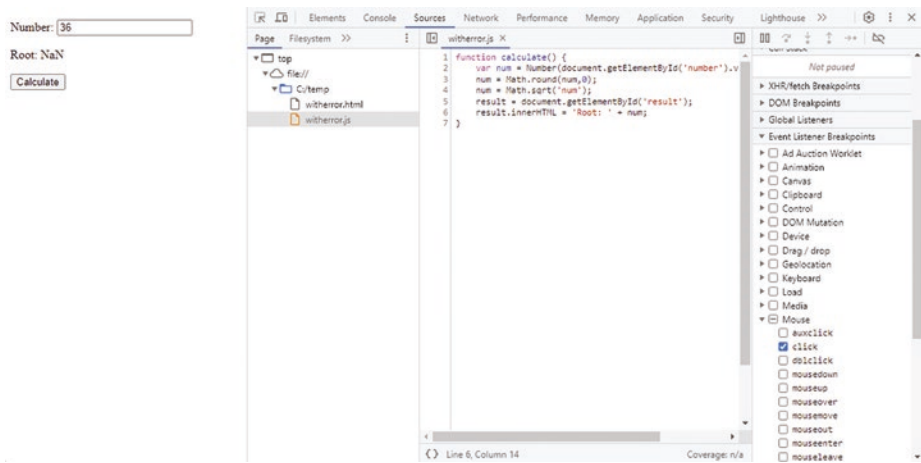
■ Fig. 36.2 Reached breakpoint in Google Chrome

So, something must have happened in lines 3 and 4. Of course, a closer look immediately reveals the source of the error: In line 4, the function **Math.sqrt()** is not passed the variable **num**, but a string **'num'** as argument (did you see it before?). Obviously, a square root cannot be calculated from a string; but because JavaScript is relatively fault-tolerant, as discussed earlier, **Math.sqrt()** simply returns **NaN** instead of throwing an exception.

Instead of moving breakpoints, you also have executed the program stepwise from the first breakpoint. Each time you <F9> (“Step”) the next program statement is executed. This way, you would have easily found that the problem comes from line 5.



■ Fig. 36.3 Reaching a breakpoint, after the error (in line 4)



■ Fig. 36.4 Setting an event breakpoint in Google Chrome

An interesting debugger feature is the possibility to add *event listeners*. This is done in the corresponding sub-area that you can see in ■ Fig. 36.4. Here you have the option of setting an event breakpoint, i.e., a breakpoint that is triggered whenever a certain event occurs, for example a mouse click. In an event-driven program flow, this variant of the breakpoint is a useful alternative to a breakpoint that attaches to a particular line, especially if you have many event handlers that respond to events of a particular type.

Breakpoints can therefore be very useful for diagnosing errors. If you want to continue the program from a breakpoint, click on the Play button in the toolbar above the right part of the debugging area.

In the first subsection on the right, the “Watch” subsection, you can set up *variable watches*, that is, expressions entered whose value you want to monitor and take a closer look at when program execution has come to a stop at a breakpoint. The expressions you enter here do not have to be simple variables, as in our example where we are only watching the value of the variable **num**, but you can also enter more complex expressions such as **Math.sqrt(num)>2.8** (a Boolean expression).

36.3 Summary

At the end of the JavaScript part, we looked at error diagnosis and handling.

Be sure to take the following points from this chapter:

- JavaScript is relatively robust against runtime errors, so it rarely terminates with an error message. However, this does not release the programmer from the obligation to ensure that the program does what it is supposed to do, even in exceptional situations.
- The Try-Catch construct also provides a way to catch exceptions in JavaScript; the **catch()** statement, when an exception occurs, is automatically passed an error object that you can evaluate to learn more about the exception.
- The developer tools of practically all modern browsers provide several useful debugging features for diagnosing errors during development; these debugging features include, in particular, functions for working with breakpoints, for observing the contents of variables, and for executing the program in single-step mode. In addition, event listeners provide the possibility of linking breakpoints to the occurrence of an event as such.
- Probably the most frequently used “debugging tool” in practice is the **console.log()** function. Output generated in this way can help to track down the causes of errors in many situations; compared to them, the debugging tools of the developer tools show their strengths especially (but not only) when there are still few clues as to where the source of the error might lie.

Supplementary Information

Index – 553

Index

A

Airbnb JavaScript Style Guide, 416
 Algorithm, 6, 7, 10, 11, 17, 22
 Allen, Paul, 15
 Analytical Engine, 5, 26
 AND (logical operator), 163–166
 AngularJS (JavaScript framework), 149, 509
 Application Programming Interfaces (APIs), 149–150
 Argument (function), 138
 – default value, 140
 – keyword argument, 141
 – optional, 140
 – passing by reference, 142
 – passing by value, 142
 – positional argument, 141
 Argument (function) (JavaScript), 500–504
 – default value, 502
 – function as argument, 502
 – optional, 502
 – passing as value, 500
 – passing by reference, 500
 Argument (function) (Python), 329–334
 – default value, 331
 – indefinite number of arguments, 331
 – keyword, 330
 – optional, 331
 – positional, 330
 – type hint annotation, 333
 Array, 92–95
 – element, 93
 – index, 93
 – multi-dimensional, 93
 – strings as arrays, 95
 Array (JavaScript), 433–440
 – creating, 433
 – length, 435
 – merging, 437
 – modifying, 436
 – selecting elements, 433, 434
 – sorting, 438
 Artificial intelligence (AI), 8, 10
 – ethical considerations, 10
 – expert system, 11
 – general/strong, 9
 – image recognition, 10
 – in judgment and decision-making, 9
 – machine learning (*See* Machine learning)

– social debate, 9
 – training of models, 11
 – weak, 8
 Auto-completion, 54, 244
 Automatic syntax check, 54

B

Babbage, Charles, 5, 19, 26
 Batch mode (of code execution), 68
 Berners-Lee, Tim, 404
 Boole, George, 86, 241
 Borland, 55
 Byron, Augusta Ada, 5, 19, 33

C

Cascading Style Sheets (CSS), 406
 ChatGPT, 9, 62, 77, 147, 224
 Class, 99
 – access rights, 107–108
 – attribute/property, 98, 99
 – constructor, 105
 – definition, 99
 – hierarchy, 101
 – inheritance, 100, 105
 – instance of, 99
 – method (*See* Method)
 Class (JavaScript) *See* Object (JavaScript)
 Class (Python), 244, 264–268, 339
 – attribute, 244–248
 – constructor, 251, 265
 – hierarchy, 341
 – inheritance, 266–268
 – instance, 244, 246
 – method (*See* Method (Python))
 – name mangling, 268
 Claude, 62, 77
 Code block, 73, 139, 157, 160
 Code block (JavaScript), 494
 Code block (Python), 226, 227
 Code editor, 53
 Code editors (specific), 53
 – Atom, 53
 – Notepad++, 53, 401
 – Sublime Text, 53, 400
 – Vim, 53
 – Visual Studio, 401
 – Visual Studio Code, 53

Code execution, 66

- batch/script mode (*See* Batch mode (of code execution))
- interactive mode (*See* Interactive mode (of code execution))

Comment, 43, 72, 76

Comment (JavaScript), 416

- inline comment, 416
- multiline, 416
- single-line, 416

Comment (Python), 230–231

- documenting open tasks, 231
- inline, 230
- in PyCharm, 231

Comparison operator, 161

Compiler, 20, 24, 26, 52, 54, 67

- bytecode compiler, 25
- just-in-time compiler (JIT compiler), 25

Conditional loop, 185–189

- examples, 186
- relation to counting loops, 188
- run condition, 185
- types of, 185

Conditional loop (while, do-while) (JavaScript), 536–537

- condition, 536
- foot-controlled, 536
- head-controlled, 536

Conditional loop (while) (Python), 372–374

Condition operator (JavaScript), 518

Console application, 115, 124–126

Console (JavaScript), 409, 454

Console (Python), 209

Constant, 92

Control (user interface), 116

- button, 118
- checkbox, 118
- combo box, 122
- edit field, 117
- list view/list box, 120
- picker, 120
- radio button, 118
- slider, 119
- toggle button, 119
- tree view, 120

Conversion (of variables), 88–89

- explicit, 88
- implicit, 88–89

Counting loop, 179–185

- examples, 182
- exiting prematurely, 184
- nested loops, 183
- relation to conditional loops, 188
- running through a field, 180
- run variable, 179, 180

Counting loop (for) (JavaScript), 528–534

- changing object run variable in loop, 535
- example, 529
- numerical run variable, 528
- object as run variable, 534–535

Counting loop (for) (Python), 366–370

- examples, 368
- list comprehension expression (*See* List comprehension expression)
- nested, 370–371
- run variable, 366

CPython, 208

D

Dark mode, 53

Data type, 83–89

- basic, 84
- boolean, 86
- character, 85
- conversion (*See* Conversion (of variables))
- date, 87
- enumeration, 87
- factor, 88
- floating-point number, 84
- integer, 84
- logical, 86
- precision, 84
- range, 83
- set, 88
- string, 85
- time, 87

Data type (JavaScript), 421–430

- array (*See* Array (JavaScript))
- boolean (logical value), 427–428
- elementary data type, 421–430
- infinity, 422
- NaN, 429
- null, 428
- number (whole and decimal numbers), 421–423
- string (*See* String (JavaScript))
- undefined, 428

Data type (Python), 238–244

- associative field (*See* Dictionary (Python))
- basic data type, 238–244
- bool (logical value), 241–242
- complex data type, 251–264
- dictionary, 259
- float (decimal number), 239
- int (whole number), 239
- list (*See* List (Python))
- None/NoneType, 242–243
- ordered data type, 251, 258
- set, 262

Index

- str (text) (*See* String (Python))
- tuple, 258 (*See Also* Tuple (Python))
- unordered data type, 260, 262

Debugger *See* Debugging

Debugging, 55, 194

- algorithmic error, 194
- breakpoint, 198
- debugger, 197
- error at development time, 194
- errors at runtime, 195–196
- exception (*See* Exception)
- methods, 197–199
- single step mode, 198
- syntax error, 194
- try-catch construct (*See* Try-catch construct)
- variable watches, 198

Debugging and error handling (JavaScript)

- debugging tools in the browser, 546
- event listener, 549
- handling exceptions (*See* Try-catch construct (JavaScript))
- working with breakpoints, 547
- working with variable watches, 550

Debugging and error handling (Python), 382–386

- executing code step-by-step, 389–390
- handling exceptions (*See* Try-except construct (Python))
- working with breakpoints, 386–388
- working with variable watches, 388–389

Dictionary, 96

Dictionary (Python), 259–262

- appending elements, 261
- creating, 260
- deleting elements, 261
- extracting keys/values, 261
- key, 260
- nested, 260
- selecting elements, 260
- value, 260

Distribution *See* Library

Django (framework), 204

Django (Python framework), 149

Documentation (of code), 78

- documentation generators, 78
- doxygen, 78
- Javadoc, 78, 79
- phpDocumentor, 78
- roxygen, 78

Documentation (of code) (Python), 232–234

- autodoc, 233
- docstring, 232
- doxygen, 233
- function annotations, 233, 333
- pydoc, 233

- pydoctor, 233

E

Eich, Brendan, 396

Embarcadero, 52

Escape sequence, 85

- in Python, 278

Event, 116, 123, 168–172

- event handler, 169

Event (JavaScript), 520–523

- assigning event handler via on-event-property, 521
 - assigning event handler with addEventListener(), 522
 - event listener (*See* Debugging and error handling (JavaScript))
 - event object, 521
 - keyboard events, 523
 - mouse events, 522
 - on-event property of input elements, 520
- Event-oriented programming (programming paradigm), 169

Event (Python, tkinter), 305–308

- callback function, 306
- event handler, 306, 361
- keyboard event, 307
- mouse event, 307

Exception, 195

Exception (JavaScript) *See* Try-catch construct (JavaScript)

Exception (Python) *See* Try-except construct (Python)

F

Facebook, 23

Features (of code editors and IDEs), 54

- auto-completion (*See* Auto-completion)
- automatic syntax check (*See* Automatic syntax check)
- debugging, 54, 55
- design of graphical user interfaces, 54
- syntax highlighting (*See* Syntax highlighting)

Field (variable)

- associative (*See* Dictionary)
- ordered (*See* Array)

File (for input and output), 126

- access mode, 127
- binary, 128
- end of file (EOF), 128
- file pointer, 128
- file variable/object, 129
- text, 128

File (for input and output)

- (Python), 315
- access mode, 315
- append mode, 315
- closing file, 317
- file object, 315
- file pointer, 315
- opening a file, 315
- read mode, 316
- reading from a file, 316
- write mode, 315
- writing to a file, 316

File system access (Python), 369

Flow control, 154, 155

- condition, 155, 160–166
- event (*See* Event)
- if-else construct (*See* If-else construct)
- switch-case construct (*See* Switch-case construct)

Flow control (JavaScript)

- composite condition, 517
- condition, 517
- event (*See* Event (JavaScript))
- if-else construct (*See* If-else construct (JavaScript))
- switch-case construct (*See* Switch-case construct (JavaScript))

Flow control (Python), 354–356

- composite condition, 358–359
- condition, 355
- event (*See* Event (Python, tkinter))

Framework, 148–149

- difference to libraries, 148
- inversion of control, 149
- popular examples, 149

Framework (JavaScript), 509

Function, 138–145

- argument (*See* Argument)
- body, 139
- definition, 139, 140, 143
- head, 139
- as object, 144
- in object-oriented programming (*See* Method)
- return value, 138
- side effect, 142

Function (JavaScript), 494–506

- anonymous, 498
- argument (*See* Argument (JavaScript))
- calling a function with arguments, 501
- defining, 494
- as objects, 495
- return value, 499–500
- scope of variables (*See* Variable (JavaScript))

Function (Python), 328–339

- argument (*See* Argument (function) (Python))
- body, 329
- defining, 328–329
- function head, 329
- return values, 334–335
- scope of variables (*See* Variable (Python))
- side effect, 337

G

Garbage collection (Python), 238

Gates, Bill, 15

Gemini, 62, 77

Generative artificial intelligence, 62

Geometry manager (Python, tkinter), 299–305

- grid manager, 300, 302–305
- pack manager, 299–302
- place manager, 300, 305
- role and function, 299

Git, 146

GitHub, 146

Google, 23, 55

Google JavaScript Style Guide, 416

Graphical user interface (GUI), 54, 115–124

- control (*See* Control (user interface))
- development, 122–124
- menu, 118

H

Hamilton, Margaret, 9, 20

Hash *See* Dictionary

Hello world program, 68

- in JavaScript, 409–412
- in Python, 281

Help, 60

- function reference, 60
- language reference, 60
- Stack Overflow (*See* Stack Overflow)
- tutorial, 60

Help (JavaScript), 401

- documentation by Mozilla Foundation, 401
- language specification, 401
- Stack Overflow, 401

Help (Python), 211

- language reference, 212
- library reference, 212
- Stack Overflow, 211

Hohmann, Joachim, 27

Hopper, Grace, 20, 27

Houston, Drew, 15

Hypertext Markup Language (HTML), 404

Index

- attribute, 405
- Document Object Model (DOM), 405, 460–461
- forms (*See* Input with forms (JavaScript))
- structure of HTML documents, 405
- tag, 405
- World Wide Web Consortium (W3C), 404

I

- Identifier, 74
- IDLE (Python IDE), 209, 223
- If-else construct, 155–161
 - nested, 166
 - simple, 155
- If-else construct (JavaScript), 516–519
- If-else construct (Python), 354–361
 - alternative conditions (elif), 359–361
 - nested, 356–357
 - simple, 354
- Indexing (JavaScript), 433
- Indexing (Python), 252
- Input and output, 114–115
 - forms of, 115
 - via databases, 131–134
 - via files (*See* File)
 - via graphical user interfaces (*See* Graphical user interface (GUI))
 - via the console (*See* Console application)
- Input and output (JavaScript), 453–454
 - via a web page (*See* Output to web page (JavaScript))
 - via dialog boxes, 457–458
 - via the console, 454–457
 - *See Also* Input with forms (JavaScript)
- Input and output (Python), 278–280
 - via files, 315–317
 - via graphical user interface, 280–315
 - via the console, 278–280
- Input with forms (JavaScript), 470–476
 - accessing forms from JavaScript, 472–476
 - example, 476–488
 - input element, 471
 - password entry, 471
 - read-only input element, 472
 - required input element, 472
 - sending data to server, 472
 - setting up an HTML form, 470–472
 - submit button, 471
 - types of input elements, 471
- Integrated Development Environment (IDE), 54
- Integrated Development Environments (specific), 55
 - Android Studio, 55

- AppCode, 58
- Aptana Studio, 58
- C++ Builder, 58
- CLion, 58
- Eclipse, 58
- IntelliJ IDEA, 58
- JBuilder, 58
- Komodo IDE, 58
- NetBeans, 55, 58, 400
- Padre, 58
- PhpStorm, 58
- PyCharm, 55, 58
- QT Creator, 58
- RAD Studio, 55
- Rcommander, 58
- Rodeo, 58
- RStudio, 55, 58
- RubyMine, 58
- SharpDevelop, 58
- Spyder, 58
- Thonny, 58
- Visual Studio, 55, 58
- Visual Studio Tools for Office (VSTO), 58
- WebStorm, 58, 400
- Xcode, 58
- Zend Studio, 58
- Interactive mode (of code execution), 67
- Interactive mode (Python), 209, 221–222
- Interpreter, 24, 26, 52, 54, 67
- Interpreter (JavaScript), 400

J

- Jacquard, Joseph-Marie, 5
- Jacquard loom, 5
- JavaScript, 396
 - ActionScript, 396
 - areas of application, 396
 - community, 508
 - ECMAScript, 396
 - embedding in web pages, 404–409
 - history, 396
 - integrating code into web page by embedding the source code, 408
 - integrating code into web page by linking to a code file, 407
 - JScript, 397
 - LiveScript, 396
 - safety aspects of code execution, 408–409
 - standardization, 396
 - TypeScript, 396
 - visibility of the source code, 409
- JavaScript Object Notation (JSON), 445–446
- JavaScript Standard Style Guide, 416
- JetBrains, 55

jQuery (JavaScript library), 509
 JS Bin (JavaScript web service), 401, 411, 412
 JS.do (JavaScript web service), 401, 412
 Jupyter, 223

K

Kay, Alan, 98
 Keras (Python library), 204

L

Large Language Models (LLMs), 62
 Library, 145–148
 – central platform, 146
 – finding, 146
 – importing into program, 148
 – installing, 148
 Library (JavaScript) *See* Module (JavaScript)
 Line break (Python), 278
 Line indentation, 73
 List comprehension expression, 371–372
 List (Python), 251–258
 – appending elements, 254
 – changing list elements, 254
 – colon operator, 253
 – deleting elements, 255
 – determine length, 256
 – joining lists, 256
 – selecting elements, 252–253
 – sorting, 255
 – sublists, 256–258
 Llama, 62
 Logical operator, 163–166
 Logical programming (programming paradigm), 32
 Loop (JavaScript), 528–535
 – conditional loop (while) (*See* Conditional loop (while, do-while) (JavaScript))
 – continuing with next run, 529
 – counting loop (for) (*See* Counting loop (for) (JavaScript))
 – exiting prematurely, 529
 Loops, 178–179
 – conditional loops (*See* Conditional loop)
 – counting loops, 179
 – foot-controlled, 185
 – foot-controlled loop, 186
 – head-controlled, 185, 186
 – types of, 179
 Loops (Python), 366–372
 – conditional loop (while) (*See* Conditional loop (while) (Python))
 – continuing with next run, 376

 – counting loop (for) (*See* Counting loop (for) (Python))
 – exiting, 375
 – head-controlled, 372

M

Machine language, 20, 24, 26, 52
 Machine learning, 11
 Macro, 14
 Map *See* Dictionary
 Menabrea, Federico Luigi, 5
 Method, 103–105, 144
 Method (Python), 244, 339
 – constructor, 341 (*See Also* Class (Python))
 – default constructor, 341
 – overloading, 340
 – standard method for output, 341
 Microsoft, 55
 Microsoft Office, 23
 Microsoft Windows, 23
 Mnemonic, 26
 Module (JavaScript), 506–509
 – searching for, 508–509
 – sourcing from GitHub, 508
 – sourcing from javascripting.com, 509
 Module (Python), 342–348
 – importing selected classes, 344
 – importing the entire contents of a module, 344
 – importing the entire module, 345
 – searching for, 345–348
 Modules *See* Library
 Monty Python, 204

N

NaN (JavaScript), 517, 544
 – *See Also* Data type (JavaScript)
 NASA, 9, 23
 Neural network, 10
 Node.js (JavaScript framework), 397
 NOT (logical operator), 163–166
 null (JavaScript), 517
 – *See Also* Data type (JavaScript)
 NumPy (Python library), 204, 243

O

Object (JavaScript), 441
 – as associative array, 443
 – constructor, 441
 – constructor function, 444–445
 – creating by calling constructor, 444

Index

- creating by direct assignment, 442
 - creating by using constructor function, 444–445
 - inheritance, 441
 - in JSON format (*See* JavaScript Object Notation (JSON))
 - method, 441, 443
 - object constructor, 444
 - property, 441, 443
 - prototype, 441
 - static property, 441
 - Object-oriented programming (OOP) (JavaScript), 441
 - Object-oriented programming (OOP) (programming paradigm), 32, 97, 144
 - attribute, 32
 - method, 32
 - Object-oriented programming (OOP) (Python), 244, 264–268, 339–342
 - Object (Python) *See* Class (Python)
 - OpenAI, 9, 62
 - Operator (JavaScript), 422
 - comparison operator, 517
 - condition operator (*See* If-else construct (JavaScript))
 - decrement, 422
 - increment, 422
 - increment operator, 529
 - modulo, 422
 - Operator (Python), 355
 - assignment operator, 355
 - comparison operator, 355
 - logical operator, 355, 358, 517
 - OR (logical operator), 163–166
 - os (Python module), 371
 - Output to web page (JavaScript), 458–460
 - adding an HTML element, 468
 - changing an attribute of an HTML element, 467
 - deleting an HTML element, 470
 - document hierarchy, 463–465
 - injecting HTML code, 466
 - querying attributes of HTML elements, 464
 - selecting an HTML element by CSS class, 463
 - selecting an HTML element by ID, 461
 - selecting an HTML element by type, 462
 - selecting children of an HTML element, 463
 - types of DOM nodes, 464, 465
 - writing to an HTML document, 458
- P**
- Package *See* Library
 - Package (Python), 342–348
 - installing packages with pip, 346
 - installing packages with PyCharm, 346
 - Python Package Index (PyPI) (*See* Python Package Index (PyPI))
 - Parameter (JavaScript) *See* Argument (JavaScript)
 - pip, 346
 - Plunker (JavaScript web service), 401, 412
 - Polymorphism, 105
 - Primitive (JavaScript) *See* Data type (JavaScript)
 - Private (access right) *See* Class
 - Procedure *See* Function
 - Programming, 14
 - additive potential, 18
 - hype, 19
 - labor market, 16
 - nerds in, 18
 - paradigm, 30
 - roadmap for learning, 63–64
 - science and art, 43
 - tips for learning, 42
 - training of logical thinking, 18
 - women in, 19
 - Programming language, vii, 22, 30
 - areas of application, 30
 - C/C++, 129
 - compiler (*See* Compiler)
 - declarative, 30
 - Delphi/Object Pascal, 122
 - dialect, 30
 - difference to a natural language, 22
 - ecosystem, 146
 - executing code, 66
 - explicitly typed, 90
 - first program, 66
 - general-purpose, 31, 204
 - high-level language, 26
 - interpreter (*See* Interpreter)
 - machine language (*See* Machine language)
 - object-oriented (*See* Object-oriented programming (OOP) (programming paradigm))
 - open-source, 37, 52
 - paradigm, 31, 38
 - Pascal, 125, 130
 - popularity, 37
 - roadmap for learning, 69
 - roadmap to learning, 80, 109, 134, 150, 172–173, 189, 199
 - source code, 53
 - special-purpose, 31
 - strongly typed, 89
 - syntax (*See* Syntax)
 - variety, 30

Programming languages (specific), 14

- ActionScript (*See* JavaScript)
- Ada, 20, 33
- Algorithmic Language (ALGOL), 27
- Assembler, 26
- BASIC, 33
- C#, 32, 33, 58
- Cascading Style Sheets (CSS), 30
- C/C++, 31–33, 39, 58, 68, 76, 77, 86, 90, 95, 101, 138, 158, 160, 168
- Common Business-Oriented Language (COBOL), 20, 27, 33
- Delphi, 33, 36, 52, 69
- Delphi/Object Pascal, 92, 102
- ECMAScript (*See* JavaScript)
- Eiffel, 33
- F#, 127
- Formula Translation (FORTRAN), 27, 33
- Gödel, 33
- Hypertext Markup Language (HTML), 30
- Java, 25, 31, 36, 39, 58, 78, 396
- JavaScript, 36, 39, 58
- JScript (*See* JavaScript)
- LiveScript (*See* JavaScript)
- Objective-C, 36
- Object Pascal, 33
- Pascal, 130, 143, 144, 148, 186
- Perl, 58, 86, 96, 146
- PHP, 30, 36, 58, 69, 78, 82, 127, 143, 146, 149, 182, 183
- Plankalkül, 27
- Prolog, 32, 33
- Python, 31, 33, 36, 39, 58, 67
- R, 30, 36, 58, 67, 78, 82, 86, 127, 146, 148, 223
- Ruby, 58, 149
- Scala, 223
- Smalltalk, 98
- Structured Query Language (SQL), 30, 133
- Swift, 30, 36, 58
- Tcl, 280
- TypeScript (*See* JavaScript)
- VisualBasic, 31, 36
- Visual Basic for Applications (VBA), 14, 33, 58, 90, 95, 159, 160, 165, 183, 188
- Visual C/C++, 36
- Wolfram Language, 33

Protected (access right) *See* Class

Prototype (JavaScript) *See* Object (JavaScript)

Public (access right) *See* Class

Punch card, 5

PyCharm, 209, 210, 222, 227

- community version, 210
- debugging tools, 386
- installation, 210

pytest (Python library), 386

Python, 204

- benevolent dictator for life, 204
- help (*See* Help (Python))
- installation, 208
- interpreter (*See* CPython)
- origins, 204
- popularity, 204

Python Enhancement Proposal (PEP), 227, 233

Python Package Index (PyPI), 345–348

Python Software Foundation, 204, 208, 212

R

React (JavaScript framework), 509

Read-eval-print-loop (REPL) *See* Interactive mode (of code execution)

Reference counter (Python), 238

Reserved word, 53

Roadmap

- controlling the program flow based on conditions and events, 172
- executing code and writing first program, 69
- inputting and outputting data via graphical or console user interfaces, files and databases, 134
- tools and help, 64
- working with functions and libraries, 150
- working with variables and classes/objects, 109
- writing easy-to-understand code, 80

Roadmap (for learning a programming language), 63

- finding and fixing errors, 199
- repeating program statements efficiently with loops, 189

ROT13 encryption (Python), 368

S

script (HTML tag), 407

Script mode (Python), 209

Set (Python), 262–264

- checking for equality, 263
- checking for subsets, 264
- creating, 263
- intersecting, 263
- merging, 264

Stack Overflow, 37, 38, 147, 204

Star Trek—The Next Generation, 9

- Commander Data, 9
- Klingon language, 22

Streamlit (Python library), 280

Strict mode (JavaScript), 420

Index

String (JavaScript), 423–427

- assigning, 423
- concatenating, 425
- escaping, 424
- methods and properties, 426
- splitting into an array, 440
- working with individual characters, 425, 440

String (Python), 239–241

- docstring (*See* Documentation (of code) (Python))
- as list of characters, 258
- method of str object, 247
- over multiple lines, 240

String substitution (JavaScript), 456

Switch-case construct, 155, 166–168

Switch-case construct (JavaScript), 519–520

Syntax, 22

Syntax (JavaScript), 414–416

- case-sensitivity, 414
- code style, 416
- comment (*See* Comment (JavaScript))
- ending statement, 414
- naming conventions, 414–416
- statements over multiple lines, 414

Syntax (Python), 226–230

- case sensitivity, 229–230
- code block (*See* Code block (Python))
- comment (*See* Comment (Python))
- ending statements, 228–229
- indentation, 226–228
- line breaks, 228
- “pythonic” identifiers, 229–230
- statements over multiple lines, 228–229
- style guide, 227

Syntax highlighting, 53, 54, 401

T

Template literal (JavaScript), 455

TensorFlow (Python library), 204

Testing, 196–197

TIOBE index, 37, 39, 204

Tk, 280

tkinter (Python), 343

tkinter (Python library), 280–315

- event (*See* Event (Python, tkinter))
- example application, 308–315
- exercise, 317–318
- geometry manager, 299
- help and documentation, 284
- main application window, 282
- widget (*See* Widget (Python, tkinter))

Tools, 52

- code editor (*See* Code editor)

- community edition, 52, 55
- compiler (*See* Compiler)
- Integrated Development Environment (IDE) (*See* Integrated Development Environment (IDE))
- interpreter (*See* Interpreter)
- linker, 67
- open-source, 55

Torvalds, Linus, 146, 508

Try-catch construct, 195, 196

Try-catch construct (JavaScript), 545

Try-except construct (Python), 383–386

Tuple (Python), 258–259

Turing, Alan, 9

Turing test, 9

U

undefined (JavaScript), 517

- *See Also* Data type (JavaScript)

unittest (Python library), 386

UNIX time, 87

User experience (UX) designer, 115

User interface (UI) designer, 115

V

van Rossum, Guido, 204, 208

Variable, 74, 82–83

- declaration, 90
- global, 142
- identifier, 82 (*See Also* Identifier)
- initialization, 91
- local, 142

Variable (JavaScript), 420–421

- converting implicitly, 430–433
- declaring, 420
- global, 504
- iterable object, 535
- local, 504
- scope, 504–506

Variable (Python), 236–238

- converting, 249–251
- converting explicitly, 251
- converting implicitly, 250
- creating, 236–238, 248–249
- data type, 237
- deleting, 238
- global, 335–339
- iterable object, 366
- local, 335–339
- as object, 244–249
- scope, 335
- size in memory, 239

Virtual environment (Python), 347
 Vue (JavaScript framework), 509

W

Web scraping, 37
 Widget (Python, tkinter), 282–299

- arrangement of controls (*See* Geometry manager (Python, tkinter))
- button, 282–284
- canvas, 299
- check button, 292–294
- file dialog, 298–299
- input field, 289–291
- listbox, 294–297
- menu, 286–288
- messagebox, 297–298

- notebook (tabs), 299
- progressbar, 299
- radio button, 292–294
- spinbox, 299
- standard options, 284
- text output/label, 291–292
- tree view, 299

Wolfram, Stephen, 33

X

XOR (logical operator), 163

Z

Zuse, Konrad, 24, 27