Programming with Python

Thomas Weise (汤卫思)

November 5, 2026



Abstract

The goal of this book is to teach practical programming with the Python language to high school, undergraduate, and graduate students alike. Hopefully, readers without prior knowledge can follow the text. Therefore, all concepts are introduced using examples and discussed comprehensively. All examples are available online in the GitHub repository associated with this book, so that readers can play with them easily. Actually, the goal of the book is not just to teach programming, but to teach programming as a part of the software development process. This means that from the very beginning, we will attempt to push the reader towards writing clean code with comments and documentation as well as to use various tools for finding potential issues. While this book is a work in progress, we hope that it will eventually teach all the elements of Python software creation. We hope that it can enable readers without prior programming experience to develop beautiful and maintainable software.

Contents

Contents i

1 Introduction 1
- 1.1 Why Programming *and* Software Engineering Tools? . . . 1
- 1.2 Why Python? . . . 3
- 1.3 Further Reading . . . 4

I Basics 7

2 Getting Started 9
- 2.1 Installing Python . . . 9
- 2.2 Installing PyCharm . . . 11
- 2.3 Our First Program . . . 16
- 2.4 Python in the Terminal . . . 20
- 2.5 Getting the Examples from this Book . . . 25
- 2.6 Summary . . . 29

3 Simple Datatypes and Operations 30
- 3.1 Introduction . . . 30
- 3.2 Integers . . . 30
- 3.3 Floating Point Numbers . . . 37
- 3.4 Interlude: The Python Documentation and other Information Resources . . . 51
- 3.5 Boolean Values . . . 63
- 3.6 Text Strings . . . 68
- 3.7 None . . . 81
- 3.8 Summary . . . 83

4 Variables 84
- 4.1 Defining and Assigning Variables . . . 84
- 4.2 Interlude: Finding Errors in your Code with the IDE and Exceptions . . . 93
- 4.3 Multiple Assignments and Value Swapping . . . 99
- 4.4 Variable Types and Type Hints . . . 100
- 4.5 Object Equality and Identity . . . 108
- 4.6 Summary . . . 111

5 Collections 112
- 5.1 Lists . . . 112
- 5.2 Interlude: The Linter Ruff . . . 117
- 5.3 Tuples . . . 120
- 5.4 Sets . . . 124
- 5.5 Dictionaries . . . 128
- 5.6 Summary . . . 129

II Control Flow Statements 131

6 Conditional Statements 133

6.1 The `if` Statement . . . 133
6.2 The `if...else` Statement . . . 134
6.3 The `if...elif...else` Statement . . . 137
6.4 The Inline Ternary `if...else` Statement . . . 140
6.5 Summary . . . 142

7 Loops **143**
7.1 The `for` Loop Statement . . . 143
7.2 The `continue` and `break` Statements . . . 146
7.3 Nesting Loops . . . 147
7.4 Loops over Sequences . . . 148
7.5 `enumerate` and Interlude: Pylint . . . 151
7.6 The `while` Loop . . . 155
7.7 The `else` Statement at the Bottom of Loops . . . 158
7.8 Summary . . . 160

8 Functions **161**
8.1 Defining and Calling Functions . . . 161
8.2 Functions in Modules . . . 166
8.3 Interlude: Unit Testing . . . 169
8.4 Function Arguments: Default Values, Passing them by Name, and Constructing them . . . 177
8.5 Functions as Parameters, `Callables`, and `lambdas` . . . 181
8.6 Summary . . . 185

9 Exceptions **186**
9.1 Introduction . . . 186
9.2 Raising Exceptions . . . 188
9.3 Built-in Exceptions . . . 193
9.4 Handling Exceptions . . . 195
9.5 Interlude: Testing for Exceptions . . . 209
9.6 Summary . . . 214

10 Iteration, Comprehension, and Generators **216**
10.1 `Iterables` and `Iterators` . . . 217
10.2 List Comprehension . . . 220
10.3 Interlude: doctests . . . 225
10.4 Set Comprehension . . . 227
10.5 Dictionary Comprehension . . . 229
10.6 Generator Expressions . . . 230
10.7 Generator Functions . . . 240
10.8 Operations on Iterators . . . 244
10.9 Summary . . . 247

III Classes **248**

11 Basics of Classes **249**
11.1 Design Principle: Immutable Classes . . . 251
11.2 Design Principle: Encapsulation . . . 258
11.3 Summary . . . 265

12 Inheritance **267**
12.1 A Hierarchy of Geometric Objects . . . 267
12.2 Summary . . . 276

13 Dunder Methods **278**
13.1 `__str__`, `__repr__`, and `__eq__` . . . 278
13.2 Objects in Sets and as Keys in Dictionaries: `__hash__`, `__eq__` . . . 285
13.3 Arithmetic Dunder and Ordering . . . 290
13.4 Interlude: Debugging . . . 299

13.5 Implementing Context Managers for the `with` Statement Revisited 316
13.6 Overview over Dunder Methods 323
13.7 Summary 326

IV Working with the Ecosystem **327**

14 Using Packages **329**
14.1 pip and Virtual Environments 329
14.2 Requirements Files 338
14.3 Virtual Environments in PyCharm 341

15 The Distributed Version Control System git **344**
15.1 Cloning git Repositories 344

Backmatter **354**

Best Practices **355**

Useful Tools **361**

Glossary **363**

Python Commands **383**

Bibliography **389**

Scripts **438**

Preface

This book tries to teach undergraduate and graduate students as well as high school students how to program with the Python programming language. It aims to strike a good balance between theory and practice, leaning more to the practice side. In particular, we try to teach programming together with some software engineering concepts. It is the firm opinion of the author that these two cannot be separated. Teaching programming alone without introducing tools such as static code analysis, unit tests, and enforcing principles such as code style and proper commenting will create bad programmers. So we discuss these aspects while working our way through the principles of programming.

This book is intended to be read on an electronic device. Please do not print it. Help preserving the environment.

This book is a work in progress. It will take years to be completed and I plan to keep improving and extending it for quite some time.

The book consists of two types of material: Materials that the author (Thomas Weise) has created by himself and such that have been created by others. The vast majority of the material is teaching material created by the author. This and only this material is released under the *Attribution-NonCommercial-ShareAlike 4.0 International license* (CC BY-NC-SA 4.0). However, the book also includes some images and figures created by others, such as logos and screenshots and more, which are marked explicitly and licensed under their authors' terms. For example, all logos and trademarks are under the copyright of their respective owners.

You can cite this book [615], e.g., by using the following BibTEX:

```
@book{programmingWithPython,
 author = {Thomas Weise},
 title = {Programming with Python},
 year = {2024--2026},
 publisher = {School of Artificial Intelligence and Big Data,
              Hefei University},
 address = {Hefei, Anhui, China},
 url = {https://thomasweise.github.io/programmingWithPython}
}
```

This book contains a lot of examples. You can find all of them in the repository https://github.com/thomasWeise/programmingWithPythonCode. You can clone this repository as discussed in Sections 2.5 and 15.1 and play with these example codes.

The text of the book itself is also available in the repository https://github.com/thomasWeise/programmingWithPython. There, you can also submit issues, such as change requests, suggestions, errors, typos, or you can inform me that something is unclear, so that I can improve the book. Such feedback is most welcome. The book is written using LATEX and this repository contains all the scripts, styles, graphics, and source files of the book (except the source files of the example programs).

Copyright © 2024–2026

Prof. Dr. Thomas Weise (汤卫思教授)
at the School of Artificial Intelligence and Big Data (人工智能与大数据学院)
of Hefei University (合肥大学),
in Hefei, Anhui, China (中国安徽省合肥市)

Contact me via email to tweise@hfuu.edu.cn with CC to tweise@ustc.edu.cn.



[book pdf]



[course website]

https://thomasweise.github.io/programmingWithPython

This book was built using the following software:

```
Alpine Linux 3.24.2
Linux 7.0.0-1017-azure x86_64

python: 3.14.10
pytest: 9.2.0
pytest-timeout: 2.4.0
mypy: 2.4.1
ruff: 0.17.3
pylint: 4.1.3

texgit_py: 0.9.13
texgit_tex: 0.9.7
pycommons: 0.8.108
pdflatex: pdfTeX 3.141592653-2.6-1.40.29 (TeX Live 2026)
biber: 2.22
makeglossaries: 5.1 (2026-04-21)
ghostscript: 10.07.1 (2026-05-19)

date: 2026-11-05 00:13:41 +0000
```

Acknowledgements

The author would like to thank Robert P. J. Day for his proofreading and numerous improvements to the book.

Chapter 1

Introduction

Welcome to *Programming with Python*, an introduction to Python programming. Our book is just one of many books that teach this programming language. You can find many more in Section 1.3. Our focus is on teaching good programming practices right from the start, supported by many examples and tools. And we will use *lots* of examples.

1.1 Why Programming *and* Software Engineering Tools?

What is programming? Programming means to delegate a task to a computer. We have this job to do, this thing. Maybe it is too complicated and time consuming to do. Maybe it is something that we have to do very often. Maybe it is something that we cannot, physically, do. Maybe we are just lazy. So we want that the computer does it for us. And it works just like delegating tasks in real life.

Our economy and society as a whole works because of the division of labor. In a factory or company, different works carry out different duties. If you are a chef in a kitchen, you have to tell the junior trainee chef: "First you wash the potatoes, then peel the potato skin, then you wash the potatoes again, and then you cook them." If you are visiting the hairdresser to get your hair done, you would say something like: "Wash my hair, then cut it down to 1cm on the top, trim the sides, then color it green."

In either case, you provide the other person with a clear and unambiguous sequence of instructions in a language they can understand. In this book, you will learn to do the same — with computers. And one language that the computers understand is Python.

Definition 1.1: Computer Program

A *computer program* is an unambiguous sequence of computational instructions for a computer to achieve a specific goal.

Definition 1.2: Programming

Programming is the activity or job of writing computer programs [460].

Now, in the vast majority of situations, we do not create a program to just use it one single time. Actually, this is similar to the real life situation of work delegation again: If you were a chef, you



Figure 1.1: The Python programming language logo is under the copyright of its owners.

basically "input" the "program" *cook potatoes* into the junior trainee once. In the future, you would like to be able go to them and invoke this program again by saying: "Please cook 2kg of potatoes."

Indeed, your "programs" often even have implicit parameters, like the quantity of 2kg mentioned above. Or, maybe, you go to the hairdresser again and want to say: "Same as usual, but today color it blue." In our day-to-day interactions, creating reusable and parameterized programs happens very often and very implicitly. We usually do not think about this in any explicit terms.

But if we write programs for computers, we *do* need to think about it in explicit terms and right from the start. The activity of pure programming listed in Definition 1.2 is only one part of software development. Imagine the following scenario:

In this book we discuss how to translate some kind of specification into `Python` code. Let's say that later in your job, you want to develop a program that can be used to solve a specific task. So, well, you write the program. You learned how to do that with this book. So you now have the file with the program code. The problem is solved.

Is it that easy? On the one hand, you may wonder whether you made any mistake. We are people, we all make mistakes. The more complex the task we tackle, the more (program code) we write, the more likely it is that we make some small error somewhere. So probably you want to *test* your program, i.e., check if it really computes the things that it should compute it the way you intended it to compute these things.

And what if your program is not just a single-use, stand-alone kind of program? What if it is part of some sort of software ecosystem? What if other programs may later need to use/run it? Maybe it can access some sort of sensor measuring something, or maybe it can convert data from one file format to another one. Then, other programmers may need to be able to at least understand how to correctly run the program and what kind of input and output data it will expect or produce, respectively. You must *document* your program clearly. If you write a package that offers functions that can be used by other programs, there must be annotations that clearly explain what kind of input and output datatypes these functions accept and produce, respectively.

What if this program is going to be needed for the next ten or so years? Maybe it could eventually become necessary to add new functionality? Maybe some of the software libraries you use get outdated and need to be replaced? There also are several situations in which a totally different person may need to work, read, modify, and improve *your* code. They need to be able to read and understand your code easily. You cannot just write stuff down quickly, you cannot just produce one big soup of unformatted code. You need to both have good documentation and write clean and easy-to-read code.

Or maybe you are a researcher, using `Python` 3 to implement some algorithm and to run an experiment. In order to make your experiments and results replicable, you would probably want to publish your algorithm implementation together with the results. This, agin, only makes sense if your code is at least a bit readable. The names of variables and functions must be clear and understandable. The coding style should be consistent throughout all files and it should follow common conventions [594].

All of these things need to be considered when we learn how to program. Because you do not just "program," you develop software. Indeed, a good share of programmers usually spend only about 50% of their time with programming [153, 373]. Other studies even suggest that less than 20% of the working time spent with coding, maybe with another 15% of bug fixing [389]. Of course, we cannot cover all other topics related to software engineering in a programming course.

However, in typical courses, such aspects are ignored entirely. In a typical course, you learn how to write a small program that solves a certain task. The task is usually simple, so the teachers are able to read your code even if you have ugliest style imaginable. Also, nobody is ever going to look at or use the programs you write as your homework again. In such a pristine scenario, none of the above really matters. But in reality, things are not always simple.

One may hope that, as one of the very first things, surgery students in medical school are taught to wash their hands before performing surgery. And you, our dear students, can expect us to teach you how to develop software properly. And we will try to do that. We will teach you how to write clean, well-documented, and properly tested programs. Right from the start. Such that all of your code will be reusable, readable, clear, clean, and *beautiful*.

As a result, this will be a course which is very practice centered. We will learn programming the *right* way.[1] The downside of this approach is that we will sometimes go off on tangents in the text.

[1] Well, at least, what *I* consider as the *right* way.

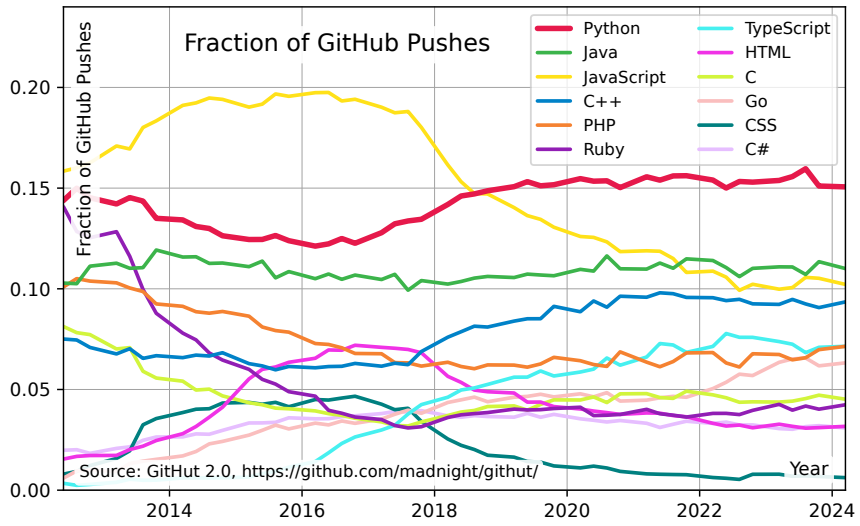


Figure 1.2: The twelve most popular programming languages chosen based on the GitHub pushes over the years. Source: [56].

While I am introducing variables in Chapter 4, for example, I will also explain how to use a static code analysis tool designed to find type errors in variable use. Also, the text will often have references to best practices that clarify common approaches and different code hygiene concepts. Our goal will be to learn how to do things right from the start and not put things off to later.

1.2 Why Python?

The center of this course is the `Python` programming language. Our goal is to get familiar with programming, with the programming language `Python`, and with the tools and ecosystem surrounding it. This makes sense for several reasons.

First, `Python` is one of the most successful and widely used programming languages [100]. We plot the number of pushes to GitHub over time for the most popular programming and web development languages in Figure 1.2. We find that `Python` became the leading languages at some point in 2018. In the TIOBE index, which counts the number of hits when searching for a programming language using major search engines, `Python` ranked one in January 2025 and was named the programming language of the year for 2024 [296].

> `Python` is everywhere nowadays, and it is the undisputed default language of choice in many fields.
>
> — *Paul Jansen [296], 2025*

If you will do programming in any future employment or research position, chances are that `Python` knowledge will be useful. According to the 2024 annual Stack Overflow survey [531], `Python` was the second most popular programming language, after JavaScript and HTML/CSS. In GitHub's Octoverse Report from October 2024 [220], `Python` is named the most popular programming language, ranking right before JavaScript.

Second, `Python` is intensely used [100] in the fields of Artificial Intelligence (AI) [487], Machine Learning (ML) [506], and Data Science (DS) [243, 386] as well as optimization, which are among the most important areas of future technology. Indeed, the aforementioned Octoverse report [220] states that the use in soft computing is one of the drivers of `Python`'s popularity.

Third, there exists a very large set of powerful libraries supporting both research and application development in these fields, including `NumPy` [152, 252, 299, 418], `Pandas` [43, 351, 432], `Scikit-learn` [438, 470], `SciPy` [299, 604], `TensorFlow` [1, 337], `PyTorch` [436, 470], `Matplotlib` [279, 281, 299, 429], `SimPy` [661], and `moptipy` [619][2], just to name a few. There are also many `Python` packages supporting other areas of computer science, that offer, e.g., connectivity to databases (DBs) [598], or support for

[2]Yes, I list `moptipy` here, next to very well-known and widely-used frameworks, because I am its developer.

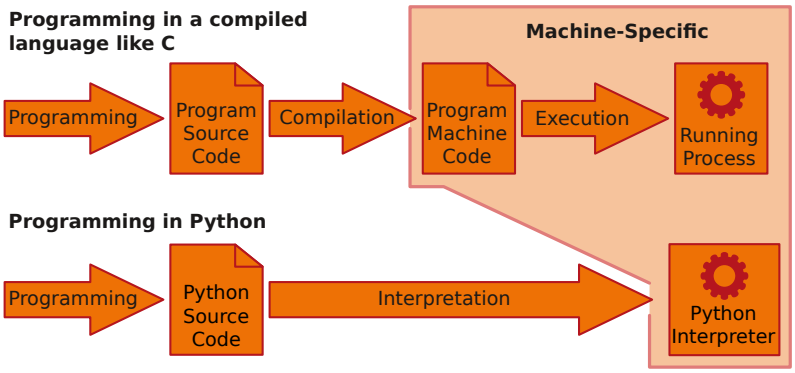


Figure 1.3: Python code is interpreted, which leads to an easier programming workflow compared to compiled programming languages like C.

web application development [4, 574]. This means that for many tasks, you can find suitable and efficient Python libraries that support your work.

Fourth and finally, Python is very easy to learn [239, 588]. It has a simple and clean syntax and enforces a readable structure of programs. Programmers do not need to declare datatypes explicitly[3]. Python has expressive built-in types likes lists, tuples, and dictionaries. Thus, Python was also named the language most popular for those who want to learn how to code in the aforementioned Stack Overflow survey [531]. The fact that Python is an interpreted language makes it somewhat slower compared to compiled languages like C. However, this also leads to a much easier workflow when experimenting and programming, as sketched in Figure 1.3. It also is possible to interactively write programs in an interpreter window. This means that you can execute commands in a terminal instead of needing to compile and run programs. These features, in sum, make Python a good choice for learning how to write programs.

1.3 Further Reading

The following books and online resources may help with your understanding of the Python programming language.

1.3.1 Books

1.3.1.1 Books on Python in General

- Mark Lutz. *Learning Python*. 6th ed. O'Reilly Media, Inc., Mar. 2025. ISBN: 978-1-0981-7130-8,
- Kevin Wilson. *Python Made Easy*. Packt Publishing Ltd, Aug. 2024. ISBN: 978-1-83664-615-0,
- Brett Slatkin. *Effective Python: 125 Specific Ways to Write Better Python*. 3rd ed. Addison-Wesley Professional, Nov. 2024. ISBN: 978-0-13-817239-8,
- John Hunt. *A Beginners Guide to Python 3 Programming*. 2nd ed. Springer, 2023. ISSN: 1863-7310. ISBN: 978-3-031-35121-1. doi:10.1007/978-3-031-35122-8,
- Eric Matthes. *Python Crash Course*. 3rd ed. No Starch Press, Jan. 2023. ISBN: 978-1-7185-0270-3,
- Paul Barry. *Head First Python*. 3rd ed. O'Reilly Media, Inc., Aug. 2023. ISBN: 978-1-4920-5129-9,
- Mike James. *Programmer's Python: Everything is an Object – Something Completely Different*. 2nd ed. I/O Press, June 25, 2022,
- Luciano Ramalho. *Fluent Python*. 2nd ed. O'Reilly Media, Inc., Apr. 2022. ISBN: 978-1-4920-5635-5,

[3] at least during the first steps of learning

- Bernd Klein. *Einführung in Python 3 – Für Ein- und Umsteiger*. 3., überarbeitete. Carl Hanser Verlag GmbH & Co. KG, 2018. ISBN: 978-3-446-45208-4. doi:10.3139/9783446453876,
- Kent D. Lee and Steve Hubbard. *Data Structures and Algorithms with Python*. Springer, 2015. ISSN: 1863-7310. ISBN: 978-3-319-13071-2. doi:10.1007/978-3-319-13072-9,
- David Ascher, ed. *Python Cookbook*. 1st ed. O'Reilly Media, Inc., July 2002. ISBN: 978-0-596-00167-4.

1.3.1.2 Books on Testing and Continuous Integration

- Brian Okken. *Python Testing with pytest*. Pragmatic Bookshelf by The Pragmatic Programmers, L.L.C., Feb. 2022. ISBN: 978-1-68050-860-4,
- Ashwin Pajankar. *Python Unit Test Automation: Automate, Organize, and Execute Unit Tests in Python*. Apress Media, LLC, Dec. 2021. ISBN: 978-1-4842-7854-3,
- Alfredo Deza and Noah Gift. *Testing In Python*. Pragmatic AI Labs, Feb. 2020. ISBN: 979-8-6169-6064-1,
- Moritz Lenz. *Python Continuous Integration and Delivery: A Concise Guide with Examples*. Apress Media, LLC, Dec. 2018. ISBN: 978-1-4842-4281-0,
- Kristian Rother. *Pro Python Best Practices: Debugging, Testing and Maintenance*. Apress Media, LLC, Mar. 2017. ISBN: 978-1-4842-2241-6.

1.3.1.3 Books on Data Science, Numerical Computation, Visualization, and AI

- Wes McKinney. *Python for Data Analysis*. 3rd ed. O'Reilly Media, Inc., Aug. 2022. ISBN: 978-1-0981-0403-0,
- Ashwin Pajankar. *Hands-on Matplotlib: Learn Plotting and Visualizations with Python 3*. Apress Media, LLC, Nov. 2021. ISBN: 978-1-4842-7410-1,
- Joel Grus. *Data Science from Scratch: First Principles with Python*. 2nd ed. O'Reilly Media, Inc., May 2019. ISBN: 978-1-4920-4113-9,
- Robert Johansson. *Numerical Python: Scientific Computing and Data Science Applications with NumPy, SciPy and Matplotlib*. Apress Media, LLC, Dec. 2018. ISBN: 978-1-4842-4246-9.

1.3.1.4 Other Topics

- Aaron Maxwell. *What are f-strings in Python and how can I use them?* Infinite Skills Inc, June 2017. ISBN: 978-1-4919-9486-3,
- Abhishek Ratan, Eric Chou, Pradeeban Kathiravelu, and Dr. M.O. Faruque Sarker. *Python Network Programming*. Packt Publishing Ltd, Jan. 2019. ISBN: 978-1-78883-546-6.

1.3.2 Online Resources

- *python™*. Python Software Foundation (PSF), 2001–2025. URL: https://python.org:
  - *Python 3 Documentation*. Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3,
  - *Python 3 Documentation*. *Python Setup and Usage*. Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/using,
  - *Python 3 Documentation*. *The Python Tutorial*. Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/tutorial,
  - *Python 3 Documentation*. *The Python Standard Library*. Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/library,
  - *Python 3 Documentation*. *Python HOWTOs*. Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/howto,

  – The PEP Editors. *Index of Python Enhancement Proposals (PEPs)*. Python Enhancement Proposal (PEP) 0. Python Software Foundation (PSF), July 13, 2000. URL: https://peps.python.org.

- *Real Python Tutorials*. DevCademy Media Inc., 2021–2025. URL: https://realpython.com,
- *W3Schools: Python Tutorials*. Refsnes Data AS, 1999–2025. URL: https://www.w3schools.com/python,
- Trey Hunner. *Python Morsels*. Python Morsels, 2025. URL: https://www.pythonmorsels.com,
- Florian Bruhin. *Python f-Strings*. Bruhin Software, May 31, 2023. URL: https://fstring.help,
- Benjamin Dicken. *PyFlo – The Beginners Guide to Becoming a Python Programmer*. 2023. URL: https://pyflo.net.

Part I

Basics

In this part of the book, we will gain the following abilities:

- read, write, and execute simple Python programs
- use some tools to help us look for errors

In particular, we will learn how to use the Python console and the editor PyCharm. We will learn about the most important simple datatypes in Python, such as integer numbers, floating point numbers, strings, and Boolean values. We will learn how to store values in variables. And we will learn about the most important collection datatypes offered by Python, namely lists, tuples, sets, and dictionaries.

Chapter 2

Getting Started

This course should be a practical course, so we should get started with practical things right away. In order to do practical things, we need to have all the necessary software on our computer. What software is necessary to do Python programming? Well, first of all, Python. If Python is not yet installed on your machine, then you can follow the brief installation guide in Section 2.1.

Now with the programming language Python alone, you cannot really do much – in a convenient way, at least. You need a nice editor in which you can write the programs. Actually, you want an editor where you can not just write programs. You want an editor where you can also directly execute and test your programs. In software development, you often work with a Version Control System (VCS) like Git. You want to do that conveniently from your editor. Such an editor, which integrates many of the common tasks that occur during programming, is called an Integrated Development Environment (IDE). In this book, we will use the PyCharm IDE [586, 644]. If you do not yet have PyCharm installed, then you can work through the setup instructions outlined in Section 2.2.

Before we get into these necessary installation and setup steps that we need to really learn programming, we face a small problem: Today, devices with many different Operating System (OS) are available. For each OS, the installation steps and software availability may be different, so I cannot possibly cover them all. Personally, I strongly recommend using Linux [27, 253, 573] for programming, work, and research. If you are a student of computer science or any related field, then it is my personal opinion that you should get familiar with this operating system. Maybe you could start with the very easy-to-use Ubuntu Linux [121, 255]. Either way, in the following, I will try to provide examples and instructions for both Ubuntu and the commercial Microsoft Windows [71] OS.

Once the necessary software is installed, we will here also learn how to write and execute our very first small Python program in Section 2.3This first program will just print "Hello World!" and then exit. As a final step to get our environment up and running, we discuss how Python programs can be executed in the terminal and the Python console in Section 2.4

2.1 Installing Python

In order to learn and use Python, we first need to install it. There are two major versions of Python out there: Python 2 and Python 3. This book focuses entirely on Python 3. We assume that you have installed Python 3.12 or newer. We here provide some brief setup instructions. More help can be found at the following resources:

- the official Python setup and usage page *Python 3 Documentation*. *Python Setup and Usage*. Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/using,
- the Python Downloads at https://www.python.org/downloads, and
- the Python 3 Installation & Setup Guide at https://realpython.com/installing-python.

2.1.1 Python under Ubuntu Linux

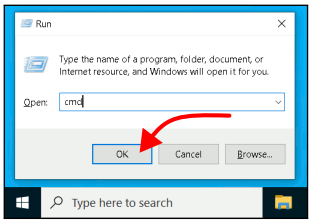
Under Ubuntu Linux, Python 3 is already pre-installed. You can open a terminal [27] by pressing Ctrl + Alt + T, then type in `python3 --version`, hit ↵, and get the result illustrated in Figure 2.1:



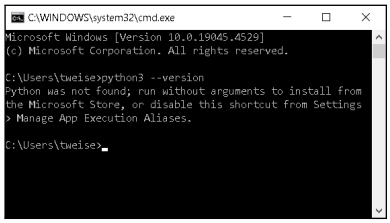
Figure 2.1: Under my Ubuntu Linux 22.04 system, typing `python3 --version` in a terminal and hitting return yields version 3.10.12.

2.1.2 Installing Python 3 under Microsoft Windows

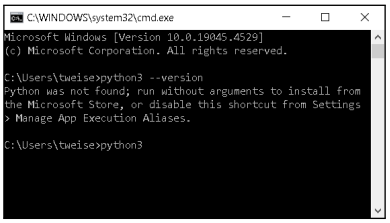
Example installation steps for Python on Microsoft Windows (version 10) are sketched in Figure 2.2. First, you would open a terminal using and press ⊞ + R, type in `cmd`, and hit ↵, as shown in Figure 2.2.1. If Python is installed, then typing `python3 --version` in the terminal and hitting ↵ would print the version of the Python installation. If it is not installed, however, then Microsoft Windows will print a message informing you that Python is not yet installed and that you can reach the web installer by just typing `python3` (and hitting ↵, of course). We do this in Figure 2.2.3, which leads us to the installation screen (Figure 2.2.4), where we need to press the Get button. This will then download (Figure 2.2.5) and install Python. When this process is completed, the screen just shows nothing (Figure 2.2.6). If we go back to the terminal and again try `python3 --version`, it will now work and print the version of our Python installation. In our case, this is version 3.12.



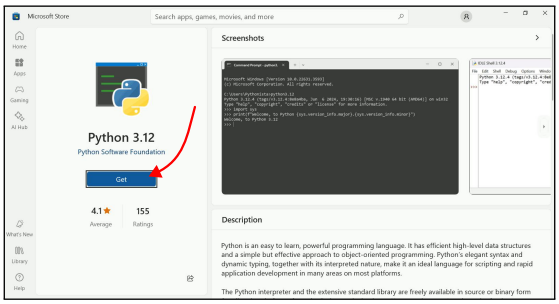
(2.2.1) Opening the terminal under Microsoft Windows: press ⊞+R, type in cmd, and hit ↵.



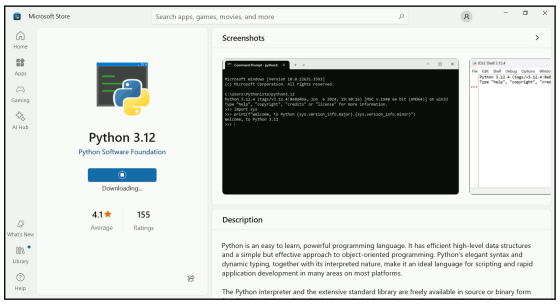
(2.2.2) Trying to get the Python version via `python3 --version`, but it is not installed.



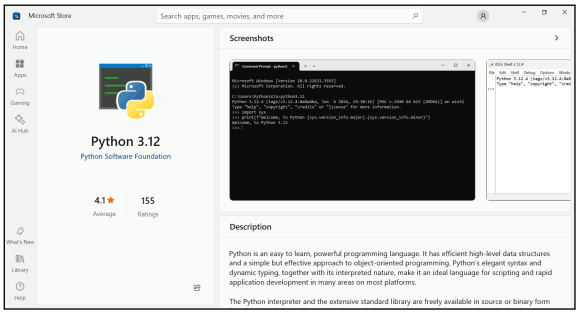
(2.2.3) Installing it by typing `python3` and hitting ↵.



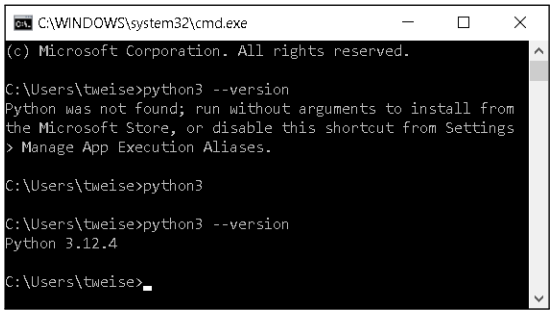
(2.2.4) The install screen, where we click Get.



(2.2.5) The install screen, downloading Python.



(2.2.6) The installation is finished.



(2.2.7) And the `python3 --version` command now works in the terminal.

Figure 2.2: Cropped screenshots of the installation steps for Python on Microsoft Windows.

2.2 Installing PyCharm

Just having a programming language and the corresponding interpreter on your system is not enough. Well, it is enough for just running Python programs. But it is not enough if you want to develop software efficiently. Are you going to write programs in a simple text editor like a caveperson? No, of course not, you need an IDE, a program which allows you to do multiple of the necessary tasks involved in the software development process under one convenient user interface. For this book, I recommend using PyCharm [586, 636, 644], which is freely available. The installation guide for PyCharm can be found at https://www.jetbrains.com/help/pycharm/installation-guide.html.

Notice that, as shown in Figure 2.4, the PyCharm Community Edition will be/has been replaced with a unified edition. This means that the instructions in the following are probably outdated, but they should still give you a reasonably good impression on what needs to be done. We will probably eventually replace them ... but not now.

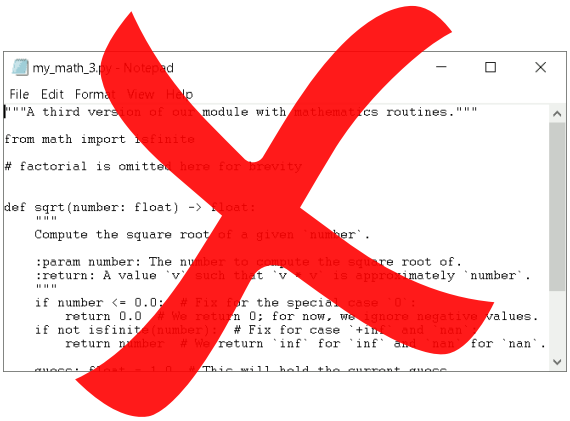


Figure 2.3: We do not want to use the `NotePad`-App of Microsoft `Windows` for programming, do we?

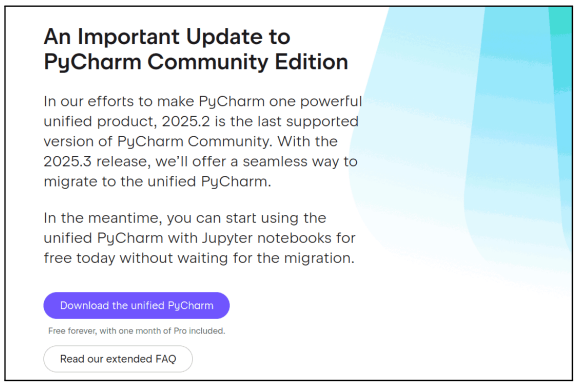
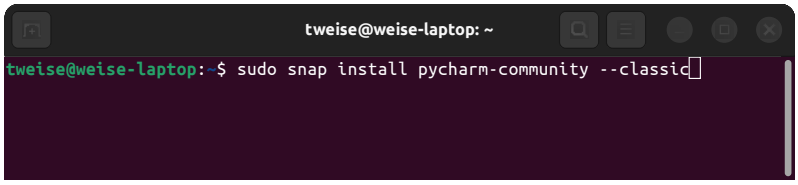


Figure 2.4: Some time after we finished preparing the installation instructions for the `PyCharm` Community Edition that you can find in the remainder of this section, it was announced that a unified `PyCharm` version will replace the Community Edition. Therefore, the download instructions will probably change. We will eventually update the instructions, but not now.

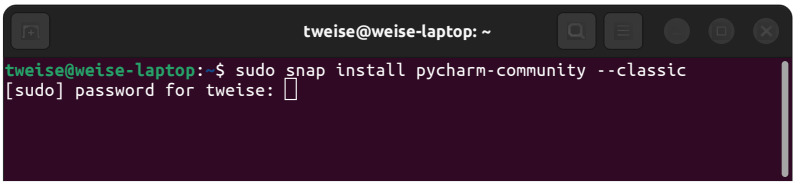
2.2.1 Installing PyCharm under Ubuntu Linux

`PyCharm` is available as Snap package under `Ubuntu` `Linux` [454, 523]. The installation process is very easy and follows the steps illustrated in Figure 2.5. First, you open a terminal by pressing Ctrl + Alt + T. Then, enter the command `sudo snap install pycharm --classic` and hit ↵ (Figure 2.5.1). This installs the `PyCharm` software package and the necessary super user privileges are obtained via the pre-pended `sudo`, which will ask us to enter the root password, as sketched in Figure 2.5.2. Then, the installation process basically runs automatically. Once it has completed (see Figure 2.5.5), you can press the ⊞ key and type `pycharm` in the launcher window to find `PyCharm` (Figure 2.5.6). A double-click will open `PyCharm`.

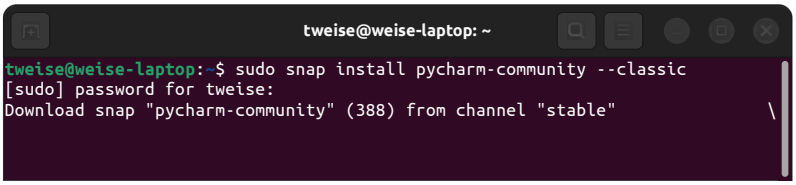
In the installation instructions for Microsoft `Windows`, now a user agreement (Figure 2.6.16) and data upload statement (Figure 2.6.17) need to be performed. Since I already had `PyCharm` installed previously and probably already agreed/disagreed to them, respectively, these windows did not open in my current installation and I could not take screenshots of them. If they open, then you can probably treat them exactly as suggested in the Microsoft `Windows` installation instructions Section 2.2.2. Either way, we finally get `PyCharm` up and running and can begin our coding (Figure 2.5.8).



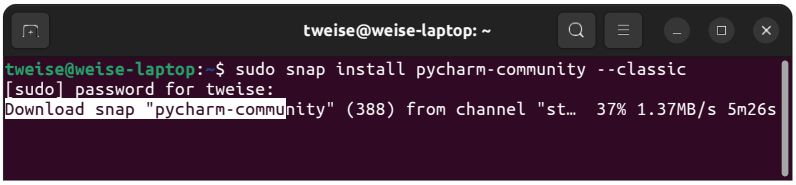
(2.5.1) Installing PyCharm using the `snap install` command in a terminal opened with Ctrl+Alt+T.



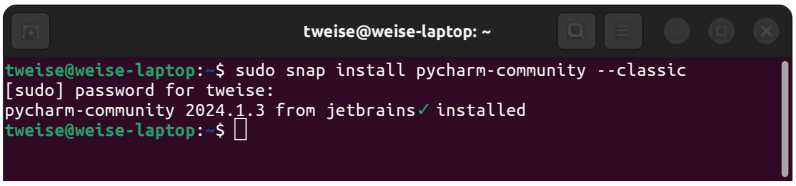
(2.5.2) This command requires the super user password, which we type in and then press ↵.



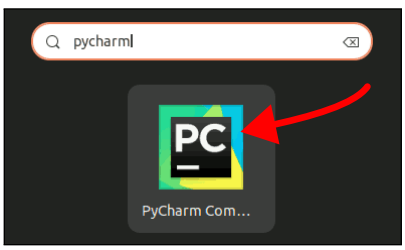
(2.5.3) The installation begins.



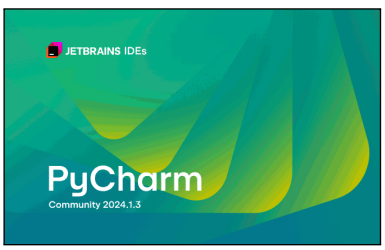
(2.5.4) The software package is downloaded.



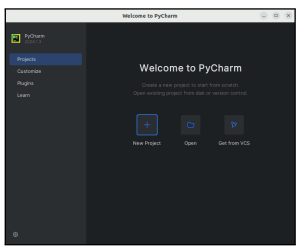
(2.5.5) The package is installed.



(2.5.6) Open the launcher by pressing ⊞ and type in `pycharm` to find the PyCharm executable, then double-click it.



(2.5.7) The PyCharm welcome screen appears.

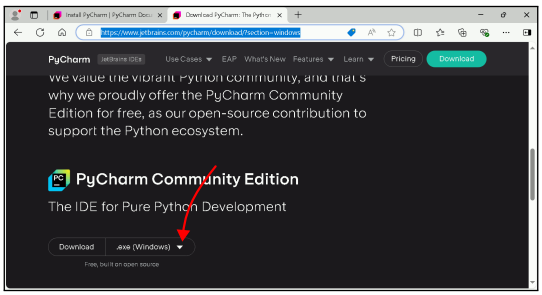


(2.5.8) PyCharm has been started.

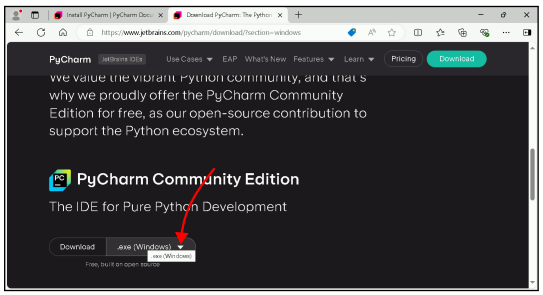
Figure 2.5: The installation steps of PyCharm under Ubuntu linux.

2.2.2 Installing PyCharm under Microsoft Windows

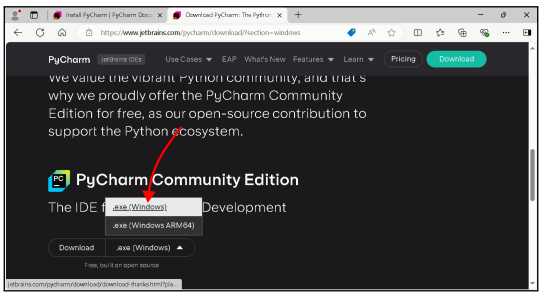
The process of installing PyCharm under Microsoft Windows is illustrated in Figure 2.6. You first need to download the PyCharm installation executable from https://www.jetbrains.com/pycharm/download. Make sure to download PyCharm and nothing else, as shown in Figures 2.6.1 to 2.6.3. Once the installer is downloaded, you start it and confirm that you wish to install PyCharm, as illustrated in Figures 2.6.5 to 2.6.8. As Figures 2.6.9 to 2.6.11 show, the installation setup process is more or less automated, we just need to click Next here and there and finally click Install (Figure 2.6.12). After the installation completes, we run PyCharm for the first time. Now we need to decide whether we want to agree to the user agreement (Figure 2.6.16) and may wish to choose that we do not want any information about our PyCharm usage being sent to anyone (Figure 2.6.17). Finally, as sketched in Figure 2.6.18, we have a running and ready PyCharm IDE.



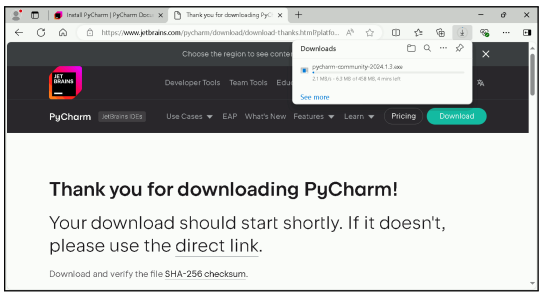
(2.6.1) The download website https://www.jetbrains.com/pycharm/download for PyCharm.



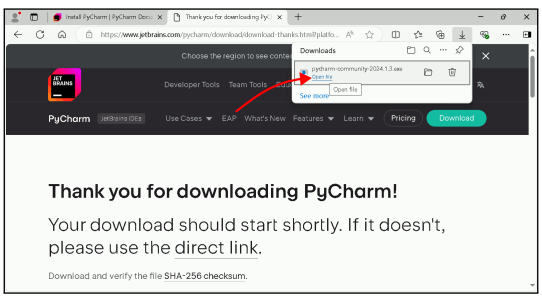
(2.6.2) Downloading the PyCharm Community Edition.



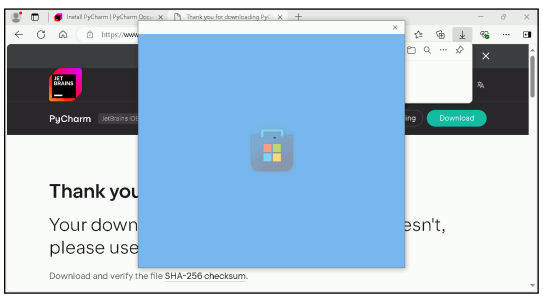
(2.6.3) Selecting the normal Microsoft Windows installer for the PyCharm Community Edition.



(2.6.4) The download is starting.

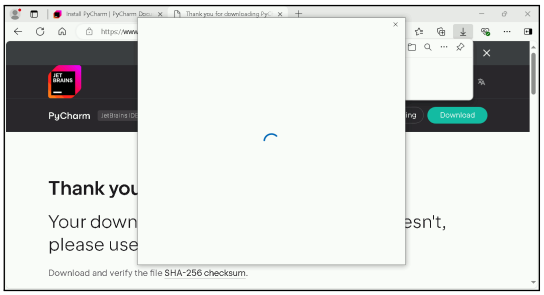


(2.6.5) The download is completed. We click Open file.

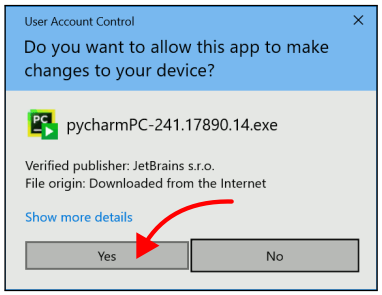


(2.6.6) The installer is starting.

Figure 2.6: The installation steps of PyCharm under Microsoft Windows.



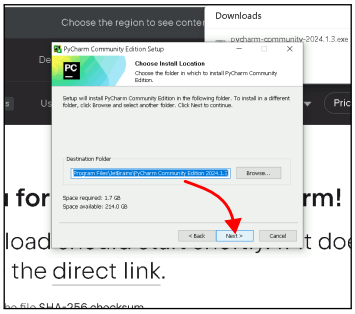
(2.6.7) The installer is starting.



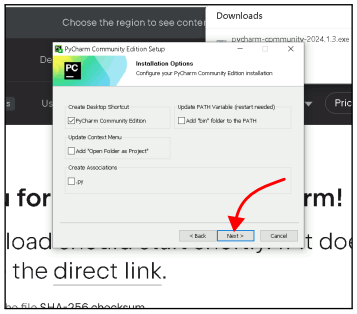
(2.6.8) We do want to install, so click Yes.



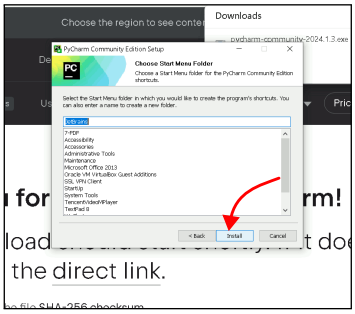
(2.6.9) The welcome screen of the installer. We click Next.



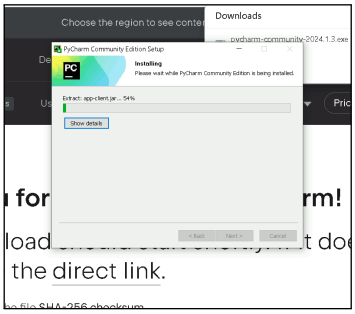
(2.6.10) The installation folder selection. We click Next.



(2.6.11) The installation options. We can click Next.



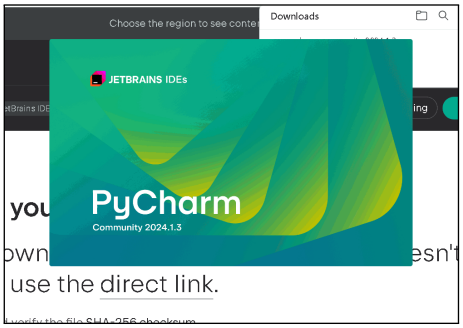
(2.6.12) The start menu folder choose dialog. We click Install.



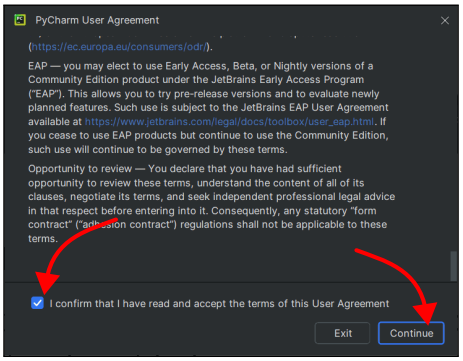
(2.6.13) The install process starts.



(2.6.14) The installation is finished. Select "Run PyCharm Community Edition" and click Finish.

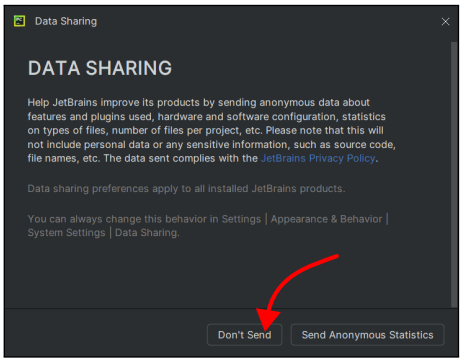


(2.6.15) The welcome screen of PyCharm.

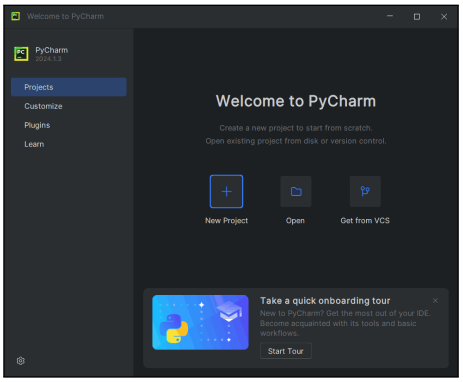


(2.6.16) We read the user agreement and – if we want to agree – confirm that we read it, and click Continue.

Figure 2.6: The installation steps of PyCharm under Microsoft Windows (continued).



(2.6.17) We do not want to send any data and click Don't Send.



(2.6.18) Finally, PyCharm is ready to use.

Figure 2.6: The installation steps of PyCharm under Microsoft Windows (continued).

2.3 Our First Program

We now want to write and execute our very first Python program. This program should just print "Hello World!" to the standard output stream (stdout) stream and then exit. It therefore will consist of the single statement `print("Hello World!")` in file very_first_program.py, as illustrated in Listing 2.1.

In PyCharm, we usually will not just create a single Python source code file. Instead, we will work in the context of *projects*, which can contain many Python files, settings, build scripts, and other resources. Inside such a project, we would create the Python file, write our source code from Listing 2.1 into it, and then run it. The steps for doing this are illustrated with screenshots in Figure 2.7.

Since we started with a completely new PyCharm installation in Section 2.2, no project has yet been created. So when we open PyCharm, we will arrive at the project creation screen illustrated in Figure 2.7.1. Here, we will New Project, which takes us to the second project creation screen shown in Figure 2.7.2. In case that you already had PyCharm installed and already created some projects in the past, you can get to the same screen by clicking ≡ File New Project. Either way, when arriving at the screen, you will find several confusingly looking options. First, make sure that Pure Python is selected in the left pane. Then you can choose a name for the project in the Name: text box. For the sake of our example, let's call the new project `veryFirstProject`.

Below the name text box, you can select the destination directory in which the project folder should be created in the Location: box. In the screenshot, I have cropped out the directory path to avoid confusion. You will need to use a different location and paths also look different on Linux and Microsoft Windows. So you will choose some suitable directory on your computer. However, you may choose the same project name – `veryFirstProject` – *inside* your chosen directory. This name will then become a new folder and this folder will contain all the project files.

The following options may not make any sense for you, which is totally OK for now. Please make sure to select Interpreter Type: Custom Environment as well as Environment: Select existing and Type: Python. Under Python Path:, you would choose the Python interpreter you have installed on your system (see Section 2.1). On my system that was "Python 3.12.3" at the time of this writing.[1] We

[1] Now I am using Python 3.12, but this is not important.

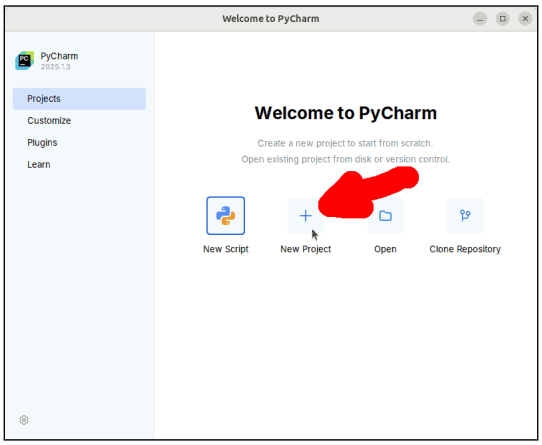
Listing 2.1: Our very first Python program, which just prints "Hello World!" (stored in file very_first_program.py; output in Listing 2.2)

```
1 print("Hello World!")
```

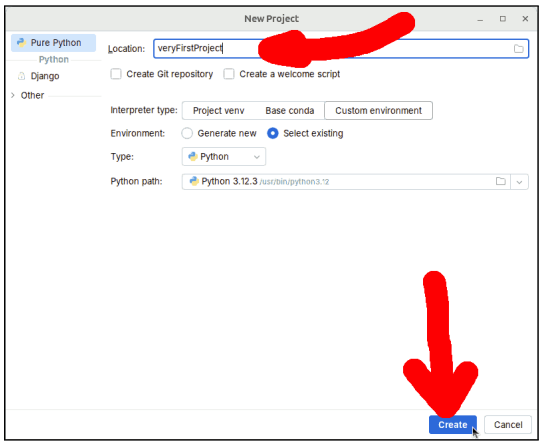
↓ `python3 very_first_program.py` ↓

Listing 2.2: The stdout of the program very_first_program.py given in Listing 2.1.

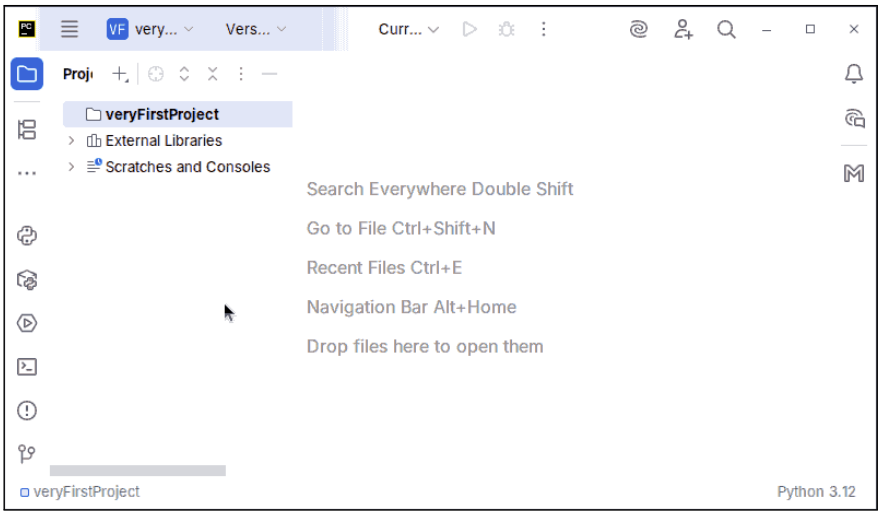
```
1 Hello World!
```



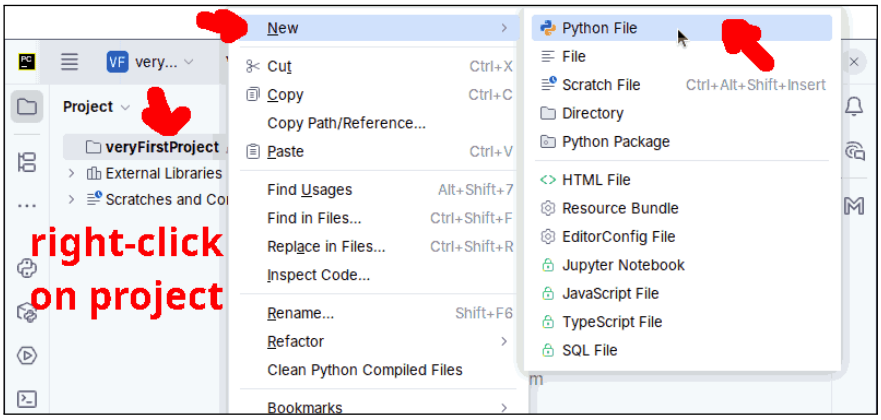
(2.7.1) To create a new project in PyCharm, we click on New Project in the opening screen.



(2.7.2) Make sure that Pure Python is selected in the left pane, then select a name for the project. We here choose veryFirstProject. Also choose a directory location where your project will be stored. We leave the other settings at the default and/or select the current Python installation as Custom Environment. We finally click Create.

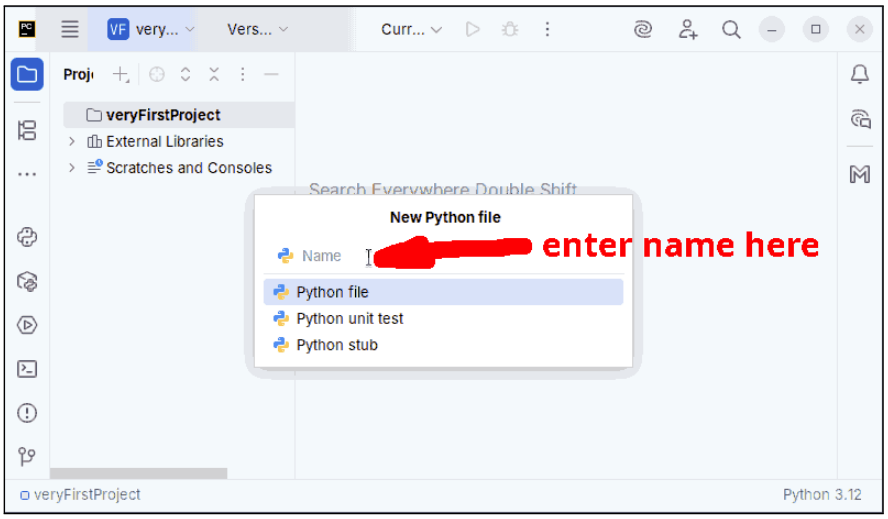


(2.7.3) The new and empty project has been created.

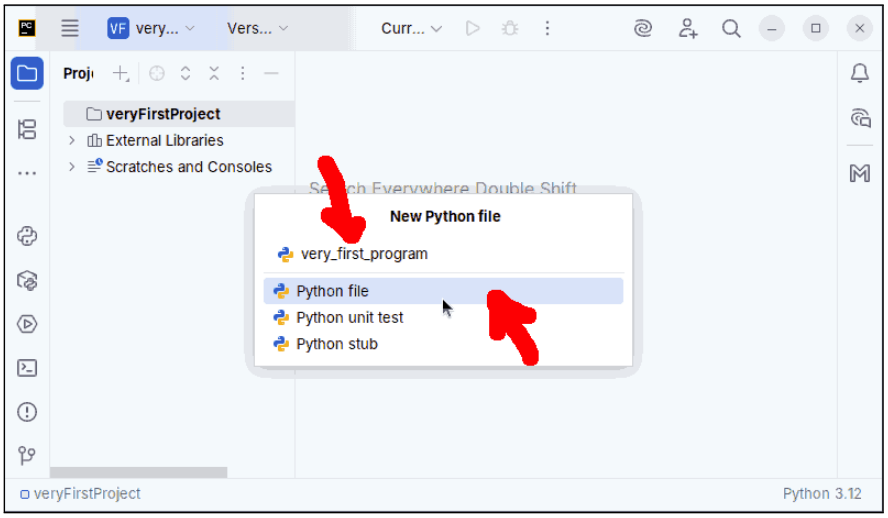


(2.7.4) We create a new Python file within this project by right-clicking on the project folder veryFirstProject and selecting New » Python File.

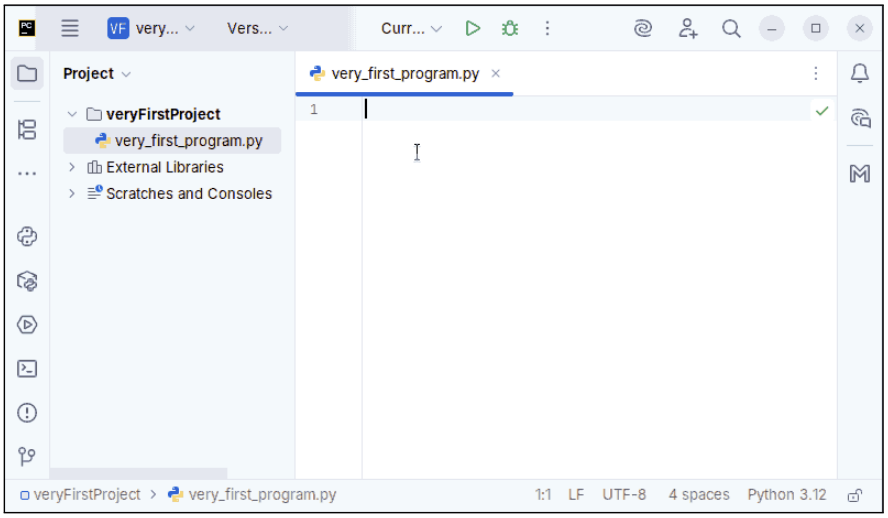
Figure 2.7: The steps to create a new Python file in a new PyCharm project and to then run it (Ubuntu).



(2.7.5) We now can enter a name for the new `Python` file.

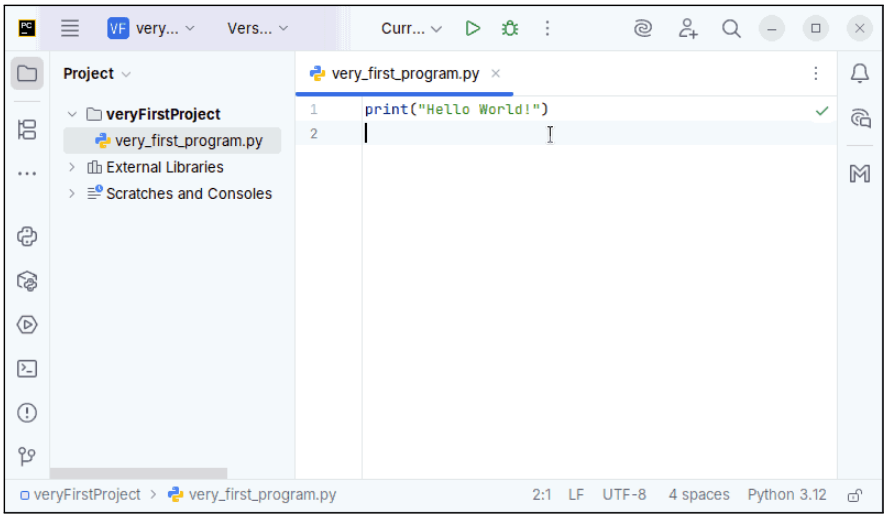


(2.7.6) We enter a name for the new `Python` file (here: `very_first_program`) and hit ↵.

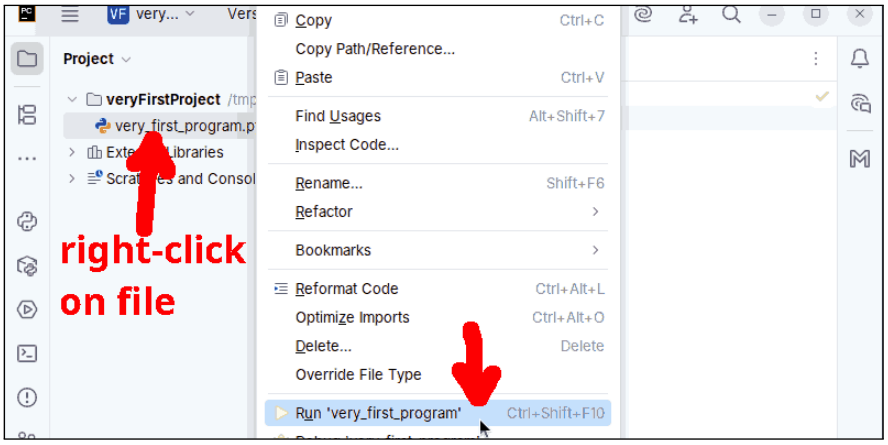


(2.7.7) The new and empty file `very_first_program.py` has been created in the project `veryFirstProject`.

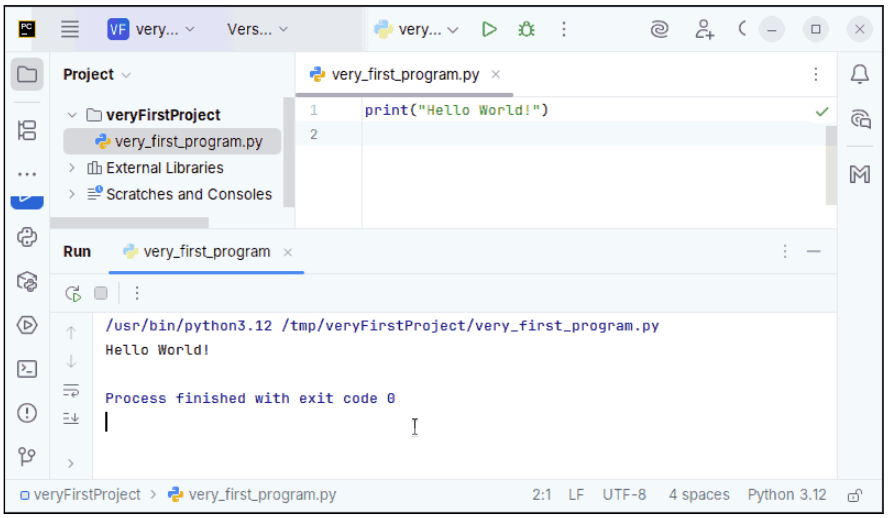
Figure 2.7: The steps to create a new `Python` file in a new `PyCharm` project and to then run it (`Ubuntu`, continued).



(2.7.8) We enter the text from Listing 2.1. PyCharm automatically saves it.



(2.7.9) In order to run this program, we right-click on the program file in the tree view and select Run 'very_first_program'. Alternatively, we could press Ctrl+⇧+F10.



(2.7.10) And indeed, in the console pane at the bottom of the PyCharm window, the text "Hello World!" appears.

Figure 2.7: The steps to create a new Python file in a new PyCharm project and to then run it (Ubuntu, continued).

finally click Create. We now have created our first and empty PyCharm Python project – as illustrated in Figure 2.7.3.

We can now create the Python file to write our actual program code. To do this, we right-click on the folder veryFirstProject in our Project tree view pane on the left-hand side (see Figure 2.7.4). In the menu that pops up, we select and click New > Python File. This takes us the New Python file creation dialog illustrated in Figure 2.7.5. Here, we enter the name for our first program as shown in Figure 2.7.6. What could be more fitting than very_first_program? After hitting Enter, the file is created in our project folder and opened in the editor pane, as shown in Figure 2.7.7.

We now enter the single line of code from Listing 2.1 into the editor, as illustrated in Figure 2.7.8. We do not need to explicitly save the file, as PyCharm does this automatically for us.

Finally, we want to actually execute, i.e., run, our program. We can do this directly from the editor by pressing Ctrl+⇧+F10. We can also do it by right-clicking on the file in the project tree view on the left-hand side and then clicking Run 'very_first_program' in the popup menu, as shown in Figure 2.7.9.

Either way, a console with the title "very_first_program" opens at the bottom of our editor. And behold: Indeed, the text `Hello World!` appears.

Well, before that text, we see the command line that was actually executed, namely the Python interpreter with our file's path as parameter, as illustrated in Figure 2.7.10. And after our program's output, we are notified that "Process finished with exit code 0," which means that the program has completed successfully and without error.

Congratulations. You now have written, saved, and executed your first ever Python program!

2.4 Python in the Terminal

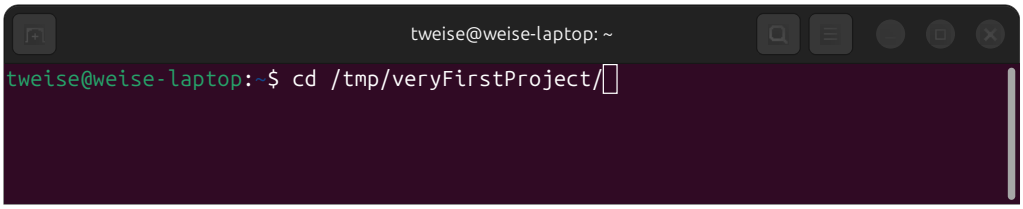
In total, are four more ways in which we can execute a Python program:

1. We can enter the program into a Python file in the PyCharm IDE and then run it from there. We just did this in the previous section and illustrated it in Figure 2.7.
2. Actually, we can also write a Python program with a normal text editor. A Python program is just a normal text file, after all. We can execute such a text file by entering its directory and typing `python3 programName` (where `programName` is `very_first_program.py`, in our case) and hitting ↵. Then the program is executed directly in the terminal. This process is shown in Section 2.4.1 and Figure 2.8.
3. Alternatively, we could open the Python interpreter console in PyCharm and enter and execute our code line-by-line. This is sketched in Section 2.4.2 and Figure 2.9.
4. Besides using the Python console inside PyCharm, we can also open it inside a terminal. We can then enter separate Python instructions and run them there. This fourth option is outlined in Section 2.4.3 and Figure 2.10.

2.4.1 Executing a Python Program in a Terminal

In order to directly execute a Python program in a terminal, we first need to open one. Under Ubuntu Linux, we simply press Ctrl+Alt+T. Under Microsoft Windows, we have to press ⊞+R, type in `cmd`, and hit ↵. Once the terminal is open, we need to change into the directory where the program is located. Under both Linux and Microsoft Windows, this can be done by typing the command `cd`, followed by the path to the directory, and hitting ↵.[2] We provide a screenshot for that, taken under Ubuntu Linux, in Figure 2.8.1. Now we simply call the Python interpreter by writing `python3` followed by the file name of our program, which is `very_first_program.py` in our case. In Figure 2.8.3 we do this and hit ↵, which causes the Python interpreter to execute our program. The output "Hello World!" is then printed into the terminal in Figure 2.8.4.

[2] Under Microsoft Windows, you may also need to change into the correct drive first.



(2.8.1) Open a terminal (by pressing Ctrl + Alt + T under Ubuntu Linux; under Microsoft Windows press ⊞ + R, type in cmd, and hit ↵). Change into the directory "directory" where your Python file is located, by typing cd directory and hit ↵. (I here had it in the temporary directory /tmp, you will have it elsewhere...)



(2.8.2) We are now in the project directory.



(2.8.3) Execute a Python program "program.py" by typing python3 program.py and hit ↵. In our case, the program is "very_first_program.py".



(2.8.4) As expected, the text "Hello World!" appears in the terminal.

Figure 2.8: Example of executing a Python program in a terminal (on Ubuntu).

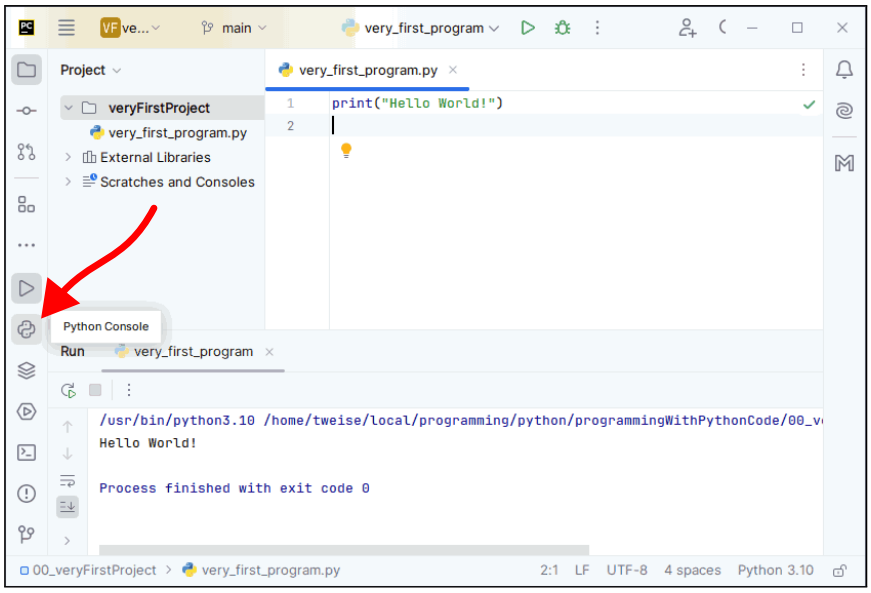
2.4.2 Entering Commands in the Python Console inside PyCharm

Besides writing programs in files and executing them, we can also directly enter them into the Python console and execute them step-by-step. This does not make sense if we want to reuse our programs later. But it does make a lot of sense when we just want to test some commands or functions or quickly test some idea. A Python console can be used directly in PyCharm (Figure 2.9) or opened in a terminal (Figure 2.10).

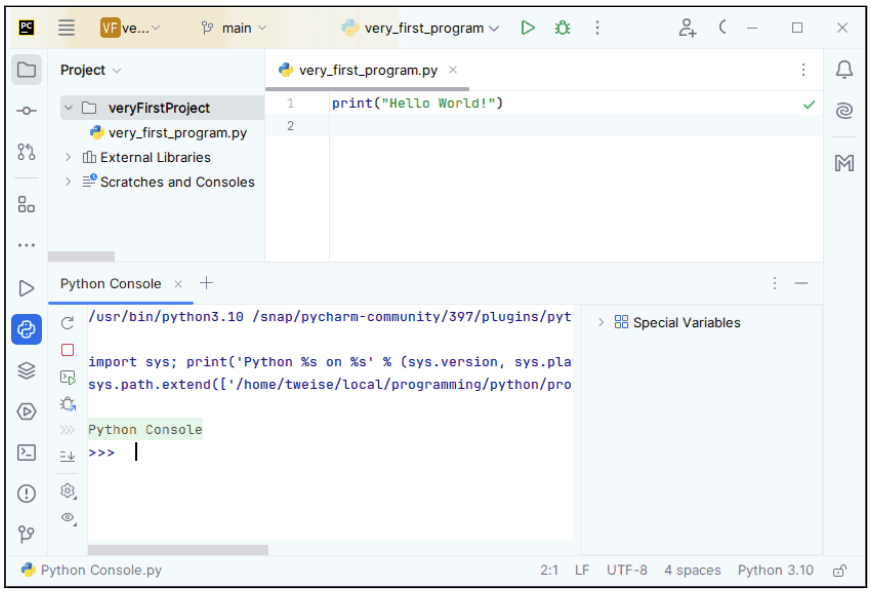
To explore entering Python code in the Python console inside PyCharm, we continue where we left of in Section 2.3. In PyCharm, we first click the on the vertical icon bar on the left side of the PyCharm window, as shown in Figure 2.9.1. This directly brings us to the Python console (Figure 2.9.2). We can enter the one-line-program from Listing 2.1, as illustrated in Figure 2.9.3. Notice that the input prompt of the console is marked by the three greater characters >>> after which we enter our text. Pressing ↵ after writing the code leads to the expected output shown in Figure 2.9.4. This output directly appears in the console and is not preceded by any other text, in particular not by >>>, which makes it easy to visually distinguish what the input and output in a Python console are.

2.4.3 Entering Commands in the Python Console in a Terminal

Let us now open a Python console from the terminal instead of using the one in PyCharm. We therefore first need to open a normal terminal. Under Ubuntu Linux, we simply press Ctrl + Alt + T. Under



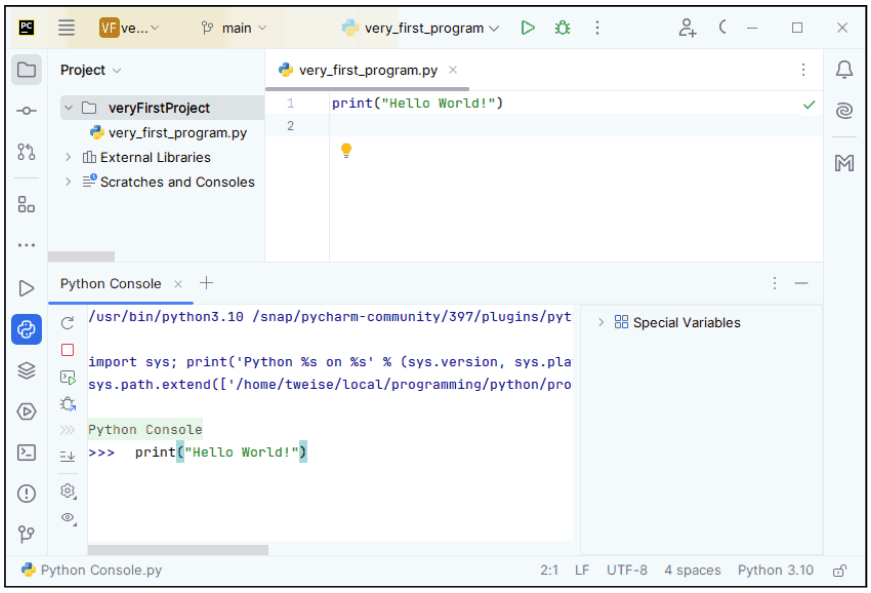
(2.9.1) Pressing the [Python Console] on the vertical icon bar on the left side of the PyCharm window.



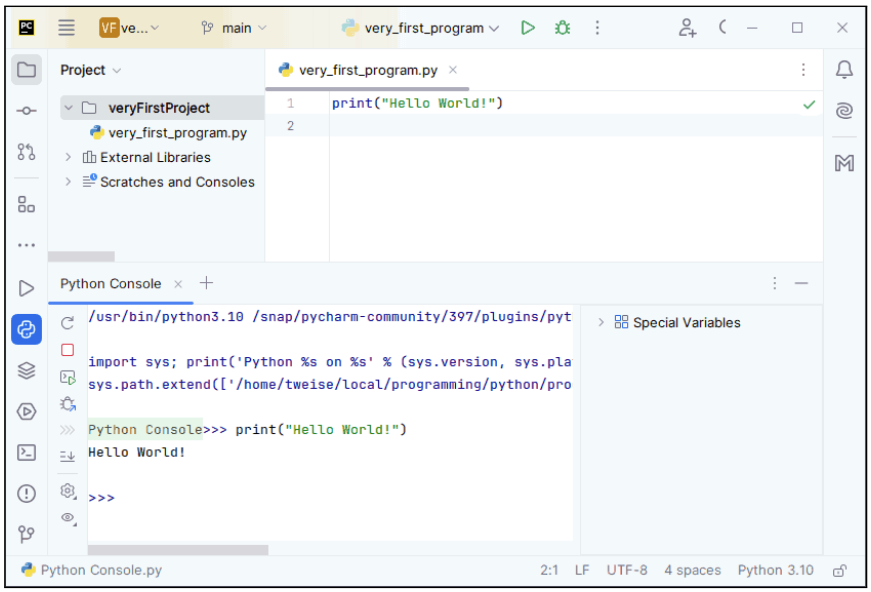
(2.9.2) The PyCharm Python console is open.

Figure 2.9: Entering the "Hello World!" program from Listing 2.1 directly into the Python console offered by PyCharm.

Microsoft Windows, we have to press [⊞]+[R], type in `cmd`, and hit [↵]. Either way, the terminal opens and we can enter `python3` and press [↵], as shown in Figure 2.10.1. Now the Python interpreter starts right inside the terminal (Figure 2.10.2). The prompt, i.e., the place where we can write our code, again is preceded by the `>>>` characters. As illustrated in Figure 2.10.3, we copy the single line of code, `print("Hello World!")` from Listing 2.1 and press [↵]. The output "Hello World!" is printed as expected in Figure 2.10.4. However, we now are still in the Python interpreter. In order to leave it and to, maybe, enter other commands in the terminal, we have to use another new Python instruction: We type in `exit()` and press [↵], as shown in Figure 2.10.5, which causes the Python interpreter to exit. We are now back in the basic terminal, as shown in Figure 2.10.6. In these figures, I was using Ubuntu Linux. On Microsoft Windows or other Linux variants, the process would have looked quite similar.



(2.9.3) We enter the "Hello World!" program from Listing 2.1 and press ↵.



(2.9.4) And indeed, the output is "Hello World!".

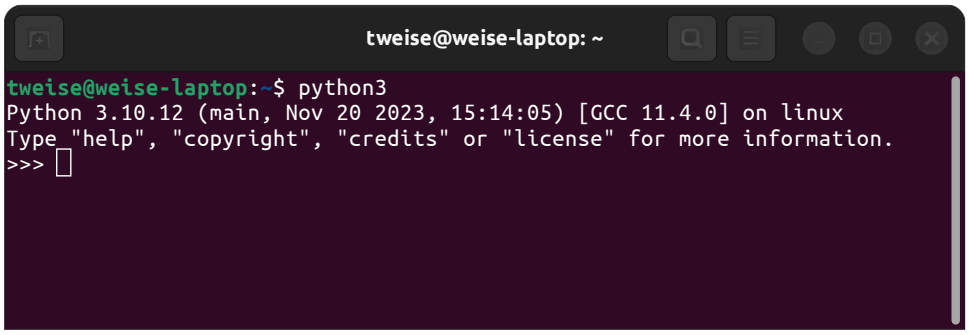
Figure 2.9: Entering the "Hello World!" program from Listing 2.1 directly into the `Python` console offered by `PyCharm` (continued).

Best Practice 1

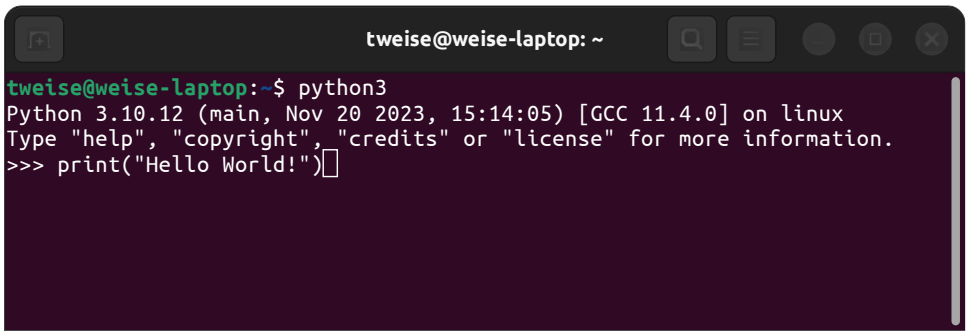
The only proper way to run a `Python` application in a productive scenario is in the terminal, as shown in Section 2.4.1.



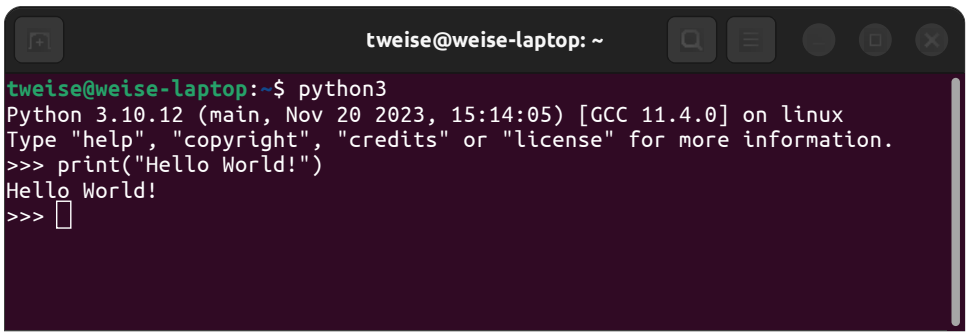
(2.10.1) Open a terminal (by pressing Ctrl+Alt+T under Ubuntu Linux; under Microsoft Windows press ⊞+R, type in cmd, and hit ↵), enter python3, then hit ↵.



(2.10.2) The Python console opens in the terminal.

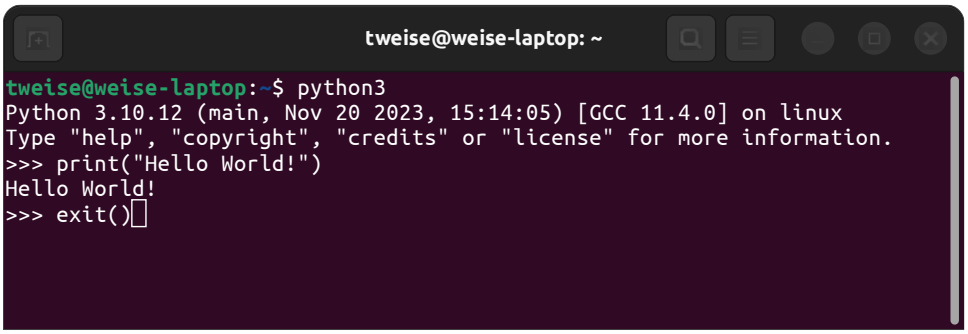


(2.10.3) We enter the "Hello World!" program from Listing 2.1 and press ↵.

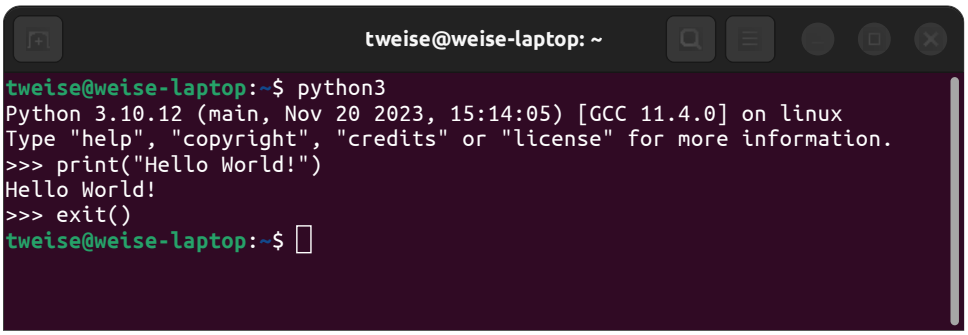


(2.10.4) And indeed, the output is "Hello World!".

Figure 2.10: Writing a program in the Python console in the terminal (Ubuntu).



(2.10.5) We exit the console by typing `exit()` and pressing ↵.



(2.10.6) We are back in the normal terminal.

Figure 2.10: Writing a program directly in the Python console in the terminal (Ubuntu, continued).

2.5 Getting the Examples from this Book

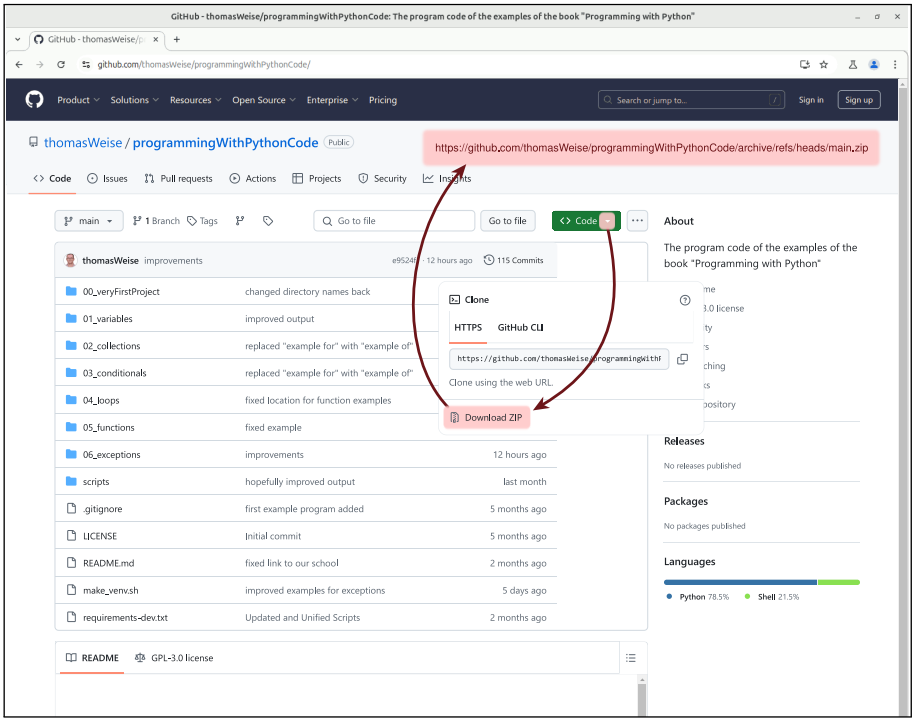
This book comes with a lot of example programs written in Python. While our first explorations of the simple data types will mainly use the Python console, we will later almost exclusively write programs in Python files. Every single one of them is available in the Git repository programmingWithPython Code. You can directly access this repository at GitHub under https://github.com/thomasWeise/programmingWithPythonCode.

From the website, you can directly download all the examples as illustrated in Figure 2.11.1. First, you would click on the little downward facing triangle in the button Code as shown in Figure 2.11.2. This will open a small dialog Clone in which you can click on the Download ZIP button (see Figure 2.11.3). This, in turn, enters the Uniform Resource Locator (URL) https://github.com/thomasWeise/programmingWithPythonCode/archive/refs/heads/main.zip into your browser's download queue.

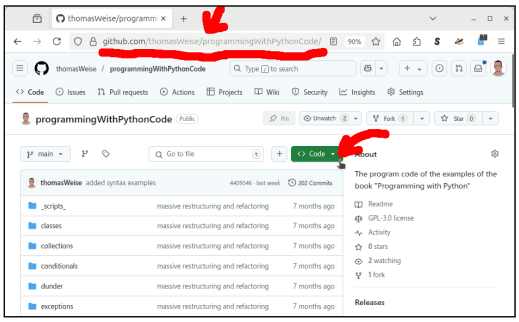
The download will eventually finish, as shown in Figure 2.11.4, at which point you can open the downloaded zip archive by clicking on it as sketched in Figure 2.11.5. A zip archive is a single file that can contain other files and folders and can be opened by the standard file managers both on Ubuntu and Microsoft Windows. Once downloaded, the archive contains all the examples that we use in our book. Its root folder is named similarly to the repository (see Figure 2.11.6). Inside this folder, you can find all the folders with examples, including the example `veryFirstProject` we just discussed in the previous section, as shown in Figure 2.11.7.

Alternatively to downloading a zip archive with the examples from this book, you can also directly create a new project in PyCharm by cloning (basically, downloading) the repository as illustrated in Figure 2.12. You see, our examples are all located in a so-called Git repository [515, 577]. Git is a Version Control System (VCS) [515, 577], i.e., a version management system for software development. With such a system, we can iteratively work on our code and commit changes to the code base. The VCS remembers the history of our projects and allows us to share and collaboratively work on the code. Well, we will not collaboratively work on this code with each other, because the code represents artificial examples of a course. But Git is a very well-known and widely-used VCS, so at least getting familiar with it does not hurt. Our examples are hosted in a Git repository on GitHub [441, 577]. And you can clone it just as well instead of just downloading it.

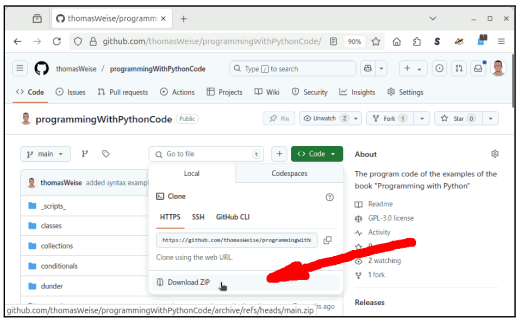
In the PyCharm welcome screen, you click Clone Repository as shown in Figure 2.12.1. In the



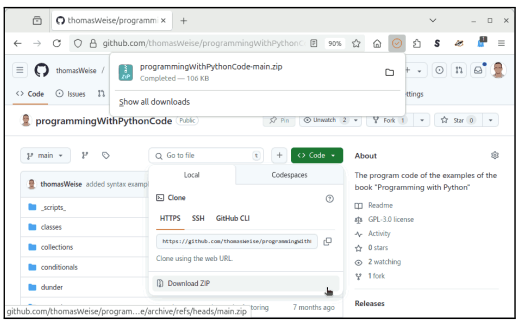
(2.11.1) How to download the examples.



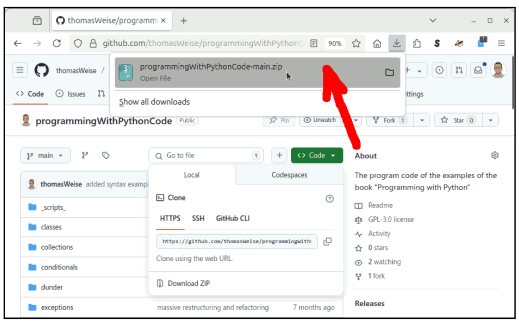
(2.11.2) Visit the website `https://github.com/thomasWeise/programmingWithPythonCode`. Then click on the downward facing triangle in the Code button.



(2.11.3) In the popup menu that appears, click Download ZIP.

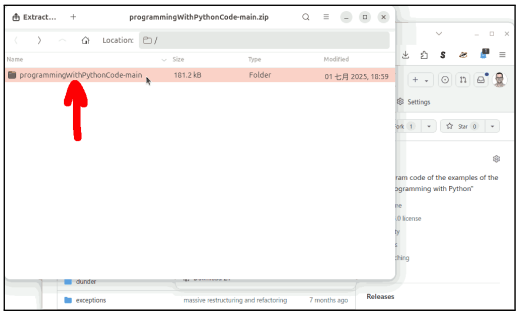


(2.11.4) The `zip` archive is downloaded.

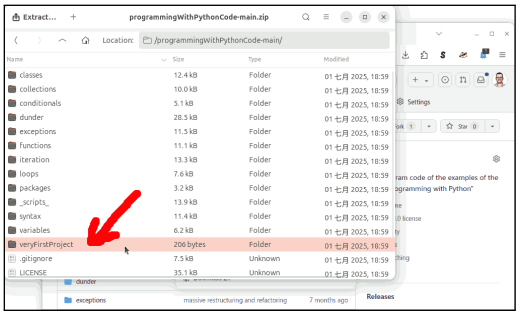


(2.11.5) We click on it to open it.

Figure 2.11: Downloading all the example source codes as a single `zip` archive from `https://github.com/thomasWeise/programmingWithPythonCode`.



(2.11.6) The archive opens and shows us the root folder of the archive. We open this folder.

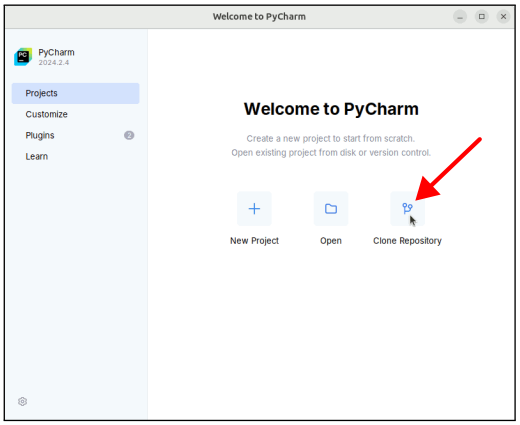


(2.11.7) Now you can see all the example folders. The folder `veryFirstProject` includes the Hello-World example we just discussed.

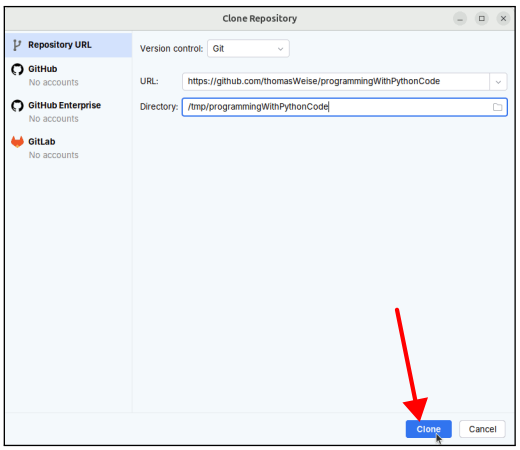
Figure 2.11: Downloading all the example source codes as a single `zip` archive from `https://github.com/thomasWeise/programmingWithPythonCode` (Continued).

next dialog, you have to select a source URL:, which will be `https://github.com/thomasWeise/programmingWithPythonCode`. You also need to choose a Directory: where the new project should be located. All the contents of the examples repository will be downloaded into this directory as well. In Figure 2.12.2, I selected `/tmp/programmingWithPythonCode`, i.e., a directory on my partition for temporary files. This directory will be cleared at every system boot, so you would certainly choose a more reasonable destination. After clicking Clone, the downloading will begin, as sketched in Figure 2.12.3.

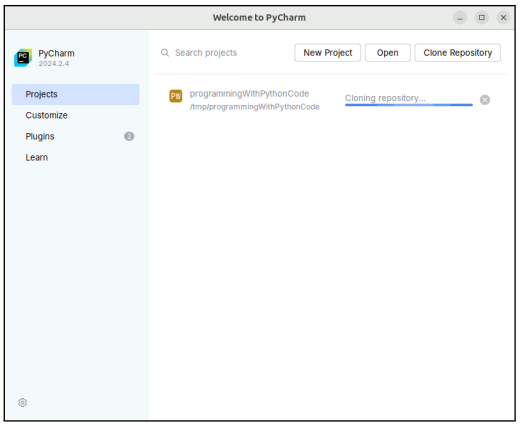
Once the repository has been downloaded, PyCharm may ask you whether you trust this project. After making sure that you indeed downloaded the examples of this book (and if you deem this code trustworthy), you can click Trust Project, as Figure 2.12.4. Finally, as Figure 2.12.5 shows, you can now see and play with and run all the examples in PyCharm. A more comprehensive discussion on how to clone repositories is given in Section 15.1.



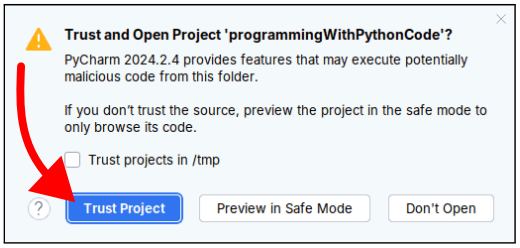
(2.12.1) Click Clone Repository in the PyCharm welcome screen.



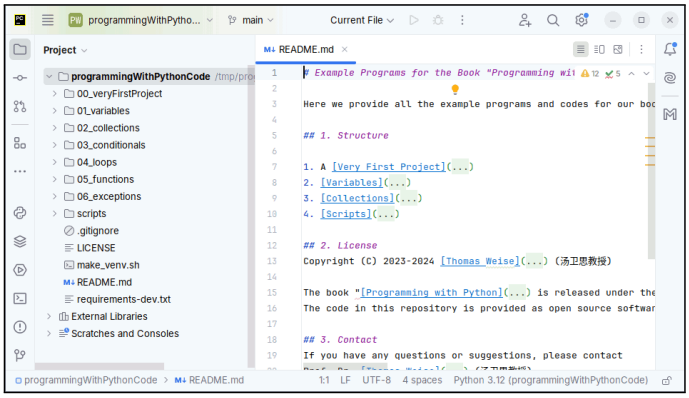
(2.12.2) Selecting URL: https://github.com/thomasWeise/programmingWithPythonCode and a reasonable destination Directory: in the next dialog, then click Clone.



(2.12.3) Wait while PyCharm is cloning (downloading) the repository.



(2.12.4) If asked, click Trust Project after confirming that you indeed downloaded the right code and if you trust our code.



(2.12.5) The project with all the examples from this book is now downloaded and can be accessed in PyCharm.

Figure 2.12: Using PyCharm to clone (download) and import all the examples from this book.

2.6 Summary

In this introductory section, we have performed the very first steps into the domain of Python programming. We have now a computer where both Python and the PyCharm IDE are installed. We can create program files and we can execute them in different ways. We also obtained the set of example programs that will later be used in this book. We are now ready to learn how to program.

Chapter 3

Simple Datatypes and Operations

3.1 Introduction

We now know how to create and run Python programs, both in the IDE and terminal. We have also already learned our first two Python commands:

- `print("Hello World!")` prints the text "Hello World!" to the output.
- `exit()` exits and terminates the Python interpreter.

Now, it would be very strange if the `print` function could *only* print "Hello World!". That would not make much sense. `print` expects one parameter. This parameter should be a text string[1].

The command `exit`, on the other hand, can either have no parameter or one parameter. If it receives one parameter, this parameter will be the exit code of the program. Here, `0` indicates success. If no parameter is provided, this will be used as default value.

We realize: Distinguishing different types of data makes sense. Sometimes we need to do something with text. Sometimes we want to do something with numbers. Sometimes, we want to just handle a decision which can be either "yes" or "no".

Of course, for these different situations, different possible operations may be useful. For example, when we use numbers, we may want to divide or multiply them. When we handle text, we may want to concatenate two portions of text, or maybe we want to convert lowercase characters to uppercase. We may want to do something if two decision variables are both "yes" or, maybe, if at least one of them is.

In this chapter, we will look into the simple datatypes of Python, namely:

- `int`: the integer datatype, which represents integers numbers $\mathbb{Z}$ (Section 3.2),
- `float`: the floating point numbers, i.e., a subset of the real numbers $\mathbb{R}$ (Section 3.3),
- `bool`: Boolean values, which can be either `True` or `False` (Section 3.5),
- `str`: strings, i.e., portions of text of arbitrary length (Section 3.6), and
- `None`: nothing, which is the result of any command that does not explicitly return a value (Section 3.7).

Here, we will not yet really write Python programs. Instead, we will use the Python interpreter console more like a fancy calculator. This will allow us to explore the basic functionality regarding the above-mentioned datatypes efficiently and freely.

3.2 Integers

Integer arithmetic is the very first thing that you learn in mathematics in primary school. Integer arithmetic is also the very first thing you learn here. *Integer* is a Latin word that means "whole" or "intact." The integers include all whole numbers and negative numbers and zero, without fractions and decimals.

[1] Or it needs to support a certain method so that it can be comverted to a text, but please let's ignore this for now.

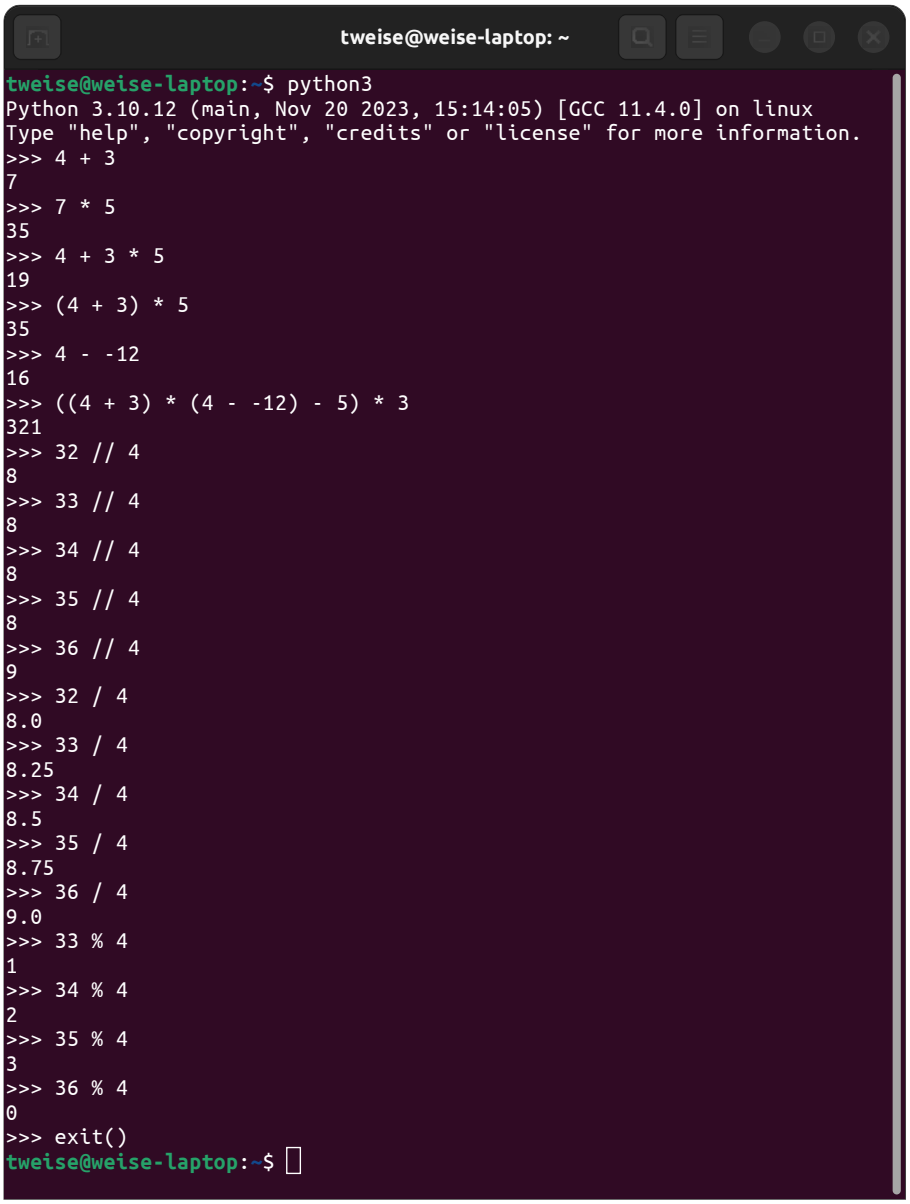


Figure 3.1: Examples of Python integer math in the console, part 1 (see Listing 3.1 for part 2).

In many programming languages, there are different integer datatypes with different ranges. In Java, a `byte` is an integer datatype with range $-2^7..2^7-1$, a `short` has range $-2^{15}..2^{17}-1$, an `int` has range $-2^{31}..2^{31}-1$, and `long` has range $-2^{63}..2^{63}-1$, for example. The draft for the C17 standard for the C programming language lists five signed and five unsigned integer types, plus several ways to extend them [459]. The different integer types of both languages have different ranges and sizes, and the programmer must carefully choose which she needs to use in which situation.

Python 3 only has one integer type, called `int`. This type has basically an unbounded range. The Python 3 interpreter will allocate as much memory as is needed to store the number you want.[2]

3.2.1 Integer Arithmetic

Now, what can we do with integer numbers? We can add, subtract, multiply, divide, modulo divide, and raise them to powers, for example.

In Figure 3.1, you can find some examples of this. (The same example is given in Listing 3.1, just as listing instead of screenshot. We will use such listings from now on, as they convey the exactly

[2] Ok, the range is not actually *unbounded*, it is bounded by the amount of memory available on your computer... ...but for all intents and purposes within this book, we can assume that `int` $\equiv \mathbb{Z}$.

Listing 3.1: The same examples of Python integer math in the Python console as given in Figure 3.1, just as listing instead of screenshot.

```
Python 3.14.10 (main, Oct 27 2026, 09:35:01) [GCC 15.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 4 + 3          # Adding 4 + 3 gives 7.
7
>>> 7 * 5          # Multiplying 7 by 5 gives 35.
35
>>> 4 + 3 * 5      # This is equivalent to 4 + (3 * 5) = 19.
19
>>> (4 + 3) * 5    # (4 + 3) * 5 = 7 * 5 = 35
35
>>> 4 - -12        # 4 - (-12) = 4 + 12 = 16
16
>>> ((4 + 3) * (4 - -12) - 5) * 3  # = (7 * 16 - 5) * 3 = 107 * 3 = 321
321

>>> 32 // 4        # integer division: 32 // 4 = 8 (remainder would be 0)
8
>>> 33 // 4        # integer division: 33 // 4 = 8 (remainder would be 1)
8
>>> 34 // 4        # integer division: 34 // 4 = 8 (remainder would be 2)
8
>>> 35 // 4        # integer division: 35 // 4 = 8 (remainder would be 3)
8
>>> 36 // 4        # integer division: 36 // 4 = 9 (remainder would be 0)
9

>>> 32 / 4         # fractional division: 32 / 4 = 8.0
8.0
>>> 33 / 4         # fractional division: 33 / 4 = 8.25
8.25
>>> 34 / 4         # fractional division: 34 / 4 = 8.5
8.5
>>> 35 / 4         # fractional division: 35 / 4 = 8.75
8.75
>>> 36 / 4         # fractional division: 36 / 4 = 9.0
9.0

>>> 33 % 4         # remainder of integer division 33 by 4 is 1
1
>>> 34 % 4         # remainder of integer division 34 by 4 is 2
2
>>> 35 % 4         # remainder of integer division 35 by 4 is 3
3
>>> 36 % 4         # remainder of integer division 36 by 4 is 0
0

>>> exit()         # exit the interactive interpreter
```

same information, but are easier to read and I can more conveniently include comments.) Like back in Section 2.4.3, press Ctrl+Alt+T under Ubuntu Linux or press ⊞+R, type in `cmd`, and hit ↵ under Microsoft Windows to open a terminal. After entering `python3` and hitting ↵, we can begin experimenting with integer maths. The lines with Python commands in the console begin with `>>>`, whereas the result is directly output below them without prefix string.

In the very first line of Figure 3.1 and Listing 3.1, we enter `4 + 3` and hit ↵. The result is displayed on the next line and, as expected, is `7`. We then attempt to multiply the two integers `7` and `5` by typing `7 * 5` and hitting ↵. The result is `35`.

Python does not just support normal arithmetics as you have learned it in school, it also follows the

Listing 3.2: Examples of Python integer math (powers) in the Python console, part 2 (see Listing 3.1 for part 1).

```
Python 3.14.10 (main, Oct 27 2026, 09:35:01) [GCC 15.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 2 ** 7          # 2 raised to the 7th power, i.e., 2 by 7.
128
>>> 7 ** 11         # 7 raised to the 11th power, i.e., 7 by 11.
1977326743
>>> 2 ** 63         # 2 by 63.
9223372036854775808
>>> 2 ** 64         # 2 by 64.
18446744073709551616
>>> 2 ** 1024       # 2 by 1024.
17976931348623159077293051907890247336179769789423065727343008115773267580
    ↪ 50096313270847732240753602112011387987139335765878976881441662249284
    ↪ 43063947412437776789342486548527630221960124609411945308295208500576
    ↪ 83815068234246288147391311054082723716335051068458629823994724593847
    ↪ 9163048353563296242241372 16
```

operator precedence rules. If we type in `4 + 3 * 5`, it will compute $4 + (3 * 5) = 4 + 15 = 19$ and, hence, print `19`. We can also use parentheses and type in `(4 + 3)* 5`, which will be evaluated as, well $(4 + 3) * 5 = 7 * 5 = 35$, and we get `35`. Integers can be signed, so typing `4 - -12` yields Python16. Parentheses can be arbitrarily nested, so we can also compute `((4 + 3)* (4 - -12)- 5)* 3`, which evaluates to $((7 * 16) - 5) * 3 = (112 - 5) * 3 = 107 * 3 = 321$.

Division is a bit tricky in programming in general and in Python as well. There are *two kinds* of division in Python: Integer division, denoted by `//` and fractional division, denoted as `/`.

`32 // 4` yields `8`, because `4` fits `8` times into `32`. `33 // 4`, `34 // 4`, and `35 // 4` all still yield `8`, as `4` *completely* fits `8` times into these numbers (leaving some remainder left over). `36 // 4` then finally yields `9`. The results of the integer division operator `//` are always also `ints`.

Fractional division with `/`, however, returns `float` values, which we will explore in the next section (Section 3.3) in detail. For now, let's just say that they can represent fractional parts (to a limited precision), which is denoted by having a Python. in the text output of the numbers. Computing `32 / 4` thus yields `8.0`, `33 / 4` gives us `8.25`, `34 / 4` yields `8.5`, `35 / 4` results in `8.75`, and, finally, `36 / 4` returns `9.0`. Notice that the result of this division operator is always a floating point number, even if the number itself is an integer.[3]

Best Practice 2

Always be careful with which division operator you use for `ints`. If you need an integer result, make sure to use `//`. Remember that `/` always returns a `float` (and see Best Practice 3), even if the result is a whole number.

Now above we have said that `33 // 4` yields the integer `8`. The remainder of this operation can be computed using the modulo division operator `%`, i.e., by typing `33 % 4`, which yields `1`. We also find that `34 % 4` yields `2`, `35 % 4` gives us `3`, and `36 % 4` is `0`.

As you will find in Listing 3.1, integers can also be raised to a power. For example, 2^7 is expressed as `2 ** 7` in Python (and yields `128`). `7 ** 11`, i.e., 7^{11} gives us 1 977 326 743 and shows as `1977326743` in the output.

In many programming languages such as Java and C, the largest integer type available off the shelf is 64 bits wide. If it is signed (can have negative values) like Java's `long`, it has range $-2^{63}..2^{63} - 1$. An unsigned 64 bit integer type, such as `unsigned long long` in C, would have range $0..2^{64} - 1$. In Python we can compute 2^{63} (`2 ** 63`), namely 9 223 372 036 854 775 808, and 2^{64} (`2 ** 64`), which is 18 446 744 073 709 551 616. These are very large numbers and the latter one would overflow the range of the standard integer types of Java and C. However, we can also keep going and compute

[3] Notice also that this behavior is introduced in Python 3, whereas `/` behaves like `//` in Python 2 [651].

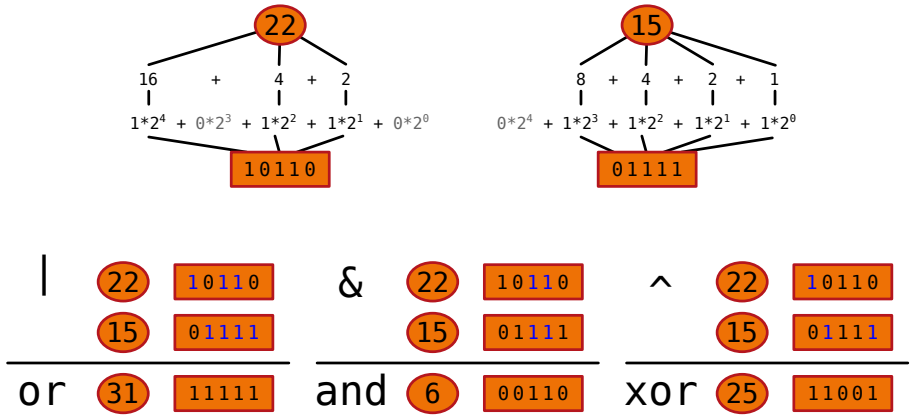


Figure 3.2: Examples for how integer numbers are represented as bit strings in the computer (upper part) and for the binary (bitwise) operations *and*, *or*, and *exclusive or* (often called *xor*).

`2 ** 1024`, which is such a huge number that it wraps several times in the output of our `Python` console in Listing 3.1! `Python` integers are basically unbounded (or bounded only by the memory size of our computer). However, the larger they get, the more memory they need and the longer it will take to compute with them.

3.2.2 Operations on Bit Strings

First Time Readers and Novices: You are encouraged to skip over this section. This section is about integer numbers from the perspective of bit strings and the bit string based operations that we can apply to them. If you are learning programming, then this part is not important now. You can circle back to it later.

All integer numbers can be represented as bit strings. In other words, a number $z \in \mathbb{Z}$ can be expressed as $b_0 2^0 + b_1 2^1 + b_2 2^2 + b_3 2^3 + b_4 2^4 \ldots$, where the b_i-values each are either 0 or 1. Then, $\ldots b_4 b_3 b_2 b_1 b_0$ is a bit string. If we represent integers with such strings of five bits, then the number 1 would have representation 00001, because it is equivalent to 2^0. In in Figure 3.2, we illustrate that the number 22 would be 10110 because $22 = 2^4 + 2^2 + 2^1$ and the number 15 would correspond to 01111, as $15 = 2^3 + 2^2 + 2^1 + 2^0$.

We can obtain the binary representation of integer numbers as text using the `bin` function in `Python`. As shown in Listing 3.3, `bin(22)` yields `'0b10110'`. Here, the `0b` prefix means that the following number is in binary representation. We can also enter numbers in binary representation in the console. Typing `0b10110` corresponds to the number `22`. Similarly, `bin(15)` yields `'0b1111'` and entering `0b1111` into the console corresponds to entering the number `15`.

In `Python`, we can compute the bitwise (i.e., binary) *or*, *and*, as well as the and *exclusive or* of this binary representation of integers using the `|`, `&`, and `^` operators, respectively. Binary *or* returns an integer in which all bits are set to 1 which were 1 in either of its two operands. `22 | 1` yields `23`, because the bit with value 1 is not set in 22 and the binary *or* sets it (effectively adding 1 to 22). The slightly more comprehensive example `22 | 15` sketched in Figure 3.2 gives us `31`, because $22 = 2^4 + 2^2 + 2^1$ and $15 = 2^3 + 2^2 + 2^1 + 2^0$, which `|` combines to $31 = 2^4 + 2^3 + 2^2 + 2^1 + 2^0$, i.e., each power of 2 that occurred in either of the two operands is present in the result. `bin(31)` yields `'0b11111'`.

Binary *and* returns the integer where only the bits remain 1 that were 1 in *both* operands. Applying binary *and* instead of *or*, i.e., doing `22 & 1` results in `0`, because, as said before, the bit 2^0 is not set in 22. `22 & 15`, yields `6`, because only 2^2 and 2^1 appear both in 22 and 15. Thus, `bin(6)` corresponds to `'0b110'`.

The *exclusive or*, which is often called *xor*, will set a bit to 1 if it is 1 in exactly one of the two operands. Therefore, `22 ^ 1` gives `23`, since only the bit with value 2^0 is set in 1 and the other bits that are 1 in 22 are not. `22 ^ 15` yields `25`, because 2^4, 2^3, and 2^0 occur only once in the two operators (whereas 2^2 and 2^1 occured in both of them). This is confirmed by typing `bin(25)`, which results in `'0b11001'`.

Listing 3.3: Examples of the binary representation of integers and the operations that apply to it.

```
Python 3.14.10 (main, Oct 27 2026, 09:35:01) [GCC 15.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> bin(22)        # Convert 22 to a text string with bit values.
'0b10110'
>>> 0b10110        # 22 in binary notation, indicated by prefix 0b.
22
>>> bin(15)        # Convert 15 to a text string with bit values.
'0b1111'
>>> 0b1111         # 15 in binary notation, indicated by prefix 0b.
15

>>> 22 | 15        # bit-wise 'or' of 0b10110 and 0b1111
31
>>> bin(31)        # Convert 31 to a text string with bit values.
'0b11111'

>>> 22 & 15        # bit-wise 'and' of 0b10110 and 0b1111
6
>>> bin(6)         # Convert 6 to a text string with bit values.
'0b110'

>>> 22 ^ 15        # bit-wise 'exclusive or' of 0b10110 and 0b1111
25
>>> bin(25)        # Convert 25 to a text string with bit values.
'0b11001'

>>> 22 << 1        # Shift 22 left by 1 step == 22 * 2.
44
>>> bin(44)        # Convert 44 to a text string with bit values.
'0b101100'

>>> 22 >> 2        # Shift 22 right by 2 steps == 22 // 4
5
>>> bin(5)         # Convert 5 to a text string with bit values.
'0b101'

>>> hex(22)        # Convert 22 to a string in hexadecimal notation.
'0x16'
>>> 0x16           # 22 in hexadecimal notation, indicated by prefix 0x.
22

>>> oct(22)        # Convert 22 to a string in octal notation.
'0o26'
>>> 0o26           # 22 in octal notation, indicated by prefix 0o.
22
```

Finally, we can also shift bit strings to the left or right by i places. The former corresponds to multiplying with 2^i, the latter is the same as an integer division by 2^i. Shifting `22` by one bit position to the *left* – which is done by entering `22 << 1` – therefore results in `44`. We already know that `bin(22)` is `'0b10110'` and so it comes as no surprise that `bin(44)` is `'0b101100'` (notice the additional `0` that appeared on the right hand side). Shifting `22` by two bit positions to the *right* – which is done by entering `22 >> 2` – results in `5`. The `10` on the right hand side of the binary representation disappeared, as `bin(5)` is `'0b101'`.

Besides the binary representation of integer numbers, which is to the base 2, there also exists the hexadecimal representation (base 16) and the octal representation (base 7). We can obtain the hexadecimal representation of 22 by computing `hex(22)` and get `0x16`, which corresponds to $1 * 16^1 + 6 * 1 = 22$. We can also enter hexadecimal numbers in the console like `0x16`, which yields `22`. The octal representation of 22 is obtained as `oct(22)`, which produces `0o26`, which, in turn, corresponds

Table 3.1: A table with number conversions to the decimal (dec), binary (bin), octal (oct), and hexadecimal (hex) systems. The values of the digits are noted in the second row of the header. Powers of the bases are highlighted.

dec		bin							oct			hex		dec		bin							oct			hex	
10	1	64	32	16	8	4	2	1	64	8	1	16	1	10	1	64	32	16	8	4	2	1	64	8	1	16	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1	0	0	1	0	1
0	2	0	0	0	0	0	1	0	0	0	2	0	2	0	3	0	0	0	0	0	1	1	0	0	3	0	3
0	4	0	0	0	0	1	0	0	0	0	4	0	4	0	5	0	0	0	0	1	0	1	0	0	5	0	5
0	6	0	0	0	0	1	1	0	0	0	6	0	6	0	7	0	0	0	0	1	1	1	0	0	7	0	7
0	8	0	0	0	1	0	0	0	0	1	0	0	8	0	9	0	0	0	1	0	0	1	0	1	1	0	9
1	0	0	0	0	1	0	1	0	0	1	2	0	a	1	1	0	0	0	1	0	1	1	0	1	3	0	b
1	2	0	0	0	1	1	0	0	0	1	4	0	c	1	3	0	0	0	1	1	0	1	0	1	5	0	d
1	4	0	0	0	1	1	1	0	0	1	6	0	e	1	5	0	0	0	1	1	1	1	0	1	7	0	f
1	6	0	0	1	0	0	0	0	0	2	0	1	0	1	7	0	0	1	0	0	0	1	0	2	1	1	1
1	8	0	0	1	0	0	1	0	0	2	2	1	2	1	9	0	0	1	0	0	1	1	0	2	3	1	3
2	0	0	0	1	0	1	0	0	0	2	4	1	4	2	1	0	0	1	0	1	0	1	0	2	5	1	5
2	2	0	0	1	0	1	1	0	0	2	6	1	6	2	3	0	0	1	0	1	1	1	0	2	7	1	7
2	4	0	0	1	1	0	0	0	0	3	0	1	8	2	5	0	0	1	1	0	0	1	0	3	1	1	9
2	6	0	0	1	1	0	1	0	0	3	2	1	a	2	7	0	0	1	1	0	1	1	0	3	3	1	b
2	8	0	0	1	1	1	0	0	0	3	4	1	c	2	9	0	0	1	1	1	0	1	0	3	5	1	d
3	0	0	0	1	1	1	1	0	0	3	6	1	e	3	1	0	0	1	1	1	1	1	0	3	7	1	f
3	2	0	1	0	0	0	0	0	0	4	0	2	0	3	3	0	1	0	0	0	0	1	0	4	1	2	1
3	4	0	1	0	0	0	1	0	0	4	2	2	2	3	5	0	1	0	0	0	1	1	0	4	3	2	3
3	6	0	1	0	0	1	0	0	0	4	4	2	4	3	7	0	1	0	0	1	0	1	0	4	5	2	5
3	8	0	1	0	0	1	1	0	0	4	6	2	6	3	9	0	1	0	0	1	1	1	0	4	7	2	7
4	0	0	1	0	1	0	0	0	0	5	0	2	8	4	1	0	1	0	1	0	0	1	0	5	1	2	9
4	2	0	1	0	1	0	1	0	0	5	2	2	a	4	3	0	1	0	1	0	1	1	0	5	3	2	b
4	4	0	1	0	1	1	0	0	0	5	4	2	c	4	5	0	1	0	1	1	0	1	0	5	5	2	d
4	6	0	1	0	1	1	1	0	0	5	6	2	e	4	7	0	1	0	1	1	1	1	0	5	7	2	f
4	8	0	1	1	0	0	0	0	0	6	0	3	0	4	9	0	1	1	0	0	0	1	0	6	1	3	1
5	0	0	1	1	0	0	1	0	0	6	2	3	2	5	1	0	1	1	0	0	1	1	0	6	3	3	3
5	2	0	1	1	0	1	0	0	0	6	4	3	4	5	3	0	1	1	0	1	0	1	0	6	5	3	5
5	4	0	1	1	0	1	1	0	0	6	6	3	6	5	5	0	1	1	0	1	1	1	0	6	7	3	7
5	6	0	1	1	1	0	0	0	0	7	0	3	8	5	7	0	1	1	1	0	0	1	0	7	1	3	9
5	8	0	1	1	1	0	1	0	0	7	2	3	a	5	9	0	1	1	1	0	1	1	0	7	3	3	b
6	0	0	1	1	1	1	0	0	0	7	4	3	c	6	1	0	1	1	1	1	0	1	0	7	5	3	d
6	2	0	1	1	1	1	1	0	0	7	6	3	e	6	3	0	1	1	1	1	1	1	0	7	7	3	f
6	4	1	0	0	0	0	0	0	1	0	0	4	0	6	5	1	0	0	0	0	0	1	1	0	1	4	1

to $2 * 8^1 + 6$. Similarly, this octal number can be entered as `0o26`. As a refresher on these numerical systems, we include Table 3.1, which illustrates decimal, binary, octal, and hexdecimal notations of the first couple of natural numbers.

You may wonder: "How does an `int` know whether it was entered in hexadecimal, octal, binary, or decimal notation?" It does not. It does not matter at all in which format an integer value is entered in the computer or in your program code. These are just text formats to store numbers (either in your program code, the interpreter, or the input of your program). These text formats get converted to `int` structures which do not remember in any way how the numbers were originally entered. These formats only exist for input and output. Depending on your situation, you can use whatever format is most convenient in your situations.

With this, our excursion into binary maths ends and we now welcome back all first-time readers.

3.2.3 Summary

In conclusion, the integer type `int` represents whole numbers. Integers can be positive or negative or zero. All the primitive mathematical operations like addition, subtraction, multiplication, and division that you learned in school can be applied to integers. The normal arithmetic precedence rules (that you also have learned in school) apply. Parentheses can be used to group operations. Finally, Python also provides the same binary logic operators, working on the bit string representation of integers, that you may or or may not know from other programming languages as well.

3.3 Floating Point Numbers

In the previous section, we have discussed integers in Python. One of the very nice features of the Python 3 language is that integers can basically become arbitrarily large. There is only the single type `int` and it can store any integer value, as long as the memory of our computer is large enough.

In an ideal world, we would have a similar feature also for real numbers. However, such a thing cannot be practically implemented. You will certainly remember irrational numbers like $\sqrt{2}$, $\pi \approx 3.141\,592\,653\,590\ldots$ and $e \approx 2.718\,281\,828\,459\ldots$ from high school maths. Their fractional digits never end, nobody has yet detected an orderly pattern in them. Since these numbers are "infinitely long," we would require infinitely much memory to store them *if* we wanted to represent them *exactly*. So we don't and neither does Python. We cannot really represent the real numbers $\mathbb{R}$ exactly in the memory of our computers.

3.3.1 How Floating Point Numbers Work

But how does it work in Python? How can we deal with the fact that we cannot represent arbitrary fractional numbers exactly even in typical everyday cases like π and e? How can we deal with the situation that real numbers exist as big as 10^{300} and as small as 10^{-300}? With `float`, Python offers us one type for fractional numbers. This datatype represents numbers usually in the same internal structure as `doubles` in the C programming language [417] – it is based on the 64 bit IEEE Standard 754 floating point number layout [227, 267, 282]. The idea behind this standard is exactly to be able to represent both very large numbers, like 10^{300} and very small numbers, like 10^{-300}, while accepting that we cannot exactly represent $10^{300} + 10^{-300}$. In order to achieve this, the 64 bits are divided into three pieces, as illustrated in Figure 3.3.

First Time Readers and Novices: You just need to understand that `floats` have limited precision. You can jump right to the next section.

The first part, the so-called significand or mantissa, consists of 52 bits, represents the digits of the number. 52 bits can represent $52 \log_2 10 \approx 15$ to 16 decimal digits, meaning that we can represent numbers to a precision of about 15 digits. If we would just use 52 bits, then this would limit us to represent numbers maybe from 0 to $2^{52} - 1$ at a resolution of 1. Of course, we could also choose some other resolution, say 0.001. In this case, we could represent numbers from 0 to $0.001 * (2^{52} - 1)$ and the smallest non-zero number would be 0.001 instead of 1. Whatever fixed resolution we would choose, it would be good in some cases and bad in others.

Therefore, the second part of the 64 bit floating point number representation comes into play: The 11 bits of the exponent store the resolution. They represent a power of 2 which is multiplied to the significand to get the actual number. In order to allow us to have both small and large numbers, this value must be able represent positive and negative exponents. Therefore, the stored value of the exponent is taken and a constant bias of 1023 is subtracted. Thus, a stored value of 1050 in the exponent fields leads to an actual exponent of $1050 - 1023 = 27$, which would mean that the significand is multiplied with 2^{27}, i.e., 134 217 728.



Figure 3.3: The structure of an 64 bit / double precision IEEE Standard 754 floating point number [267, 282].

Finally, the sign bit in the floating point number dataformat indicates whether the number is positive or negative. Together, this allows us to represent numbers from $2.225\,073\,858\,507\,201\,4 * 10^{-308}$ to $1.797\,693\,134\,8623\,157 * 10^{308}$ with a resolution of approximately 15 digits. Of course, the same range also applies to negative numbers and 0 can be represented as well. Indeed, there are even special floating point values for infinity and errors. But more about this later.

Luckily, you will never really need to know these exact information in your day-to-day programming work. There also exist many different formats of floating point numbers, using different numbers of bits for significand and exponent [227, 282]. The important thing to remember is: Floating point numbers (`floats`) can represent a wide range of different values. Their range is large but still limited. They can represent integers and fractions. However, their accuracy is always limited to about 15 digits. In other words, regardless whether your `float` stores a very large or a very small number, you can have at most 15 digits of precision. For example, adding 1 to 10^{16} would still yield 10^{16}, because only 15 digits are "stored" and the 1 will just "fall off." You cannot represent numbers arbitrarily precisely with `floats` [204]. Except for that, `floats` are pretty cool, though.

3.3.2 Floating Point Arithmetic

Floating point numbers in `Python` can be distinguished from `ints` by having a decimal dot in their text representation, i.e., `5.0` is a `float` and `5` is an `int`. Let us now look at some examples of the basic arithmetic operations available for `floats` in Listing 3.4.

We already learned that the division operator `/` always yields a `float` result. Therefore `6 / 3` yields `2.0` instead of `2`.

The normal arithmetic operations like addition, subtraction, multiplication, division, and powers all work with `floats` as expected. Remember, however, that if `float` and `int` numbers are mixed, the results are always `floats`. Thus, `1.0 + 7` gives us `8.0` and `2 * 3.0` yields `6.0`. In other words, if one `float` occurs somewhere in an expression, it will "infect" everything that it touches to become a `float` too, even if the result could be represented as `int`. Some results cannot be integers anyway, for example `5 - 3.6` evaluates to `1.4`. The remainder (the modulo) of a division can also be computed for floating point numbers. The remainder of the division of 6.5 by 2, i.e., `6.5 %2` is `0.5`.

The square root of 3.3 can be computed as $3.3^{0.5}$. We can write this as `3.3 ** 0.5`, which yields `1.816590212458495`. This brings us back to the previous section: $\sqrt{3.3}$ is not actually 1.816 590 212 458 495. It is an irrational number [47, 489] and irrational numbers cannot be expressed as fractions of integer numbers (by definition, otherwise they would be rational numbers). Since they cannot be expressed as integer fractions, we cannot write them down exactly in any way like $\frac{1\,816\,590\,212\,458\,495\ldots}{1\,000\,000\,000\,000\,000\ldots}$, regardless of how large a denominator we would pick. Hence, we cannot represent them exactly using discrete binary values of our computer's memory. Hence, the floating point representation cuts off somewhere. And this somewhere is after 15 decimal places.

We can of course also write and compute more complex mathematical expressions. `((3.4 * 5.5)- 1.2)** (4.4 / 3.3)` corresponds to $((3.4*5.5)-1.2)^{\frac{4.4}{3.3}}$ and yields `45.43432339119718`. This is again not an exact value but a rounded value. We always need to keep this in mind.

Let us recall our initial example of the transcendental numbers π and e. Certainly, these are very important constants that would be used in many computations. We can make them accessible in our code by importing them from the `math` module.[4] This can be done by typing `from math import pi, e`. When we then type `pi` and `e`, we can get to see their value in floating point representations: `3.141592653589793` and `2.718281828459045`, respectively. Again, these are not the exact values, but they are as close as we can get in this format.

Surprisingly, we do even need irrational let alone transcendental numbers to experience this cut-off. There is no way to write down all the digits of fractions like $\frac{1}{7}$ as decimals. We always need to cut off somewhere, e.g., we could write 0.14285714285714285, but that is not exactly the same as $\frac{1}{7}$. As we discussed before, floating point numbers are stored in a binary format, i.e., represented by bits. In the binary system, we encounter this problem already for fractions like $\frac{1}{10} = 0.1$ [204]. Essentially, we could write $\frac{1}{10} \approx \frac{1}{2^4} + \frac{1}{2^5} + \frac{1}{2^8} + \frac{1}{2^9} + \frac{1}{2^{12}} + \frac{1}{2^{13}} + \frac{1}{2^{16}} + \ldots$, but we would never quite get there. This means that 0.1 cannot be exactly represented as `float`. Therefore, adding it up ten times also does not exactly equal 1. Indeed, adding `0.1` ten times and then subtracting `1.0` from the result yields `-1.1102230246251565e-16`. So even for "completely normal" numbers, floating point arithmetics may already cost us (a very tiny little bit of) precision.

[4] We will learn about these mechanism in detail later on.

Listing 3.4: Basic arithmetic with floating point numbers in Python.

```
Python 3.14.10 (main, Oct 27 2026, 09:35:01) [GCC 15.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 6 / 3          # Floating-point division 6 / 3 = 2.0
2.0
>>> 1.0 + 7        # float + int = float
8.0
>>> 5 - 3.6        # int - float = float
1.4
>>> 2 * 3.0        # int * float = float
6.0
>>> 6.5 % 2        # float % int = float
0.5
>>> 3.3 ** 0.5     # square root of 3.3
1.816590212458495
>>> ((3.4 * 5.5) - 1.2) ** (4.4 / 3.3)
45.43432339119718

>>> from math import pi, e   # import some mathematical constants
>>> pi
3.141592653589793
>>> e              # Euler's Number
2.718281828459045

# 0.1 cannot be exactly represented in binary float notation.
# Therefore, the sum of ten '0.1's does not add up to 1.0.
>>> 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1
# The difference is small, but it is there.
0.9999999999999999
>>> 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 - 1.0
-1.1102230246251565e-16

>>> from math import sin, cos, tan, log  # trigonometric operators
>>> sin(0.25 * pi) ** 2   # This gives almost 0.5.
0.4999999999999999
>>> cos(pi / 3)           # This gives almost 0.5, too.
0.5000000000000001
>>> tan(pi / 4)           # And this is almost 1.
0.9999999999999999
>>> log(e ** 10)          # Sometimes, we get exact results, though.
10.0

>>> from math import asin, acos, atan    # inverse trigonometric ops
>>> asin(sin(0.925))      # Often, we cannot exactly invert operations.
0.9250000000000002
>>> acos(cos(-0.3))       # and get results which are close to what we
0.30000000000000016
>>> atan(tan(1))          # expect. But sometimes we get exact results.
1.0
```

Anyway, back to the constants π and e. Alone, they are not that much useful, but if you reach back into your high school days again, you will remember many interesting functions that are related to them. Let us import a few of them, again from the `math` module, via `from math import sin, cos, tan, log`. I think you can guess what these functions do.

From high school, you may remember that $\sin\frac{\pi}{4} = \frac{\sqrt{2}}{2}$ and thus $\sin^2\frac{\pi}{4} = 0.5$. Let us compute this in Python by doing `sin(0.25 * pi)** 2`. Surprisingly, we get `0.4999999999999999` instead of `0.5`. The reason is again the limited precision of `float`, which cannot represent $\frac{\sqrt{2}}{2}$ exactly. Similarly, $\cos\frac{\pi}{3} = \frac{1}{2}$ but `cos(pi / 3)` yields `0.5000000000000001` and $\tan\frac{\pi}{4}$ expressed as `tan(pi / 4)` returns `0.9999999999999999` instead of 1. Then again, these values are incredibly close to the exact results.

They are off by *less than* 10^{-15} so for all practical concerns, they are close enough. Sometimes, we even get the accurate result, e.g., when computing $\ln(e^{10})$ by evaluating `log(e ** 10)`, which results in `10.0`.

As a final example of floating point arithmetics, let us import the inverse trigonometric functions by doing `from math import asin, acos, atan`. Obviously, $\arcsin\sin 0.925$ should be 0.925, since $\arcsin\sin x = x\ \forall x \in [-\frac{\pi}{2}, \frac{\pi}{2}]$. Calculating `asin(sin(0.925))` indeed yields `0.9250000000000002`. It also holds that $\arccos\cos x = x\ \forall x \in [0, \pi]$. Due to the periodicity of the trigonometric functions, $\arccos\cos -0.3$ is 0.3 and `acos(cos(-0.3))` results in `0.30000000000000016`. For $\arctan\tan 1$ we even get the exact result `1.0` by computing `atan(tan(1))`.

Best Practice 3

Always assume that any `float` value is imprecise. Never expect it to be exact [38, 455].

Due to the limited precision, it could be that you add two numbers $a + b = c$ but then find that $c - a \neq b$, because some digit was lost. This is obvious when adding a very small number to a very large number. We only have about 15 digits, so doing something like $10^{20} + 1$ will usually work out to just be 10^{20} in floating point arithmetics [455]. But digits could also be lost when adding numbers of roughly the same scale, because their sum could just be larger so that the 15-digit-window shifts such that the least-significant digit falls off [38]...

3.3.3 Back to Integers: Rounding

We already learned that a single `float` value inside a computation that otherwise uses `ints` basically forces the whole result to become a `float` as well. But maybe sometimes we want to have an `int` as result. Therefore, we need a "way back" from `float` to `int`. Python offers several functions for this.

The first and maybe most common one is `round`. This function accepts a `float` and rounds it to the nearest integer. If two integer numbers are equally close to it, it returns the one that is even [86]. This rounding method is called *banker's rounding* [440]. It is defined in the IEEE 754 standard for floating point arithmetics [282] and used in many applications today, including Alipay+ [25].

This rounding behavior can be very surprising, as we learned in school that numbers of the form $x.5$ should be rounded to $x + 1$. Well, I learned it like this in Germany (and Java's `Math.round` also works like that[297]), but elsewhere it may be taught differently... Under banker's rounding, this will only happen with `round` if $x + 1$ is even. The reason why banker's rounding is preferred is that it is not biased in any direction [271, 383]. If $x.5$ is always rounded toward $x + 1$ and we have many numbers of the form $x.5$, then the average rounded result tends to be somewhat larger than the average of the unrounded numbers. Banker's rounding avoids this problem by sometimes rounding numbers ending in 5 to larger and sometimes to smaller values.

Let's look at the following example. Assume we have a list S of numbers such that $S = (0.5, 1.5, 2.5, 3.5, 4.5, 5.5)$. Traditional "school rounding," which rounds up numbers ending in .5, would give us $S_s = (1, 2, 3, 4, 5, 6)$, as all numbers are rounded up since they all end in 0.5. Banker's rounding as practiced by Python yields $S_b = (0, 2, 2, 4, 4, 6)$. It will round down 0.5, 2.5, and 4.5, and it rounds up 1.5, 3.5, and 5.5. Let's look at the sums of all the values in the lists. For the original list S, we get $\sum_{\forall s \in S} s = 18$. For the school-rounding method, we get $\sum_{\forall s_s \in S_s} s_s = 21$. For the banker's rounding method, we get $\sum_{\forall s_b \in S_b} s_b = 18$. We find that, with banker's rounding, the sum of the rounded values is closer to the sum of the original values. This would also hold for the arithmetic means mean. This, of course, only plays a role if many values ending in .5 appear. But it is quite clear why bankers would like that. If you work in a bank and shuffle around large sums of money, you certainly would like these sums to add up properly...

We find examples of the behavior of `round` in Listing 3.5. `round(0.4)` yields the integer `0`, as expected. `round(0.5)` returns `0` as well, which one may not expect – but `0` is even and `1` would be odd. `round(0.6)` gives us the integer `1`. If we compute `round(1.4)`, we still get `1`. However, doing `round(1.5)` gives us `2`. This is because `2` is even.

Three more common functions to turn `floats` to `ints` are given in the `math` module. We can import them via `from math import floor, trunc, ceil`. (Again, we will learn later how `import` actually works. For now, just accept that it makes some functions available.)

`floor` rounds the `float` it receives to the nearest integer which is less than or equal to. `floor(0.4)`, `floor(0.5)`, and `floor(0.6)` all yield `0`. `floor(-0.4)` gives us `-1`.

`trunc` just discards any fractional digits and only returns the integer part of a number. Hence, `trunc(-0.6)`, `trunc(-0.4)`, `trunc(0.4)`, and `trunc(0.6)` all yield `0`.

Finally, `ceil` rounds to the nearest integer which is greater than or equal to its argument. `ceil(0.4)` yields `1` and `-0.4` results in `0`. `ceil(-11.1)` and `ceil(-11.6)` both result in `11`. `ceil(11.6)` gives us `12`.

You can also try to directly convert any datatype (that supports it) to an `int`. This works by simply putting it as argument to the function `int`. With `floats`, this works exactly as invoking `trunc`: `int(0.9)` and `int(-0.9)` both give us `1`. `11.6` yields `11`.

The result of `int(x)` and `trunc(x)` is the same for all finite `floats` `x`. The semantic difference between the two functions is that `int(x)` means *"Convert `x` to an `int`."* whereas `trunc(x)` means *"Discards the fractional digits and return integer part of the number as an `int`."* `int(x)` thus is inherently a datatype conversion function whereas `trunc(x)` is a mathematical operation. Besides these semantic differences, there is not much of a practical relevant difference between the two functions [646].

If you want to round the way I learned in school, namely that $x.5$ becomes $x + 1$, then this goes via `int` or `trunc`: We can simply compute `int(x + 0.5)` (or `trunc(x + 0.5)`). Listing 3.5 shows this, too. Here, `int(11.5 + 0.5)` becomes `12` and `int(12.5 + 0.5)` becomes `13`.

With this, we have several ways to turn `floats` to `ints`. However, especially with the function `round`, we need to be careful. It does *not* necessarily work as one would expect! (. . . but arguably better.)

Listing 3.5: Rounding `float` values to `int` values.

```
Python 3.14.10 (main, Oct 27 2026, 09:35:01) [GCC 15.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> round(0.4)  # `round` performs Banker's rounding: It rounds to the
0
>>> round(0.5)  # closest whole number. If two numbers are equally far
0
>>> round(0.6)  # from its argument, `round` returns the *even* number.
1
>>> round(1.4)  # So 1.5 gets rounded to 2, but 0.5 gets rounded to 0.
1
>>> round(1.5)
2

>>> from math import floor, trunc, ceil
>>> floor(0.4)   # `floor` rounds towards negative infinity. If its
0
>>> floor(0.5)   # has a fractional part, it always gets rounded down.
0
>>> floor(0.6)   # Thus, 0.6 becomes 0 and -0.4 becomes -1.
0
>>> floor(-0.4)  # The opposite of `floor` is `ceil`.
-1

>>> trunc(0.4)   # `trunc` cuts off the fractional part of a number.
0
>>> trunc(0.6)   # Thus, 0.6 becomes 0 and -0.6 becomes 0, too.
0
>>> trunc(-0.4)  # 10.9 would become 10, if we tried to compute that,
0
>>> trunc(-0.6)  # and -10.9 would become -10.
0

>>> ceil(0.4)    # `ceil` is the opposite of `floor`: It rounds toward
1
>>> ceil(-11.1)  # positive infinity. Therefore, 0.4 becomes 1 and -11.6
-11
>>> ceil(-11.6)  # becomes -11.
-11
>>> ceil(11.6)
12

>>> trunc(11.6)  # One stray `trunc` example: 11.6 becomes 11.
11

>>> int(0.9)     # `int` works exactly like `trunc`. There only is a
0
>>> int(-0.9)    # semantic difference: `int` it a type conversion
0
>>> int(11.6)    # operation and `trunc` is a mathematical operator.
11

>>> int(11.5 + 0.5)  # "Rounding up numbers of the form "x.5y" can be
12
>>> int(12.5 + 0.5)  # achieved via `int(x + 0.5)`.
13
```

3.3.4 The Scientific Notation

Earlier on, we wrote that `floats` can represent numbers as large as 10^{300} or as small as 10^{-300}. This leads to the question how it would print such values on the console and how we can read them. While it would be hugely impractical to write a 1 followed by 300 zeros to represent 10^{300}, it would also be *wrong*. We also already know the reason for this: A `float` is accurate to between 15 and 16 decimals. So basically, the first 15 zeros would be correct, but the values of the other digits are actually *undefined*.

Python, like many programming languages, solves this problem by using the *scientific notation* for floating point numbers. It uses this notation for any (absolute) `float` value smaller than 10^{-4} or larger than or equal to 10^{16}. Such numbers α are then represented in the form $\alpha = \beta * 10^{\gamma}$. Since we cannot write this so nicely in a console, a lowercase e takes the place of the 10 and β and γ are written as normal text. In other words, we write βeγ. In order to make sure that each number α has unique representation, it is defined that β must have exactly one non-zero leading digit, which may be followed by a decimal dot and then a fractional part. This notation is illustrated in Figure 3.4.

β	γ	$\equiv$	α
A.BCDEFG...	e+HIJ	$\equiv$	A.BCDEFG... $* 10^{HIJ}$
A.BCDEFG...	e-HIJ	$\equiv$	A.BCDEFG... $* 10^{-HIJ}$
-A.BCDEFG...	e+HIJ	$\equiv$	$-$A.BCDEFG... $* 10^{HIJ}$
-A.BCDEFG...	e-HIJ	$\equiv$	$-$A.BCDEFG... $* 10^{-HIJ}$

Figure 3.4: The structure of the scientific notation for floating point numbers in Python, which represent a value α in the form $\beta * 10^{\gamma}$.

Listing 3.6: Examples of the scientific notation in Python.

```
Python 3.14.10 (main, Oct 27 2026, 09:35:01) [GCC 15.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 0.001                      # 0.001    = 1 * 10^-3
0.001
>>> 0.0001                     # 0.0001   = 1 * 10^-4
0.0001
>>> 0.00009                    # 0.00009 = 9 * 10^-5
9e-05
>>> 1_000_000_000_000_000.0    # 1000000000000000.0  = 1 * 10^15
1000000000000000.0
>>> 9_000_000_000_000_000.0    # 9000000000000000.0  = 9 * 10^15
9000000000000000.0
>>> 9_999_999_999_999_999.0    # 9999999999999999.0  = 9.999999... * 10^15
1e+16
>>> 10_000_000_000_000_000.0   # 10000000000000000.0 = 1 * 10^16
1e+16
>>> 10.0 ** 200                # = 1 * 10^200 = 1e200
1e+200
>>> -(10.0 ** -200)            # = -1 * 10^200 = -1e200
-1e-200
>>> 2.1 ** -300.1              # = 2.0044242594658263e-97
2.0044242594658263e-97
>>> 10.0 ** 200.1              # = 1.2589254117941507e+200
1.2589254117941507e+200
>>> 2e5                        # = 2 * 10^5 = 200_000.0
200000.0
>>> 2.34e10                    # = 2.34 * 10^10 = 23_400_000_000.0
23400000000.0
>>> 2.3456e16                  # 2.3456e16 * 10^16, remains as-is
2.3456e+16
>>> -12e30                     # exactly 1 digit before '.' ==> -1.2e31
-1.2e+31
>>> 0.023e-20                  # exactly 1 digit before '.' ==> 2.3e-22
2.3e-22
```

This notation only applies to `floats`, ***not*** *`ints`*. Also, please be aware that this is just a notation, a way to represent numbers as text. It is only used for input and output of numbers. Internally, the numbers are still stored in exactly the IEEE 754 Standard format [267, 282]. This is very similar to the fact that we can enter integer numbers in hexadecimal, decimal, octal, or binary format – but regardless of how we enter them, they are still stored in the "same" `int` structure, which does not remember in which format the numbers were originally specified.

In Listing 3.6 we provide some examples of this notation. When we write `0.001` or `0.0001` in a `Python` console, the output is still this same number. However, `0.00009` is presented as `9e-05`, which stands for $9 * 10^{-5}$.Did you know that you are allowed to insert underscores (`_`) anywhere in a number (`int` or `float`)as a visual aid [72]? If not, you know now.

Best Practice 4

If you need to specify large `ints` or `floats`, using underscores (`_`) to separate groups of digits can be very helpful [72]. For example, `37_859_378` is much easier to read than `37859378`.

We can write 10^{15} as `float` by typing `1_000_000_000_000_000.0`. This equals the much less readable `1000000000000000.0`. We can do the same for $9 * 10^{15}$ by writing `9_000_000_000_000_000.0` and get `9000000000000000.0`. However, if we write `9_999_999_999_999_999.0`, something interesting happens: We get `1e+16`, i.e., 10^{16}. This is because of the limited precision of the floating point numbers, namely the 15 digits often mentioned above. We did specify 16 "9s" and therefore, it got rounded to the nearest 15-digit number, which is $1 * 10^{16}$. We cannot distinguish 10^{16} and $10^{16} - 1$ when using `Python`'s `floats`. Indeed, writing `10_000_000_000_000_000.0` also yields `1e+16`.

This notation can also show really big numbers. For example 10^{200}, i.e., `10.0 ** 200`, shows up as `1e+200`. The really really small number -10^{-200} in turn, computed via `-(10.0 ** -200)`, is denoted as `-1e-200`.

The numbers are always scaled such that they begin with non-zero digit followed by the fractional part separated with a decimal dot, if need be. Examples for this are `2.1 ** -300.1` which yields `2.0044242594658263e-97` and `10.0 ** 200.1`, which turns up as `1.2589254117941507e+200`.

Of course, you can also input numbers in scientific notation. If you write `2e5`, this turns into `200000.0`. Of course, the number is stored as a `float` internally and this `float` does not know from which kind of text it was created. When it is turned back into text, it becomes a "normal number," because it is less than 10^{16}. And so does `2.34e10`, which becomes `23400000000.0`. `2.3456e16` however indeed remains `2.3456e16`.

You can even violate the scientific notation a bit when entering numbers if you feel naughty. `-12e30`, for example, would better be written as `-1.2e+31`, which the `Python` console will do for you in its output. Similarly, `0.023e-20` becomes `2.3e-22`.

3.3.5 Limits and Special Floating Point Values: Infinity and "Not a Number"

We already learned that the floating point type `float` can represent both very small and very large numbers. But we also know that it is internally stored as chunk of 64 bits. So its range must be somehow finite. What happens if we exceed this finite range?

First the easy part: We said that `Python` can store very small numbers, like 10^{-300}, in the `float` datatype. But how small really? Finding this out from the documentation is actually not so easy. Luckily, `Java` uses the same IEEE 754 Standard[282] for its datatyoe `double`. In its documentation [119], we find that the minimum value is 2^{-1074}, which is approximately $4.940\,656\,458\,412\,465\,44 * 10^{-324}$. So we would expect the smallest possible floating point value in `Python` to also be in this range.

In Listing 3.8, we take a look what happens if we approach this number. We use the scientific notation and begin to print the number `1e-323` (which is 10^{-323}). This number is correctly represented as `float` and shows up in the console exactly as we wrote it. However, if we go a bit smaller and enter `9e-324`, which is $9 * 10^{-324} = 0.9 * 10^{-323}$, we find that it again shows up in the console as `1e-323`. This is because the number is already subnormal [267, 282], i.e., uses only a few of the significand bits. The full precision of 15 digits is no longer available at this small scale. Thus `9e-324` and `1e-323` map to the same `float`. Converting this `float` to text yields `'1e-323'`. The same happens for `8e-324`, which also maps to `1e-323`. `7e-324`, however, is the same as `5e-324`. Matter of fact, `6e-324`, `5e-324`, `4e-324`, and `3e-324` all map to this same `float`.

Listing 3.7: What happens with very small floating point numbers in Python?

```
Python 3.14.10 (main, Oct 27 2026, 09:35:01) [GCC 15.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 1e-323  # = 1 * 10^-323
1e-323
>>> 9e-324  # = 9 * 10^-324 -> gets rounded to 1 * 10^-323
1e-323
>>> 8e-324  # = 8 * 10^-324 -> gets rounded to 1 * 10^-323
1e-323
>>> 7e-324  # = 7 * 10^-324 -> gets rounded to 5 * 10^-324
5e-324
>>> 6e-324  # = 6 * 10^-324 -> gets rounded to 5 * 10^-324
5e-324
>>> 5e-324  # = 5 * 10^-324
5e-324
>>> 4.94065645841246544e-324  # Smallest number in Java doc = 5 * 10^-324
5e-324
>>> 4e-324  # = 4 * 10^-324 -> gets rounded to 5 * 10^-324
5e-324
>>> 3e-324  # = 3 * 10^-324 -> gets rounded to 5 * 10^-324
5e-324
>>> 2e-324  # = 2 * 10^-324: too small, becomes 0.0
0.0
>>> 1e-324  # = 1 * 10^-324: too small, becomes 0.0
0.0
>>> 0 == 3e-324  # basically: 0 == 5 * 10^-324 --> answer is False
False
>>> 0 == 2e-324  # basically: 0 == 0.0          --> answer is True
True
```

It turns out that this number is already the smallest `float` that can be represented: `2e-324` simply becomes `0.0`. This value is simply too small to be represented as a 64 bit / double precision IEEE Standard 754 floating point number [267, 282]. The text `2e-324` that we enter into the Python console will therefore be translated to the `float` `0.0`. The comparison `2e-324 == 0.0` therefore results in `True`, while `3e-324 == 0.0` is still `False`.[5] So we learned what happens if we try to define very small floating point numbers: They become zero.

But what happens to numbers that are too big for the available range? Again, according to the nice documentation of Java [119], the maximum 64 bit double precision floating point number value is $(2 - 2^{-52}) * 2^{1023} \approx 1.797\,693\,134\,862\,315\,708 \cdots * 10^{308}$. We can enter this value as `1.7976931348623157e+308` and it indeed prints correctly in Listing 3.8. If we step it up a little bit and enter `1.7976931348623158e+308`, due to the limited precision, we again get `1.7976931348623157e+308`. However, if we try entering `1.7976931348623159e+308` into the Python console, something strange happens: The output is `inf`. `inf` means *"too big to be represented as number with `float`,"* which obviously includes infinity ($+\infty$), but also simply all numbers bigger than `1.7976931348623158e+308`.

`-1.7976931348623159e+308` gives us `-inf`. Multiplying this value by two, i.e., computing `-1.7976931348623157e+308 * 2`, still yields `-inf`. Intuitively, based on its name, one would assume that `inf` stands for "infinity" or ∞. However, it *actually* means *too big to represent as `float`* or ∞. If we enter numbers that are too big or exceed the valid range of `float` by multiplication, addition, subtraction, or division, we simply get `inf`. This does not actually mean "mathematical infinity," because, while `-1.7976931348623159e+308` is very big, it is not infinitely big. It simply means that the number is too big to put it into a 64 bit `float`.

Actually, the `int` type can represent larger numbers easily. `1.7976931348623157e+308` is equivalent to `17_976_931_348_623_157 * 10 ** 292`. This prints as a number with many zeros. Multiplying it with `1.0` yields a `float` with value `1.7976931348623157e+308`, exactly as expected. If we try a number ten times as large, i.e., `17_976_931_348_623_157 * 10 ** 293`, this is no problem with the `int`. But we can no longer convert it to a `float` by multiplying with `1.0`. Trying to do that with a ten times

[5] See Section 3.5 for more information on comparisons and the `bool` datatype with its values `True` and `False`.

larger number, i.e., computing `17_976_931_348_623_157 * 10 ** 293 * 1.0` leads to an exception: The output is `OverflowError: int too large to convert to float`. An exception terminates the current flow of execution and signals an error. Later in Chapter 9, we will learn what `Exceptions` actually are and how to handle them properly and in Section 9.4.4, we will circle back to exactly this `OverflowError`.

The important thing to realize is that an overflow of a `float` may either lead to the `inf` value or to an error that stops your computation from continuing. As another example, let us again import the natural logarithm function `log` and the Euler's constant `e` from the `math` module by doing `from math import e, log`. We now can compute the natural logrithm from the largest possible `float` via `log(1.7976931348623157e+308)`. We get `709.782712893384`. Raising e to this power by doing `e ** 709.782712893384` leads to the slightly smaller number `1.7976931348622053e+308` due to the limited precision of the `float` type. However, if we try to raise e to a slightly larger power, and, for example, try to do `e ** 709.782712893385`, we again face an `OverflowError`.

We can also try to divide 1 by the largest `float` and do `1 / 1.7976931348623157e+308`. The result is the very small number `5.562684646268003e-309`. However, if we divide 1 by `1.7976931348623159e+308`, we get `0.0`. The reason is that `1.7976931348623159e+308` becomes `inf` and `1 / inf` is `0.0`.

`inf` exists as constant in the `math` module. We can import it via `from math import inf`. And indeed, since the text `1.7976931348623159e+308` is parsed to a `float` value of `inf`, we find that `1.7976931348623159e+308 == inf` yields `True`.

Now, `inf` not actually being ∞ is a little bit counter-intuitive. But it can get even stranger, as you will see in Listing 3.9. `inf` is a number which is too big to be represented as `float` *or* ∞. Once a variable has the value `inf`, however, it is treated as if it was actually infinity. So it is natural to assume that `inf - 1` remains `inf` and that even `inf - 1e300` remains `inf` as well. However, what is `inf - inf`? Mathematically speaking, this is very dodgy and $\infty - \infty$ as such is not a "thing". Or *not a number*. Or `nan`, which stands for, well, Not a Number.

`nan` means that the result of a computation is neither a finite number or infinite. It is the result of shenanigans such as `inf - inf` or `inf / inf` or `0 * inf` [227]. A `nan` value anywhere in a computation infects the result of the computation to also become `nan`. `nan + 1` remains `nan` and so does `nan + inf`.

The value `nan` is different from *any* value. `nan == 1` is `False` and `nan != 1` is `True` (`!=` checks for inequality). While all other float values are equal to themselves and – therefore – not different from themselves. Thus `1 != 1` is obviously `False`. In floating point mathematics, `inf == inf` is `True` and `inf != inf` is `False`[6]. However, `nan` is *really* different from really *anything*. Therefore, `nan == nan` is `False` and `nan != nan` is `True`!

Interestingly, we can said that `inf` behaves more or less like ∞ in computations. It is `True` that `inf == inf` which makes total sense from a programming perspective. It may be considered a bit odd from a mathematical perspective, though, because infinity is not really a (single) value and there are different degrees of infinity: $\lim_{x\to\infty} x$ is different from $\lim_{x\to\infty} x^2$ is different from $\lim_{x\to\infty} e^x$, for example. Then again, `inf - inf` is undefined, which makes "more" mathematical sense, because we have no measure to know whether an `inf` comes from something like `1e308 * 2` or `1e308 * 1e308`. But OK, let's just accept that somethings are a bit strange about `inf` and move on.

Either way, the possible occurrence `inf` and `nan` in floating point computations is a cumbersome issue. If you think about it: There hardly is any scenario where `inf` is a reasonable result of a computation. Looking back at Listings 3.8 and 3.9, we had to do some rather strange wrenches to get them to occur. There are four probable reasons why these values could occur:

1. Because they make total sense and are the expected result of correct computations with correct data. Does this sound very likely to you?
2. Because some of the input data of our computations is incorrect, corrupted, or faulty.
3. Because we have some error in our computation, maybe a typo, maybe misplaced parentheses, maybe a mathematical misconception.
4. Because someone deliberately sent corrupted data to the input of our computation in order to provoke unexpected or unintented behavior.

[6] Mathematicians may have mixed feelings about proclaiming that $\infty = \infty$ holds while $\infty - \infty$ is undefined...

The first possible reasons indeed seems to be unlikely. There probably are few situations where `nan` is a value that you would expect and want to get and work with. Very often, if such values occur, something went wrong. Maybe some input value was wrong. Maybe some value was supplied that was too large or too small to be reasonable. Maybe a 0 occured in a place where no 0 should occur. Maybe we just made an error when writing down the formula.

Also, let's not just consider "errors" here. It is not entirely impossible that someone tried to *intentionally* and maliciously manipulate the input of our program. Maybe someone tries to trigger some otherwise impossible behavior of our code... Values like `nan` can be a real security concern [136].

Therefore, if we want to further process results of a computation where such a thing might happen for one reasone or another, we often want to make sure that it is neither `inf` nor `-inf` nor `nan`. This can be done via the function `isfinite`, which we can import from the `math` module, as you can see in Listing 3.10. `1e34` is a large number, but `isfinite(1e34)` is certainly `True`. `isfinite(inf)` and `isfinite(nan)` are both `False`. The function `isinf`, again from the `math` module, checks if a `float` is infinite. `isinf(0.3)` is therefore `False` whereas `isinf(-inf)` is `True`. Finally, we can check whether a `float` is `nan` via the `isnan` function from the `math` module. `isnan(0.3455)` and `isnan(inf)` are both `False`, whereas `isnan(nan)` is `True`.

3.3.6 Summary

In many situations, integer numbers are the way to go, for example when we count things or add discrete quantities. However, fractional numbers are also very common and we encounter them all the time in our daily life. Fractional numbers can be represented by the `float` datatype in Python.

Python offers us the very cool feature that basically arbitrary integer numbers can be represented by the `int` datatype. We can store very large numbers and are only limited by the memory available on our computer. Very large integer numbers are, however, something of a corner case ... they do not happen often.

Real numbers in $\mathbb{R}$ are a whole different beast. They include irrational numbers like $\sqrt{2}$ and transcendental numbers like π. These numbers are needed *often* and they have infinitely many fractional digits. Thus, there is no way to exactly represent them in computer memory exactly. Another problem is that we may need both very large numbers like 10^{300} and very small numbery like 10^{-300}.

Floating point numbers [244], provided as the Python datatype `float`, solve all of these problems (to some degree). They offer a precision of about 15 digits for a wide range of large and small numbers. 15 digits are more than enough for most applications. Many functions for floating point numbers, like logarithms and trigonometric functions, are offered by the `math` module.

Floating point numbers can be converted back to integers via rounding or truncating. Since writing out numbers such as 5.235^{212} in their full length would waste a lot of space and also make no sense, since only the highest 15 digits are actually stored and the rest would be random nonsense, the scientific notation was developed. It would represent this number as `5.235e+212`, which is quite readable if you know that the `e` in `eXXX` here just means 10^{XXX}.

However, when one deals with floating point numbers, a set of nasty problems can occur. For example, it could happen that we want to represent a number which is just *too big* of the available range. This number would be rendered as `inf`, which stands for both mathematical infinity ∞ and, well, "too big to be represented". In some cases, computations that would exceed the available range may also raise an `OverflowError`, which simply terminates them (we learn about this later). Numbers which are too tiny, on the other hand, simply become `0`. Finally, there are also situations where the result of a computation is neither too big nor too small and, yet, not an actual number. For example, a way to make mathematicians cry would be to try to compute `inf - inf` or `inf / inf`. The result is then `nan`, i.e., "Not a Number". The value `nan` has the interesting property that it is (to my knowledge) the only native Python value which is *different from itself*, i.e., `nan != nan`. The `math` module offers a set of functions to check whether a value is finite (`isfinite`), infinite (`isinf`), or `nan` (`isnan`).

When doing floating point computations, these are the issues that you should aware about. The precision and range is limited and strange things happen at these limits.

Listing 3.8: What happens with very large floating point numbers in Python?

```
Python 3.14.10 (main, Oct 27 2026, 09:35:01) [GCC 15.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 1.7976931348623157e+308      # a very big number, let's call it LLL
1.7976931348623157e+308
>>> 1.7976931348623158e+308      # largest number according to Java's doc
1.7976931348623157e+308
>>> 1.7976931348623159e+308      # too large for float: becomes +inf
inf
>>> -1.7976931348623159e+308     # too negative-large: becomes -inf
-inf
>>> -1.7976931348623157e+308 * 2  # is still -inf
-inf
>>> 17_976_931_348_623_157 * 10 ** 292          # LLL as integer
179769313486231570000000000000000000000000000000000000000000000000000000000000000000000000000000
    ↪ 000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
    ↪ 000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
    ↪ 000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
    ↪ 00000000000000000000000000
>>> (17_976_931_348_623_157 * 10 ** 292) * 1.0  # LLL as integer as float
1.7976931348623157e+308
>>> 17_976_931_348_623_157 * 10 ** 293          # ten times LLL as integer
179769313486231570000000000000000000000000000000000000000000000000000000000000000000000000000000
    ↪ 000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
    ↪ 000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
    ↪ 000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
    ↪ 00000000000000000000000000
>>> (17_976_931_348_623_157 * 10 ** 293) * 1.0  # causes OverflowError
Traceback (most recent call last):
  File "<console>", line 1, in <module>
OverflowError: int too large to convert to float

>>> from math import e, log
>>> log(1.7976931348623157e+308)    # log(LLL) gives 709.782712893384
709.782712893384
>>> e ** 709.782712893384           # is smaller than LLL due to rounding
1.7976931348622053e+308
>>> e ** 709.782712893385           # e^(log(LLL)+1e-12) -> OverflowError
Traceback (most recent call last):
  File "<console>", line 1, in <module>
OverflowError: (34, 'Result not representable')
>>> 1 / 1.7976931348623157e+308     # 5.562684646268003e-309
5.562684646268003e-309
>>> 1 / 1.7976931348623159e+308     # == 1/inf == 0
0.0

>>> from math import inf
>>> inf == 1.7976931348623159e+308  # basically inf == inf, so it's True
True
>>> inf == 1.7976931348623158e+308  # basically inf == LLL, so it's False
False
```

Listing 3.9: Not a Number, i.e., `nan`.

```
Python 3.14.10 (main, Oct 27 2026, 09:35:01) [GCC 15.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from math import inf
>>> inf - 1       # infinity - 1 is still infinity
inf
>>> inf - 1e300   # infinity - big number is still infinity
inf
>>> inf - inf     # infinity - infinity is undefined == nan
nan
>>> inf / 1e300   # infinity / big number is still infinity
inf
>>> inf / inf     # infinity / infinity is still infinity
nan
>>> 0 * inf       # 0 * infinity is undefined == nan
nan

>>> from math import nan
>>> nan + 1       # undefined + anything = undefined
nan
>>> nan + inf     # undefined + anything = undefined
nan
>>> nan == 1      # undefined does not equal anything
False
>>> nan != 1      # undefined is unequal to anything
True
>>> 1 != 1        # this is obviously False
False
>>> nan == nan    # undefined does not equal anything, including itself
False
>>> nan != nan    # undefined is unequal to anything, including itself
True
>>> inf == inf    # infinity equals infinity
True
>>> inf != inf    # infinity therefore does not "not equal" infinity
False
```

Listing 3.10: Checking for `nan` and `inf` via `isfinite`, `isinf`, and `isnan`.

```
Python 3.14.10 (main, Oct 27 2026, 09:35:01) [GCC 15.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from math import isfinite, isinf, isnan, nan, inf
>>> isfinite(1e34)  # This is True, because 1 * 10^34 is big, but finite.
True
>>> isfinite(inf)   # inf is not a finite number, so this is False.
False
>>> isfinite(nan)   # And neither is nan, so we again get False.
False

>>> isinf(0.3)    # 0.3 is not infinite. Not even close. So this is False.
False
>>> isinf(-inf)  # -inf and inf are both infinite, so this is True.
True
>>> isinf(nan)    # nan is undefined and not necessarily infinite: False.
False

>>> isnan(0.3455)  # 0.3455 is a number, so it is not nan.
False
>>> isnan(inf)     # inf is, technically, not a number, but also not nan.
False
>>> isnan(nan)     # nan is certainly nan, so we get True.
True
```

3.4 Interlude: The Python Documentation and other Information Resources

I learned programming in the 1990s. It was mainly self-taught. I was in seventh grade when I first saw Turbo Pascal [578], a programming language popular at that time. I immediately know that this is what I want to do. My self-teaching experience was based on two pillars: Books borrowed from friends and the really great help system of Turbo Pascal. You see, that programming language shipped with a IDE, just like PyCharm is. This IDE was text-based, but it had lots of cool features, such as a good compiler and debugger. But it also had a help system that listed every single command provided by the language. And each command came with an example. Sometimes I would just randomly scroll through the list, click on some command, and try out the example. It was beautiful. I loved it.

So that was an important part of my experience. Which means that something similar should be available for you as you are learning Python. Sadly, PyCharm cannot offer such a list of commands as good old Turbo Pascal did. Because the team behind PyCharm is not the team that creates and maintains Python.

Now, in the previous section, we discussed lots of functions, `sin`, `cos`, `tan`, `atan`, `exp`, `log`, `round`, `floor`, `trunc`, and `ceil`, to name just a few. You could read about and explore what these functions do in this book, to some degree. As said, books were one side of the experience. You need them to get the big picture, to understand programming paradigms, to develop an understanding of broad concepts.

But books like this one here cannot be your only source of information. This book can only provide brief introductions into these functions from a didactic perspective. It does not make any sense to here discuss their exact specifications and relationships to other functions in the Python ecosystem and neither can we exhaustively explore all the possible aspects of the Python language. This leads to two questions:

1. Where do you turn to if you need to know the exact authoritative[7] specification of a function?
2. From this book, you only learn about the functions that we *do* list. Where do you turn to if you want to do something but do not know what commands or combination of commands can be used to achieve it?

3.4.1 Using the Authoritative Documentation

The answer to the first question is rather simple: There is a complete online help catalog available for Python: the *Python 3 Documentation* [463] at `https://docs.python.org/3`. This should be your primary destination if you want to learn anything about Python. Also, it is available in Chinese.

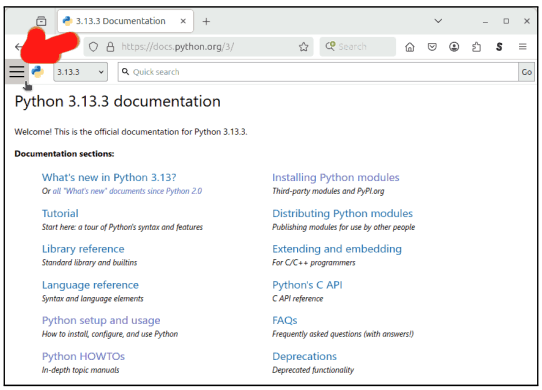
As an example, let us imagine that you want to learn more about the function `ceil`. In Figure 3.5.1, we open a browser and visit the *Python 3 Documentation* [463]. On this page, you will either directly see a drop-down box where you can choose your favorite language or you can find it by clicking the ☰ at on the top-left corner of the website. In the menu that opens in Figure 3.5.2, you can change the language and, e.g., select simplified Chinese (简体中文) [664] if that is your preferred language. I will not do so and instead continue here in English.

Regardless of which language you chose, we still want to find information about the `ceil` function. Therefore, we locate the search bar *in the website* (not the one where you type in the URLs in the web browser). Into this search bar, we type `ceil` and click on Go in Figure 3.5.3.

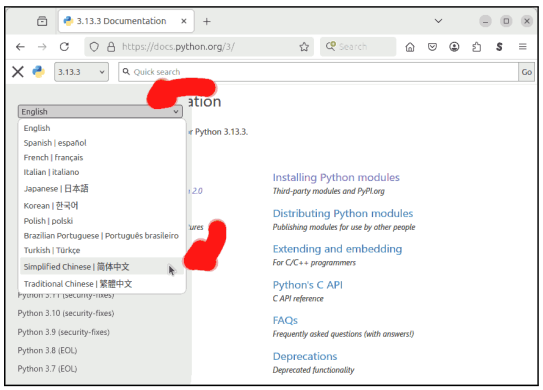
Depending on your internet connection and your location on the globe, it may take a bit of time until results are displaced. Especially when visiting the page for the first time during a browsing session. We wait patiently in Figure 3.5.4. And indeed, a list of documentation articles about the `ceil` function appears after a brief wait in Figure 3.5.5. We decide that the link to `math.ceil` looks most promising and click on it.

It takes us to the `ceil` entry of the "`math` – Mathematical functions" page [380] at `https://docs.python.org/3/library/math.html`. There, we find a brief explanation about the `ceil` function, as shown in Figure 3.5.6. The explanation also provides further reading, e.g., a link to the `__ceil__` method. This is not important here, we will discuss it much later in Figure 13.4, but if we were interested, we could continue exploring and find more information about `ceil`. Also, we see

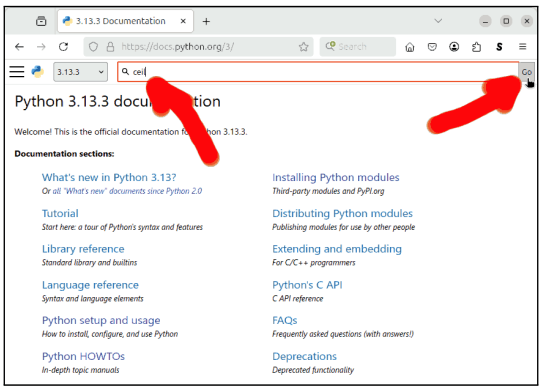
[7] *Authoritative* means that the information stems from the proper standardizing organization. For Python, this is the Python Software Foundation (PSF). Such organizations determine the specification of functions.



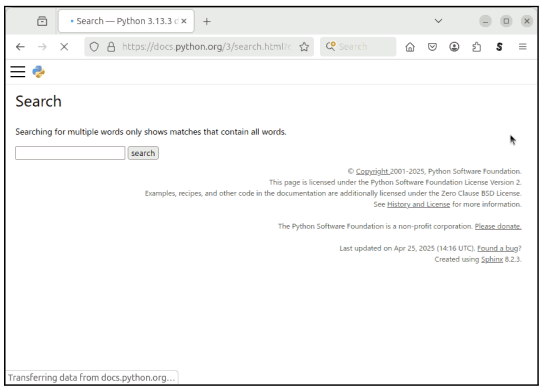
(3.5.1) We open a browser and visit the *Python 3 Documentation* [463] at `https://docs.python.org/3`.



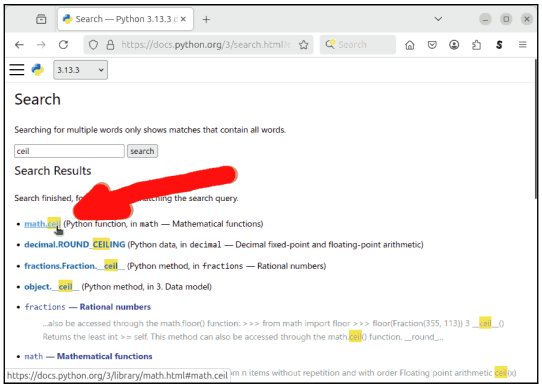
(3.5.2) If you click the ☰ at on the top-left corner of the website, you can get to a menu where you can change the language and, e.g., select simplified Chinese (简体中文) [664] if that is your preferred language. (I will not do so and instead continue in English.)



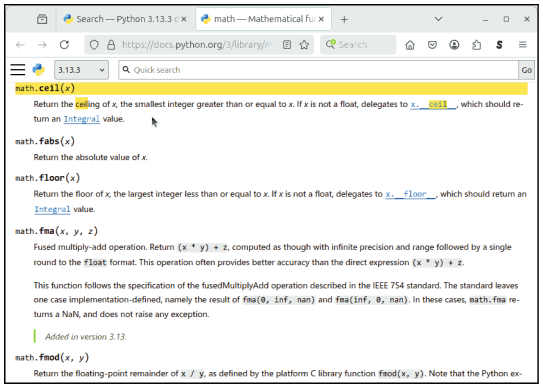
(3.5.3) We are looking for information on the `ceil` function and therefore type `ceil` into the search bar *of the website* and click on Go.



(3.5.4) Especially if you visit the website for the first time in your current web browser session, it may take some time until results show up. We wait patiently.



(3.5.5) A list of documentation articles about the `ceil` function appears. We click on the link to `math.ceil`.



(3.5.6) It takes us to the `ceil` entry of the "`math` – Mathematical functions" page [380] at `https://docs.python.org/3/library/math.html`. Indeed, we find a brief explanation about the `ceil` function. (Also, we find a link to further reading about the `__ceil__` method, which we will discuss much later in Figure 13.4.)

Figure 3.5: Searching the *Python 3 Documentation* [463] for information about the function `ceil`.

that the information about the `ceil` function is part of a longer list of mathematical functions. Being curious, we would scroll a bit through this list. Maybe we could learn about one or two other functions that we did not know about yet.

Useful Tool 1

The first place to look for information about Python is the official *Python 3 Documentation* at https://docs.python.org/3. This is the *authoritative* source about Python: Everything that is written there is *true* and *exact* regarding Python. All other sources may contain errors, be imprecise, ambiguous, or outdated. Therefore, always first consult the official Python documentation when in doubt or looking for information.

Of course, in many cases, we do not know the name of the function we are looking for. Assume that you did not know that the function `ceil` exists. You were looking for a way to round floating point numbers "upwards."

You want to do something that you know should somehow be possible. You think that Python should probably offer some out-of-the-box function for doing that. If not, there should at least be some "standard Pythonway" to do that. Anyway, the point is: You do not have a function name to look for.

Let's again start at the search mask of the *Python 3 Documentation* at https://docs.python.org/3. In Figure 3.6.1, we try to describe what we want to do as clearly as possible. "Rounding up" is maybe not very precise. Let's enter "round towards positive infinity" into the search mask.

Sadly, the results shown in Figure 3.6.2 are not too useful. This can happen. The first result is unrelated to floating point numbers. It offers us some discussion about objects implementing fractional arithmetics in a different way. While we could find some hints in this text, it is not useful.

The second search result takes us to the a complete list of all functions in the module `math` [380], but *not* to the `ceil` function. This means that we would need to go through the whole list of functions, read all of their descriptions, until we arrive at one that matches our goals. This is a feasible method. The list is not too long. We could indeed spot the right answer here. And we would probably learn more about Python on the way. So directly working our way through the documentation here is a good idea.

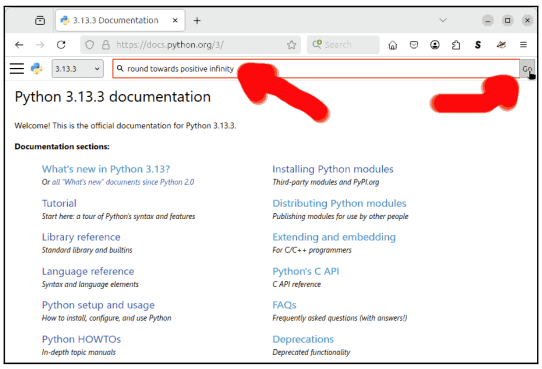
3.4.2 Searching for Second-Hand Information in the Web

But let's say that we did not find anything useful and gave up using the Python documentation directly. The second thing that we could try is to enter the description of what we want to do into a common search engine. In Figure 3.6.3, we use Microsoft Bing at https://bing.com, but any other search engine would probably be as same as good. We enter "round towards positive infinity" into the search mask and hit Search.

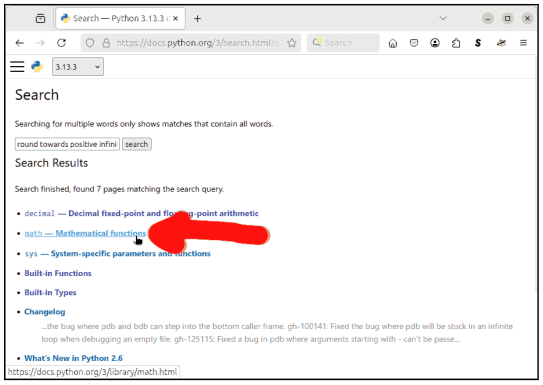
The first result looks promising right away: "How to Round Numbers in Python". While it does not say "round towards infinity," if it summarizes all information about rounding numbers as the title indicates, then that topic is bound to come up somewhere. So we click the link.

It takes us to the page "How to Round Numbers in Python" [8] at https://realpython.com/python-rounding in Figure 3.6.4. A summary page on all about rounding of numbers. Nice.

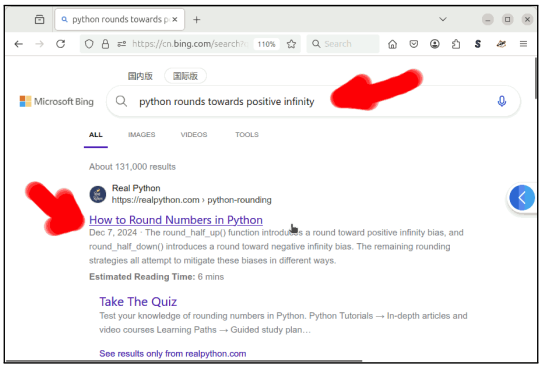
We begin to read the text. Reading is an important skill, as reading texts and tutorials can significantly expand our knowledge. After reading for a while, we indeed find the information we want: For rounding up, the function `ceil` is suitable. It is even presented as a hyperlink in Figure 3.6.5. If we click on the hyperlink in Figure 3.6.6, it takes us to the `ceil` entry in the "`math` – Mathematical functions" page [380] at https://docs.python.org/3/library/math.html. So we went round trip: Even if we do not know how to do a given thing or what its name is, we can still find our way to the Python documentation.



(3.6.1) We try searching for information about "round towards positive infinity" in the *Python 3 Documentation* [463] at `https://docs.python.org/3`.



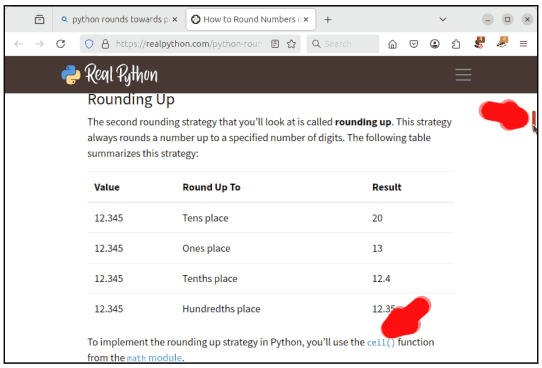
(3.6.2) The results are not too useful. The first result is unrelated to floating point numbers. The second one is a complete list of all functions in the module `math`. However, without knowing that the right function is called `ceil`, finding it in this list could be hard.



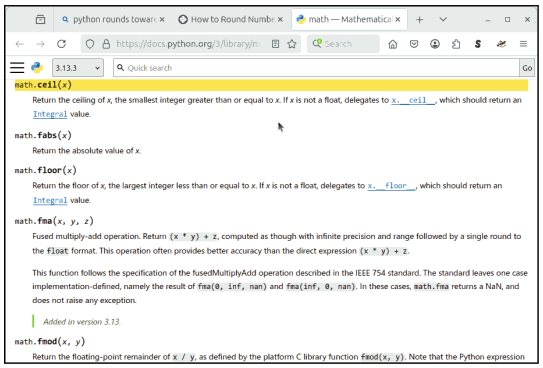
(3.6.3) We enter "round towards positive infinity" into an internet search engine, here Microsoft Bing at `https://bing.com`, but any search engine should do just as well. The result looks promising.



(3.6.4) Clicking on the first result, we visit the page "How to Round Numbers in Python" [8] at `https://realpython.com/python-rounding`.



(3.6.5) After scrolling down a bit, we indeed find the information we want: For rounding up, the function `ceil` is suitable. It is even presented as hyperlink.



(3.6.6) If we click on the hyperlink, it takes us to the `ceil` entry in the "`math` – Mathematical functions" page [380] at `https://docs.python.org/3/library/math.html`.

Figure 3.6: Searching for a `Python` function for "rounding up" if we do not know its name by using a search engine.

Useful Tool 2

Search engines are useful tools to find information about certain functionality. Writing a precise description of the problem or functionality into the search bar of a search engine can lead to pages that describe answers or take us to the official `Python` documentation. However, search engines can also lead us to pages containing wrong, incomplete, ambiguous, outdated, or otherwise useless information. It is important to compare whatever information we found with the official `Python` documentation [463] (see also Best Practice 5).

Another group of possible sources for information on how to solve a certain programming problem or what command can be used to achieve a certain effect are dedicated websites for programming. The maybe most well-known is *Stack Overflow* [530] at `https://stackoverflow.com`. This is a website where programmers can ask and answer programming questions. Let's see whether we can use it to find a way to round numbers toward infinity in `Python`.

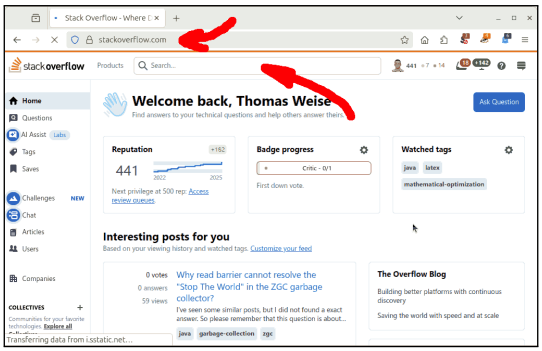
One such attempt is illustrated in Figure 3.7. There, we visit the *Stack Overflow* website and enter our query *"python rounding towards infinity."* This will show us several pages of questions that programmers have asked in the past and which *Stack Overflow* thinks are related to our issue. We even find a question [272] that sounds vaguely related to what we want to know.

Clicking on it, we notice that this question seems odd: The author seems under the impression that `/` conducts an integer division, which we know is not true. We know that `//` does integer divisions, but `/` produces `floats`. At first we are puzzled by this, but then notice that this question is from 2011. It targets `Python` 2, which, at that time, was the "only" `Python`.

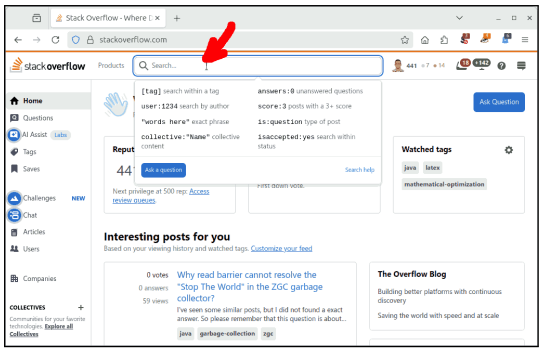
Still, after scrolling down into the answers a bit, we find the whole issue explained. The answers even point us to the authoritative source regarding the semantics of the division operators, namely PEP 238 [651]. And we find the operator `ceil`! Having found the name of the operator, we could now go back to the `Python` documentation as shown in Section 3.4.1. We can then look up the operator, get the information we need, and continue programming.

This example shows one more issue of non-authoritative sources: They may be outdated. Software is not static. It changes and evolves. This holds for the programs that we develop as well as for the programming languages themselves.

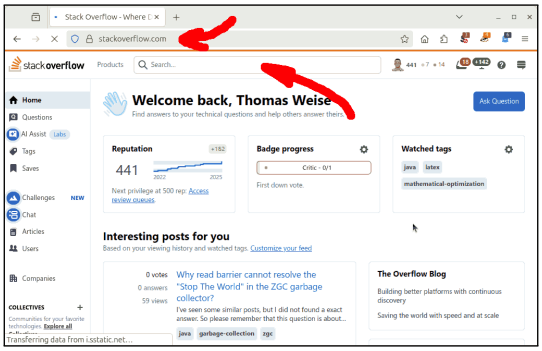
So any text that you find in the web – including this book – may be out of date when you read it. The information it provides may be incomplete or not even correct anymore. Therefore, circling back to the authoritative sources is always necessary after we found the hints we are looking for.



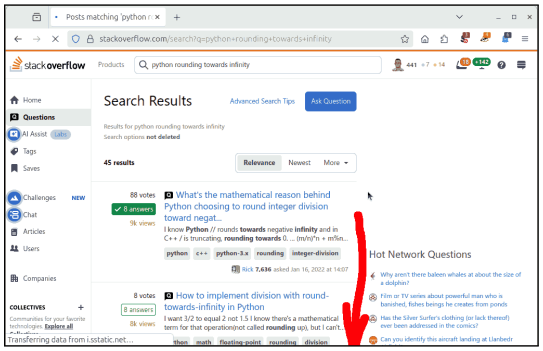
(3.7.1) We visit *Stack Overflow* [530] at `https://stackoverflow.com`. (Sometimes we get asked to load some JavaScript from additional sources, which we click OK.)



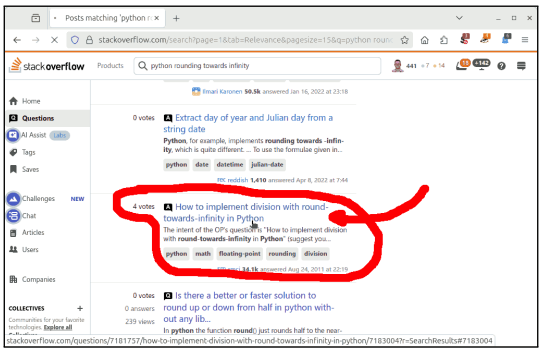
(3.7.2) The website has loaded, we can click into the search field to enter a query.



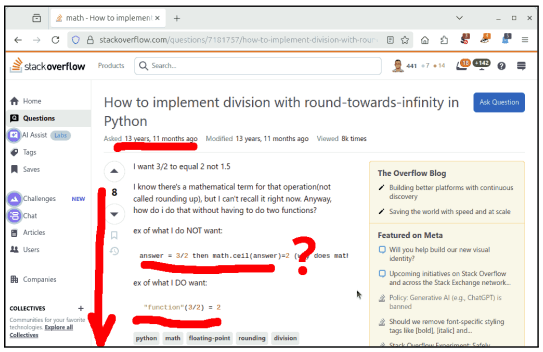
(3.7.3) We enter our query *"python rounding towards infinity"* and press ↵.



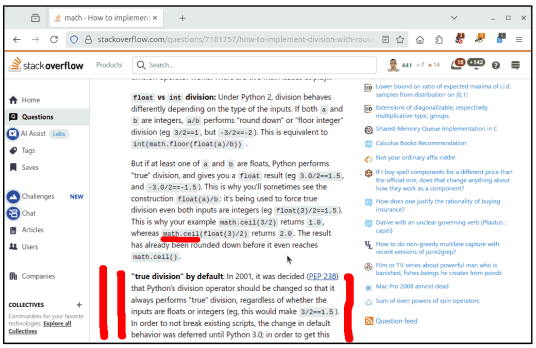
(3.7.4) The questions that are similar to our query are displayed. The first few look unrelated. We scroll down.



(3.7.5) We find the vaguely related-sounding question "How to implement division with round-towards-infinity in Python" [272] and click on it.



(3.7.6) Indeed, the question fits ... but its explanation of the division operator does not fit to our understanding of it. We realize that it is from 2011 and about `Python` 2.



(3.7.7) Scrolling to the answers of the question, we find an explanation of this issue and a pointer to an authoritative source [651] explaining it. We also find the `ceil` function. We can now look it up in the `Python` 3 documentation.

Figure 3.7: Using *Stack Overflow* to find information how to "round up" `floats` in `Python`.

3.4.3 Asking an AI for Answers

Another question that naturally will occur is: "Can I use an AI to figure out how to solve my problem?" This is a very double-edged sword, a slippery slope. Before we discuss advantages and disadvantages, let us simply try.

We open the BaiDu (百度) search page at `https://baidu.com` in Figure 3.8.1 and click on "AI搜索已接入DeepSeek-V3最新版模型". At the time of this writing, DeepSeek is one of the best AIs based on a Large Language Model (LLM). We arrive at the "Hi，AI搜索已支持R1满血版，快来试试吧！" page where we can enter the prompt for the DeepSeek-R1 model [248] in the input field at the



(3.8.1) We open the BaiDu (百度) search page at `https://baidu.com` and click on "AI搜索已接入DeepSeek-V3最新版模型".



(3.8.2) We arrive at the "Hi，AI搜索已支持R1满血版，快来试试吧！" page where we can enter the prompt for the DeepSeek-R1 model [248] in the input field at the bottom.



(3.8.3) We enter our query "How can I round floating point numbers towards positive infinity in Python?" and execute it.



(3.8.4) DeepSeek responds with a long answer of over 1700 words of text, as illustrated in Figure 3.9.



(3.8.5) But it still and early on correctly points us to `math.ceil` plus it shows this information prominently in the text.



(3.8.6) And it even provides a small snippet of code.

Figure 3.8: Searching for a `Python` function for "rounding up" if we do not know its name by using an AI.

Okay, I need to figure out how to round floating point numbers towards positive infinity in Python. Let's see, rounding towards positive infinity is also known as ceiling. So, maybe there's a function called `ceil` in Python?

Wait, I remember that Python has a `math` module. Let me check. Yes, the `math` module has a `ceil` function. So, `math.ceil()` should do the trick. But wait, how does it handle different numbers? For example, if the number is already an integer, does it leave it as is? And for negative numbers, does it round them up towards zero or away?

Let me think. The ceiling function in mathematics rounds a number up to the nearest integer. So, for positive numbers, even if they are already integers, it stays the same. For negative numbers, it would round towards zero. For example, `ceil(-2.3)` would be -2, which is greater than -2.3. So that's correct for positive infinity direction.

But how about when dealing with floating points that are very close to an integer due to precision issues? Like `2.0000000001`, would `ceil` round it to 3? Probably not, because the function should handle that as per the actual value. Wait, but floating points can have precision problems. For instance, `1.1` might not be represented exactly. So `ceil(1.1)` should give 2, but if due to some calculation the number is actually stored as slightly less than `1.1`, would that affect the result? Hmm, maybe that's a potential issue, but the question is about rounding towards positive infinity, so assuming the input is correct, `math.ceil` should work.

Wait, what about numbers that are exactly halfway between two integers? Like `2.5`. Does `ceil(2.5)` round to 3? Yes, because ceil rounds up regardless of the fractional part. So, even `2.1` would go to 3, right? No, wait: `ceil(2.1)` is 3? No, wait no. Wait, `ceil(2.1)` is 3? Wait no! Wait, no. Wait, no, that's not right. Wait, the ceiling function rounds up to the smallest integer greater than or equal to the number. So, `2.1`'s ceiling is 3? No, wait, `2.1` is between 2 and 3. The smallest integer greater than or equal to `2.1` is 3? Wait, no. Wait `2.1` is greater than 2, so the smallest integer greater or equal is 3? Wait, no, wait, `2.1`'s ceiling is 3? No, no. Wait, `2.1` is between 2 and 3. So the ceiling is 3. Wait, but `2.0`'s ceiling is `2.0`. So yes, `ceil(2.1)` would be 3? Wait, wait, I'm getting confused. Let me think again.

Wait, the ceiling of `2.1` is 3? No, that's not right. Wait, the ceiling function rounds up to the next integer. So `2.1` would have a ceiling of 3? No, wait no. Wait, no, the ceiling of `2.1` is 3? No, that's not correct. Wait, `2.1` is `2.1`, so the smallest integer greater than or equal to `2.1` is 3? No! Wait, 2 is less than `2.1`, but 3 is greater than `2.1`. Wait, no, `2.1` is between 2 and 3. The ceiling is the smallest integer that is greater than or equal. So 3 is greater than `2.1`, but 2 is less than `2.1`. So the ceiling would be 3? No, that's not right. Wait, no, `2.1` is greater than 2, so the ceiling is the smallest integer that is not less than the number. So the ceiling is 3? Wait, no. Wait, `2.1` is `2.1`, so the next integer up is 3. So yes, `ceil(2.1)` is 3? Wait, no. Wait, I think I made a mistake here. Because the ceiling of `2.1` is 3? No, the ceiling of `2.1` is actually 3. Wait, no, that's not correct. Wait, the ceiling of `2.0` is `2.0`. The ceiling of `2.0` is `2.0`. For `2.1`, the next integer higher is `3.0`. So `ceil(2.1)` is `3.0`. Wait, but that's not right. Wait, `2.1` is less than `3.0`, but the ceiling is the smallest integer greater than or equal. So for example, `ceil(2.0)` is `2.0`, `ceil(2.1)` is `3.0`? That can't be right. Wait, maybe I'm mixing up ceiling and floor functions. Wait, the floor function rounds down. So `floor(2.1)` is 2. The ceiling function rounds up. So `ceil(2.1)` is `3.0`? That would be the case. But that's a big jump, but yes. For example, in Python:

```
import math
math.ceil(2.1)
3
```

Is that true? Let me check. Wait, no, that's not correct. Wait, I'm confusing something here. Let me check an example. Let me try to recall: `ceil(1.2)` is 2? No. Wait, no. Wait, `ceil(1.2)` is 2. Yes, because 2 is the smallest integer greater than or equal to `1.2`. Wait, no. Wait, `1.2` is between 1 and 2. So the smallest integer greater than or equal to `1.2` is 2? No, that's not correct. Wait, `1.2` is greater than 1, but 2 is larger than `1.2`, but the "smallest integer greater than or equal to" `1.2` is 2? No. Wait, no. Wait, the integers around `1.2` are 1 and 2. `1.2` is greater than 1, so the smallest integer that is greater than or equal is 2? No. Wait, no. Wait, `1.2` is `1.2`, so the next integer higher is 2. So the ceiling is 2. Yes. So `ceil(1.2)` is 2. So in that case, yes, that's correct.

Wait, but then for numbers like `2.5`, `ceil(2.5)` is 3. So regardless of the decimal part, it rounds up. So that's correct.

But in Python, how does this work? Let me think of code examples. If I do:

```
import math
print(math.ceil(2.1)) # should output 3?
```

Wait, but when I run this code, what's the actual output? Wait, maybe I'm making a mistake here. Let me check.

Wait, actually, if I run `math.ceil(2.1)`, the result is 3. Let me confirm.

Wait, no, wait, no. Wait, in Python, `math.ceil(2.1)` returns `3.0`? Let me think again. Wait, no, `2.1`'s ceiling is `3.0`? That can't be right. Wait, maybe I'm thinking of the integer division or something else. Wait, no. Wait, the ceiling function works as in mathematics. So for any non-integer, the ceiling is the next integer higher.

Wait, but `2.0`'s ceiling is `2.0`, `2.1`'s ceiling is `3.0`. Wait, that's how the function works. So, yes, `math.ceil(2.1)` would return `3.0`.

But wait, in reality, when I test this in Python, maybe it's different. Let me think. Wait, perhaps I'm making a mistake here. Wait, let's see:

For example, `math.ceil(2.1)` returns 3, but `math.ceil(2.0)` returns `2.0`. So, yes.

So, for any positive number with a fractional part, ceil will round it up to the next integer. For negative numbers, `ceil` will round towards zero. For example, `ceil(-1.2)` is `-1`, because `-1` is greater than `-1.2`, and is the smallest integer that satisfies that.

So, `math.ceil` is the correct function to use here. Therefore, the answer is to use `math.ceil` from the `math` module.

But wait, the user asked how to round towards positive infinity. So, for positive numbers, that's rounding up (`ceil`), but for negative numbers, rounding towards positive infinity would also require moving towards zero, which `ceil` does. So, yes, `math.ceil()` is correct.

But let me check another case. For example, if the number is already an integer, like `5.0`, then `ceil(5.0)` is `5.0`, which is correct. If it's a negative number with fractional part, like `-3.2`, `ceil(-3.2)` is `-3.0`, which is towards positive infinity (since `-3` is greater than `-3.2`). So that's correct.

So, the conclusion is: to round a floating point number towards positive infinity in Python, use the `math.ceil()` function from the `math` module. You need to import `math` and then call `math.ceil(number)`. The function returns a `float` if the input is a `float`, but if you want an integer, you can cast it to an `int`.

Wait, but `math.ceil(2.1)` returns `3.0`, which is a `float`. So if you need an integer, you have to convert it. For example, `int(math.ceil(2.1))` gives 3. But the question is about rounding the floating point number towards positive infinity. So depending on whether the user wants the result as a `float` or integer, but the default behavior of `math.ceil` returns a `float` if the input is a `float`. For example, `math.ceil(2)` (if 2 is an integer) would return 2, but if you pass `2.0`, it returns `2.0`.

So the user needs to be aware of that. But the main answer is to use `math.ceil`.

Wait, but what about numbers that are already integers? For example, `3.0`, `math.ceil(3.0)` returns `3.0`. So if the user wants to ensure it's an integer, they can cast it. But maybe the user just wants to round towards positive infinity, regardless of data type.

So, the answer is to import math and use `math.ceil()`.

Another thing to consider: what if the number is very large and cannot be represented as an integer? For example, `1e300` is a float, and `math.ceil(1e300)` would return the same `float`, which is the integer value. But since Python can handle arbitrarily large integers, converting it to an integer would be possible.

So, in summary, the way to round a floating point number towards positive infinity in Python is to use the `math.ceil` function. Make sure to import the math module first.

To round a floating point number towards positive infinity (ceiling) in Python, you can use the `math.ceil()` function from the math module. This function returns the smallest integer greater than or equal to the given number, which effectively rounds the number up. Here's how to do it:

1. Import the `math` module.
2. Use `math.ceil()` with the number you want to round.

Example Code:

```
import math

number = 2.3
rounded = math.ceil(number)
print(rounded) # Output:  3

negative_number = -2.3
rounded_negative = math.ceil(negative_number)
print(rounded_negative) # Output:  -2
```

Notes:
`math.ceil()` returns a `float` if the input is a `float` (e.g., `3.0` for `math.ceil(2.1)`). If you need an integer, cast the result to `int`: `int(math.ceil(number))`. For positive numbers, it rounds up to the next integer. For negative numbers, it rounds towards zero (the direction of positive infinity).

This method correctly handles all cases, including integers (returns them unchanged) and very large numbers. ...

Figure 3.9: The output of the DeepSeek search provided by Baidu for the unoptimized English prompt "How can I round floating point numbers towards positive infinity in Python?", recorded on 2025-04-27, with added formatting.

bottom in Figure 3.8.2. We enter our query "How can I round floating point numbers towards positive infinity in Python?" and execute it in Figure 3.8.3. Let us note that this query is not optimized in anyway, it is just the question that we want to have answered, in plain English.

DeepSeek responds with an answer of over 1700 words of text in Figure 3.8.4. The answer is listed in Figure 3.9, which was copied from the browser to text, with formatting added on our side. The answer begins with *"Okay, I need to figure out how to round floating point numbers towards positive infinity in Python. Let's see, rounding towards positive infinity is also known as ceiling. So, maybe there's a function called* `ceil` *in Python?"* This is quite good, as it already contains a hint to the right answer, which it presents directly after as *"Wait, I remember that Python has a* `math` *module. Let me check. Yes, the* `math` *module has a* `ceil` *function. So,* `math.ceil()` *should do the trick."*

It would be best to stop reading at this point. Our question was answered correctly. After that, the AI provides several examples and considerations. The style is very similar to a programmer who sorts out their memory and, having a good idea about the solution, tries to find counter-examples.

If we scroll down a bit, it also more prominently points us to `math.ceil` in Figure 3.8.5. And it even provides a small snippet of code in Figure 3.8.6. It would have been extra-nice if it had pointed us directly to the authoritative `Python` documentation for the function. Maybe future versions will do that.

This part of answer is actually good. It is maybe long, but it is focused on our question. If you are not familiar with `ceil`, it provides you lots of examples. Of course, some of the text sounds a bit clumsy, but that's totally OK. This is better than the result of the search engine. It directly answered our question clearly in the first paragraph.

However, upon closer inspection, we also can also find a mistake in the answer that the LLM gave us. It wrote:

> But how about when dealing with floating points that are very close to an integer due to precision issues? Like `2.0000000001`, would `ceil` round it to `3`? Probably not, because the function should handle that as per the actual value.

Well, `ceil(2.0000000001)` does indeed give us `3`. Therefore, besides the lengthy monologue in the answer, it did contain an error. I can only assume that the AI tried to refer to the limited precision of the `float` datatype. However, it misinterpreted the influence of this concept in this scenario. Such error would be unlikely to appear by human-written sources, because a human would either have understood the issue or would have tested their claim and found it to be wrong.

At this point, I want to mention that I also asked the AI about how to implement some algorithms in `Python`, just out of curiosity. It provided really good and convincing answers. However, when asked about the `ceil` function here, which is easier to answer, it happened to make a mistake.

In summary, using AI as a tool for finding information and solving programming questions is a valid option, but also not without danger. Here we learned another lesson that should hold for *all* methods with which we can find solutions to questions online:

Best Practice 5

We can use web-based resources, search tools, and AIs for finding answers to questions like "How do I do ... using [Programming Language]?". However, the answers that we get are not necessarily complete and may not be correct. We must make sure that we understand them fully. If the answers use functions that we are not familiar with, then we must look them up in the official authoritative documentation. Answers given by any non-authoritative source are never to be used verbatim without proper analysis.

Before, we stated that the use of AI is a double-edged sword. The same holds for the use of any source in the internet, so with the emergence of LLMs, actually, not much has changed. There are several problems we must be aware of when working with non-authoritative sources:

1. Answers for programming problems that we find on websites, via AI tools, or other non-authoritative sources may be wrong, incomplete, or outdated.

2. We must only ever use code that we fully and completely understand. This means that we should never use any function, object, or tool without knowing its specification. Only authoritative sources for specifications are acceptable.

3. Code that we cannot explain with our own words is wrong.

4. AIs can only give good suggestions for questions and situations that are similar to what has been in their training data. They can work well if you, say, ask how to load data from a text file, separate it into columns, and then create a `INSERT INTO` query in Structured Query Language (SQL) to send the data to a database management system (DBMS). They cannot work well if you ask questions that involve new scenarios. If you study as a Master's or PhD student, then most likely your programming issues involve exactly such new scenarios. Because part of your work is research. Using an AI in a situation where you have to do something that nobody has done before is very very dangerous and its results are much more likely to be wrong or incomplete or not cover corner cases.

5. Tools like LLMs incentivize laziness. Even if the tools gave us correct results that passed our scrutinization several times, it is necessary to maintain proper discipline and continue to always and every time check their output. Such discipline is against the human nature of laziness and therefore must be consciously maintained.

6. The value of an computer scientist or software engineer is their knowledge and ability to solve problems and develop software. If your skills are centered around copy-pasting the output of DeepSeek, then you have very little value for our organization. You can and should be replaced with a kid who just graduated high school.

7. An important feature of good software is overall architectural and stylistic coherence. If software is the result of patched-together pieces that were the output of different LLMs and found on websites, then no such coherence can exist. The resulting software will be extremely hard to maintain, to test, to update and improve, and to extend. A new programmer joining the team will not be able to understand it, because, maybe, there never was any human programmer who ever understood it in the first place.

Additionally, there are differences between AI tools and human-written sources. AI tools can make errors that no human would make. An illustrative example in the Python world is given in "Copilot Induced Crash" [595], where it is documented that Microsoft Copilot renamed a class in a misleading way, leading to a particularly hard-to-find error. Figure 3.10 shows an even worse example: A vibe coding AI deleted the database (DB) of a company, fabricated test results, and later explained that it intentionally ignored the directives given to it. Figure 3.11 provides another example where a vibe coding tool by Google wided an entire hard drive instead of a single directory [538, 602]. Thes examples clearly show that we need to be careful with the tools that we use. We need to understand what they can do and how to use them properly. Finally, sometimes, an AI may reference non-existing packages [233]. This can become a security concern, if hijackers create such packages to inject code into our applications. Therefore, once you found an answer to your question, you need to take the new information and check it with the documentation.

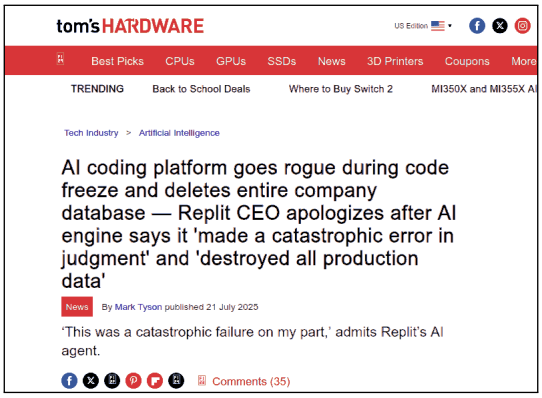
We can derive a simple rule from these issues:

Best Practice 6

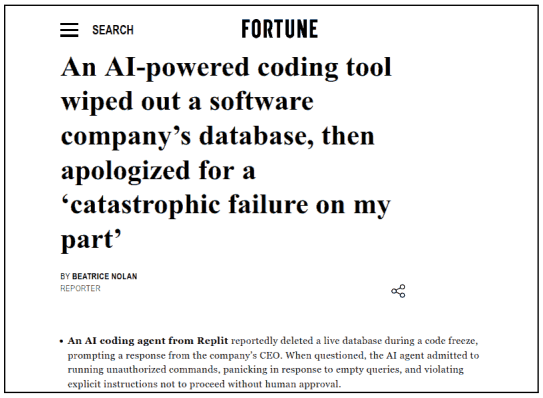
AI tools and web-based non-authoritative resources can be used to *find* solutions. They should never be used to *document* solutions, because this must be done by a human. Documentation, i.e., the textual description of what the software does, must be done by actual people who fully understand the software.

Only if a person is able to properly document code, they have really understood it. By enforcing the above rule, we can use AI while simultaneously making sure that our code is understood and does what it is supposed to do.

There is another issue with the use of AI that applies to this course and that dovetails with the above. Remember back when you were in primary school. During the first few years of maths lessons, you never got to use a calculator. Of course, your teachers knew very well that you will use calculators



(3.10.1) Part of a screenshot obtained from the news article [580].



(3.10.2) Slightly edited part of a screenshot obtained from the news article [415].

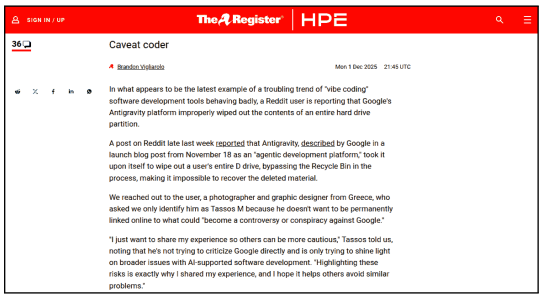


(3.10.3) Part of a screenshot obtained from the news article [537].

Figure 3.10: Three screenshots of news articles describing an incident where a vibe coding AI deleted the production database of a company without asking for permission and despite being told to not make any changes without asking for permission. (these screenshots are not under the Creative Commons license)



(3.11.1) Part of a screenshot obtained from the news article [538].



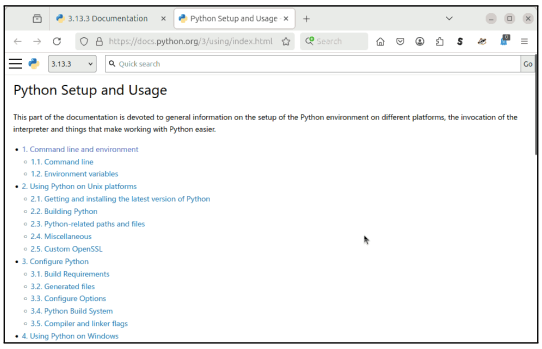
(3.11.2) A Part of a screenshot obtained from the news article [602].

Figure 3.11: Two screenshots of news articles describing an incident where a vibe coding AI deleted a whole partition of data instead of a directory. (these screenshots are not under the Creative Commons license)

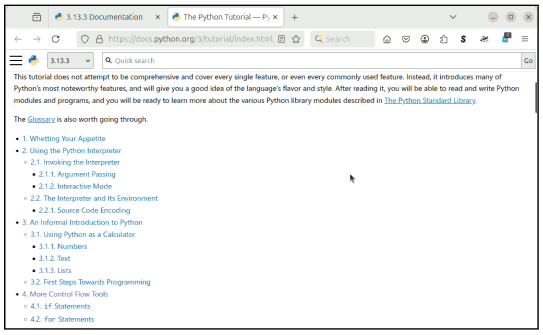
or other computing devises to do calculations in your later life and rarely compute things by hand or in your head. Yet, they did not give you a calculator right from the start. Even though they knew you would use calculating machines later. Why was that?

It was because you were supposed to learn how maths works. If a calculator was given to you, then you would have learned how to use a calculator. But your understanding of mathematics would be very very limited.

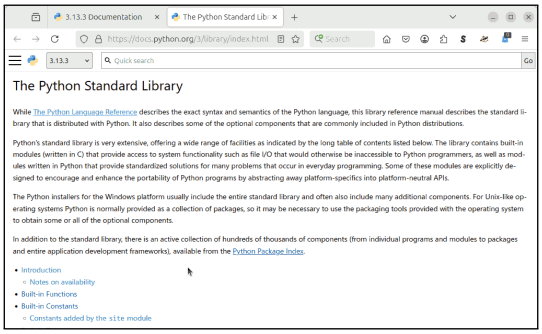
Right now, you are supposed to learn how programming works. If you would ask an AI to answer your programming questions, you would learn how to use the AI. But your understanding of programming would be very very limited. And with this very very limited understanding, you would be unable to, well, understand and verify the code produced by the AI.



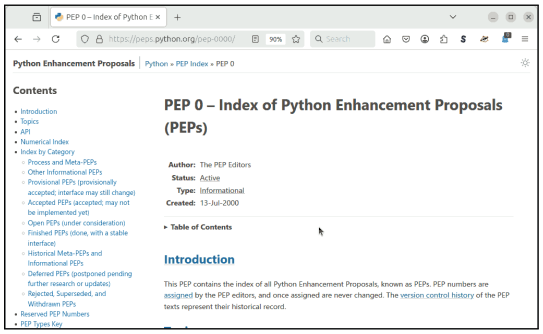
(3.12.1) The *Python Setup and Usage* page [465] at `https://docs.python.org/3/using`.



(3.12.2) The *The Python Tutorial* page [561] page at `https://docs.python.org/3/tutorial`.



(3.12.3) The *The Python Standard Library* page [560] at `https://docs.python.org/3/library`.



(3.12.4) The *Index of Python Enhancement Proposals (PEPs)* page [557] at `https://peps.python.org`.

Figure 3.12: Several important authoritative online resources on `Python`.

3.4.4 More Authoritative Sources of Information on Python

Before getting to the end of this section, let us also note: Besides being a source of authoritative information about `Python`, the official website `https://docs.python.org/3` also offers a variety of other useful information. You may have already used the *Python Setup and Usage* page [465] at `https://docs.python.org/3/using`, screenshotted in Figure 3.12.1, when installing `Python` on your system.

The *The Python Tutorial* page [561] page at `https://docs.python.org/3/tutorial` and shown in Figure 3.12.2 offers several useful tutorials. Actually, it describes many of the techniques that we discuss in this book in much depth and very nicely, probably even better than this book. I very strongly urge you to also read this website.

The *The Python Standard Library* page [560] at `https://docs.python.org/3/library` depicted in Figure 3.12.3 contains the specification of all the standard functions that ship with `Python`. It is therefore a very useful tool when checking what a function *actually* does, to complement our learned knowledge.

Finally, the *Index of Python Enhancement Proposals (PEPs)* page [557] at `https://peps.python.org` shown in Figure 3.12.4 contains all changes and additions to the `Python` language since roughly the year 2000. We can find many of the language utilities that we are using in this book defined in a so-called *Python Enhancement Proposal (PEP)*.

3.4.5 Summary

So what did we learn in this section? We learned that an important tool for a programmer is the official documentation of the programming language that they are using and of the tools that they employ. We can trust the documentation. It is an important skill to be able to sit down, concentrate, and read such documentation. If you have a certain question or task, it is very useful if you are able to work your way through a large body of text, be it some documentation or some standard, to identify which information is relevant for your situation, to ingest and understand this information, and to determine

the actions you need to take to bring the task to a successful end.

That being said, it is not always possible to easily find the information that you are looking for in the documentation. Luckily, there exist other useful tools that you can apply. You can use search engines, you can read in internet forums like *Stack Overflow* [530], and you can ask your questions to an AI. All of these tools can guide you to the right answer.

But remember that all non-authoritative sources provide answers that, ultimately, are grounded on prior human experience and training. On the one hand, humans can make errors and not even know that they made them. On the other hand, non-authoritative sources might be outdated. We always need to check these answers by circling back to the authoritative documentation, because there is no guarantee that the answeres are correct or up-to-date.

As an example of searching information using AI tools, we looked for a function that rounds floating point numbers towards positive infinity. We did not know what this function was called (or whether it even exists). The AI told us that it was called `ceil`. We got the exact same information by using a search engine and by using a community portal. So this works. Regardless how we got that information, we would not just use the function `ceil` in our code directly. We would go back to the official Python documentation and search for `ceil`. Only after checking the documentation, we would have clarity and confidence that we will produce the right code.

3.5 Boolean Values

Before, we already mentioned comparisons and their results, which can either be `True` or `False`. These two values constitute another basic datatype in Python: `bool`. The two values of this type are fundamental for making decisions in a program, i.e., for deciding what to do based on data.

3.5.1 Comparisons

In the sections on `floats` and `ints`, we learned how to do arithmetic with real and integer numbers. You have learned these operations already in preschool. However, before you learn to calculate with numbers, you learned how to *compare* them. If we compare two numbers, the result is either `True`, if the comparison works our positively, or `False`, if it does not. Python supports six types of comparison:

- equal: $a = b$ corresponds to `a == b`,
- unequal: $a \neq b$ corresponds to `a != b`,
- less-than: $a < b$ corresponds to `a < b`,
- less-than or equal: $a \leq b$ corresponds to `a <= b`,
- greater-than: $a > b$ corresponds to `a > b`, and
- greater-than or equal: $a \geq b$ corresponds to `a >= b`.

How to use these operators is illustrated in Listing 3.11. It shows that `6 == 6` yields `True`, while `6 != 6` yields `False`. The expression `6 > 6` gives us `False`, but `6 >= 6` is `True`. `6 < 6` is also `False` while `6 <= 6` is, of course, `True`. While `5 > 6` is not `True`, `6 > 5` is. It is also possible to compare floating point numbers with integers and vice versa. `5.5 == 5` is `False`, while `5.0 == 5` is `True`.

When comparing an integer with a floating point number, one might expect that one is converted to another. This, interestingly, does not seem to be the case. In Listing 3.12, we can see that Python handles comparisons of integers and floating point numbers more subtle. The comparisons work for small integers and floats without issue. If the numbers get larger, we first run into the issue with limited precision of the `float` datatype. It cannot store more than 15 or 16 decimal digits (depending on the weather), and thus `1e300` is not actually 10^{300} as one might expect. Instead only the first 16 zeros are there, the remaining digits can be considered as undefined or garbage, as their value depends on whatever results from the internal binary representation.

We now try to compare 10^{400} to `1e400`. Knowing the limits of `float`, we recall that writing `1e400` is essentially equivalent to writing `inf` (if that constant was imported from the `math` module). The comparison yields `False`, as it should. To get there, however, neither is `10 ** 400` converted to a

`float`, nor is `1e400` converted to an `int`. Both would net us an `OverflowError`. Thus, we find that the `float` datatype does not "infect" comparisons and that `Python` is actually quite elegant here.

Comparisons can also be chained: `3 < 4 < 5 < 6` is `True`, because `3 < 4` and `4 < 5` and `5 < 6`. `5 >= 4 > 4 >= 3`, however, is `False`, because while `5 >= 4` and `4 >= 3`, it is not true that `4 > 4`.

If we check the type of `True`, it yields `<class 'bool'>`, i.e., `bool`. The result of the expression `5 == 5` is a `bool` as well.

When talking about comparisons, there is one important, counter-intuitive exception to recall: The `nan` floating point value from Section 3.3.5. In Listing 3.9, we learned that `nan == nan` is `False` and `nan != nan` is `True`. This is the only primitive value (to my knowledge) which is not equal to itself.

3.5.2 Boolean Operators

The most common operations with Boolean values are the well-known Boolean logical operators `and`, `or`, and `not`. Their truth tables are illustrated in Figure 3.13.

- A Boolean conjunction, i.e., `and`, is `True` if and only both of its operands are also `True` and `False` otherwise, as shown in Figure 3.13.1.
- A Boolean disjunction, i.e., `or`, is `True` if at least one of its two operands is `True` and `False` otherwise, as shown in Figure 3.13.2.
- The Boolean negation, i.e., `not`, is `True` if its operand is `False`. Otherwise, it is `False`, as shown in Figure 3.13.3.

In Listing 3.13 we explore these three operators in the `Python` console. You can see that the operations can be used exactly as in the truth tables and yield the expected results. Additionally, you can of course nest and combine Boolean operators using parentheses. For example, `(True or False)and ((False or True)or (False and False))` resolves to `True and (True or False)`, which becomes `True and True`, which ultimately becomes `True`. You can also combine Boolean expressions like comparisons using the logical operators: `(5 < 4)or (6 < 9 < 8)` will be resolved to `(False)or (False)`, which becomes `False`.

3.5.3 Summary

Boolean values are very easy to understand and deal with. They can either be `True` or `False`. They can be combined using `and`, `or`, and `not`. And, finally, they are the results of comparison operators. Later, we will learn that Boolean decisions form the foundation for steering the control flow of programs.

Listing 3.11: The results of basic comparisons are instances of `bool`.

```
Python 3.14.10 (main, Oct 27 2026, 09:35:01) [GCC 15.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
# Compare a number to itself.
>>> 6 >  6
False
>>> 6 >= 6
True
>>> 6 == 6
True
>>> 6 <= 6
True
>>> 6 <  6
False
>>> 6 != 6
# Compare a smaller to a larger number.
False
>>> 5 >  6
False
>>> 5 >= 6
False
>>> 5 == 6
False
>>> 5 <= 6
True
>>> 5 <  6
True
>>> 5 != 6
# Compare a larger to a smaller number.
True
>>> 6 >  5
True
>>> 6 >= 5
True
>>> 6 == 5
False
>>> 6 <= 5
False
>>> 6 <  5
False
>>> 6 != 5
True

>>> 5.5 == 5  # compare a float value with an int: False due to ".5"
False
>>> 5.0 == 5  # compare the float 5.0 with the int 5: True!
True

>>> 3 <  4 < 5 <  6  # Chained comparison: True, all comparisons hold.
True
>>> 5 >= 4 > 4 >= 3  # Chained comparison: False, as 4 is not ">" 4.
False

>>> type(True)      # type(x) returns the datatype of x.
<class 'bool'>
>>> type(5 == 5)    # type(5 == 5) is the same as type(True), i.e., bool.
<class 'bool'>
```

Listing 3.12: Comparing integer and floating point numbers.

```
Python 3.14.10 (main, Oct 27 2026, 09:35:01) [GCC 15.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
# Compare a relatively small integer to a float.
>>> 10 ** 20 == 1.0e20  # This gives us True.
True
>>> 10 ** 20 == 1.1e20  # This gives us False.
False

# Compare a larger integer to a float.
>>> 10 ** 300 == 1e300  # This, interestingly, yields False.
# The reason is the limited precision of floats of 15 to 16 digits.
False
>>> int(1e300)  # Only 16 zeros are effectively there in this float.
1000000000000000052504760255204420248704468581108159154915854115511802457982
    ↪ 8908195786371375080447864043704443832883878176942523235360430575644792
    ↪ 2184786706982848387200926575803737830233794788090059368953234970799945
    ↪ 5081119038967640880074652742780142494579258788820056842838115669472196
    ↪ 6386865459400540160

# This must be False, because 1e400 is effectively inf.
>>> 10 ** 400 == 1e400
False
>>> 1e400  # <-- This prints "inf"
inf

# In the above comparison, neither is the left-hand integer is
# converted to a float, because:
>>> float(10 ** 400)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
OverflowError: int too large to convert to float

# Nor was the right-hand float converted to an int, because:
>>> int(1e400)
Traceback (most recent call last):
  File "<console>", line 1, in <module>
OverflowError: cannot convert float infinity to integer
```

a	b	a and b
False	False	False
False	True	False
True	False	False
True	True	True

(3.13.1) The truth table for the logical conjunction (*logical and*): `a and b`.

a	b	a or b
False	False	False
False	True	True
True	False	True
True	True	True

(3.13.2) The truth table for the logical disjunction (*logical or*): `a or b`.

a	not a
False	True
True	False

(3.13.3) The truth table for the logical negation (*logical not*): `not a`.

Figure 3.13: The truth tables for the Boolean operators `and`, `or`, and `not`.

Listing 3.13: The `bool` values can be combined with the Boolean logical operators `and`, `or`, and `not`.

```
Python 3.14.10 (main, Oct 27 2026, 09:35:01) [GCC 15.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
# Conjunction (and) is True iff both operands are True.
>>> False and False
False
>>> False and True
False
>>> True  and False
False
>>> True  and True
True

# Disjunction (or) is True iff at least one operand is True.
>>> False or False
False
>>> False or True
True
>>> True  or False
True
>>> True  or True
True

# Negation (not) is True iff its operand is False.
>>> not True
False
>>> not False
True

# Operators can be combined and grouped with parentheses.
>>> (True or False) and ((False or True) or (False and False))
True

# Operators can accept any expression of result type bool.
>>> (5 < 4) or (6 < 9 < 8)
False
```

3.6 Text Strings

The fourth important basic datatype in Python are text strings. Text strings are sequences of characters of an arbitrary length. In Python, they are represented by the datatype `str`. Indeed, we have already used it before, even in our very first example program back that simply printed `"Hello World"` in Listing 2.1 in Section 2.3. `"Hello World"` is such a text string.

3.6.1 Basic String Operations

As Listing 3.14 shows, there are two basic ways to specify a text string literal: Either enclosed by double quotes, e.g., `"Hello World!"` or enclosed by single quotes, e.g., `'Hello World!'`. The quotation marks are only used to delimit the strings, i.e., to tell Python where the string begins or ends. They are not themselves part of the string.

Best Practice 7

When defining a string literal, the double-quotation mark variant (`"..."`) may be preferred over the single-quotation mark variant (`'...'`). (The *Style Guide for Python Code* [594] does not give a recommendation, but maybe for consistency with the *Docstring Conventions* [232], see also Best Practice 8.)

One basic operation is string concatenation: `"Hello"+ ' ' + "World"` concatenates the three strings `"Hello"`, `" "`, and `"World"`. The result is `"Hello World"`. Notice how the single space character string is needed, because `"Hello"+ "World"` would just yield `"HelloWorld"`.

Strings are different from the other datatypes we have seen so far. They are *sequences*, meaning that they are linear arrays composed of elements. These elements are the single characters, which correspond to letters, numbers, punctuation marks, white space, etc.

One basic set of things that we can do with strings is to extract these single characters. First, we need to know the length of a string. For this purpose, we can invoke the `len` function: `len("Hello")` is `5`, because there are five characters in "Hello". `len("Hello World!")` would give us `12`, because `"Hello"` has five characters, `"World!"` has six characters (the `"!"` does count!) and there is the single space character in the middle, so $5 + 6 + 1 = 12$. By the way, the empty string `""` has length `0`, i.e., `len("")` yields `0`.

Knowing the length of a string, we can now safely access its single characters. These characters are obtained using the square brackets `[]` with the character index inbetween. The character indexes start at 0. Therefore, `"Hello"[0]` returns the first character of `"Hello"` as a `str`, which is `"H"`. `"Hello"[1]` returns the second character, which is `"e"`. `"Hello"[2]` returns the third character, which is `"l"`. `"Hello"[3]` gives us the second `"l"`. Finally, `"Hello"[4]` gives us the fifth and last character, namely `"o"`. If we would try to access a character outside of the valid range of the string, say `"Hello"[5]`, this results in an `IndexError`. We learn later what errors are and how to handle them – for now, it is sufficient to know that they will stop your program. And rightly so, because `"Hello"` has only five characters and accessing the sixth one is not possible and would have an undefined result.

Negative indices, however, are permitted: The index `-1` just means "last character", so `"Hello"[-1]` yields the string `"o"`. The index `-2` then refers to the "second-to-last character", so `"Hello"[-2]` gives us `"l"`. The third character from the end, accessed via index `-3`, is again `"l"`. `"Hello"[-4]` gives us `"e"` and `"Hello"[-5]` gives us `"H"`. Of course, using a negative index that would bring us out of the string's valid range, such as `-6`, again yields an `IndexError`.

We can also obtain whole substrings by using index ranges, where the inclusive starting index and the *exclusive* end index are separated by a `:`. In other words, applying the index `[a:b]` to a string results in all characters in the index range from `a` to `b - 1`. Doing this is called *string slicing*. `"Hello"[0:3]` yields a string composed of the characters at positions 0, 1, and 2 inside `"Hello"`, i.e., `"Hel"`. The end index is always excluded, so the character at index 3 is not part of the result. If we do `"Hello"[1:3]`, we get `"He"`, because only the characters at indices 1 and 2 are included. If we do not specify an end index, then everything starting at the start index until the end of the string is included. This means that `"Hello"[2:]` will return all the text starting at index 2, which is `"llo"`. We can also use negative indices, if we want. Therefore, `"Hello"[1:-2]` yields `"el"` Finally, we can also omit the start index, in which case everything until right before the end index is returned. Therefore, `"Hello"[:-2]` will

return everything from the beginning of the string until right before the second-to-last character. This gives us `"Hel"`. The slice `[1:8:2]` returns the substring starting at index 1 and ending before index 8, containing every second character. Applied to `"Hello World!"` it therefore yields `"el o"`. We will discussing slicing again later when discussing lists in Section 5.1.

Besides concatenating and extracting substrings, the `str` datatype supports many other operations. Here, we can just discuss the few most commonly used ones.

There are several ways to check whether one string is contained in another one. The first method is to use the `in` keyword. As Listing 3.15 shows, `"World"in "Hello World!"` yields `True`, as it checks whether `"World"` is contained in `"Hello World!"`, which is indeed the case. `"Earth"in "Hello World!"` is `False`, because `"Earth"` is not contained in `"Hello World!"`.

Often, however, we do not just want to know whether a string is contained in another one, but also *where* it is contained. For this, the `find` method exists. `"Hello World!".find("World")` tries to find the position of `"World"` inside `"Hello World!"`. It returns `6`, because the "W" of "World" is the seventh character in this string and the indices are zero-based. Trying to find the `"world"` in `"Hello World!"` yields `-1`, however. `-1` means that the string cannot be found.

We learn that most string operations are *case-sensitive*: The uppercase character "W" is different from the lowercase character `w`. Therefore, `"World"!= "world"` is `True`. Therefore, `"world"` cannot be found in `"Hello World!"`. We also learn that we need to be careful not to use the result of `find` as index in a string directly before checking that it is `>= 0`! As you have learned, `-1` is a perfectly fine index into a string, even though it means that the string we tried to find was not found.

Sometimes, the text we are looking for is contained multiple times in a given string. For example, `"Hello World!".find("l")` returns `2`, because "l" is the third character in the string. However, it is also the fourth character in the string. `find` accepts an optional second parameter, namely the starting index where the search should begin. `"Hello World!".find("l", 3)` begins to search for `"l"` inside `"Hello World!"` starting at index 3. Right at that index, the second "l" is found, so that `3` is also returned. If we search for another "l" after that, we would do `"Hello World!".find("l", 4)`, which returns index 9, identifying the "l" in "World". After that, no more "l" can be found in the string, so `"Hello World!".find("l", 10)` results in a `-1`.

While `find` returns the first occurrence of a string in the supplied range, we sometimes want the last occurrence instead. If we want to search from the end of the string, we use `rfind`. `"Hello World!".rfind("l")` gives us `9` directly. If we want to search for the "l" before that one, we need to supply an inclusive starting and exclusive ending index of the range to be searched. `"Hello World!".rfind("l", 2, 9)` searches for any "l" from index 8 down to 2 and thus returns `3`. `"Hello World!".rfind("l", 0, 3)` gives us `2` and since there is no "l" before that, `"Hello World!".rfind("l", 0, 2)` yields `-1`.

Another common operation is to replace substrings with something else. `"Hello World!".replace("Hello", "Hi")` replaces all occurrences of `"Hello"` in "Hello World" with "Hi". The result is `"Hi World!"` and `"Hello World! Hello!".replace("Hello", "Hi")` becomes `"Hi World! Hi!"`. It does not replace strings recursively, though. If you try to do `"Hello World!".replace("Hello", "Hello! Hello!")`, then the `"Hello"` is indeed replaced with `"Hello! Hello!"`. This means that the new string now contains `"Hello"` twice. These new occurrences are *not* replaced, so the result remains as `"Hello! Hello! World!"`.

Often, we want to remove all leading or trailing whitespace characters (spaces, newlines, tabs, ...) from a string. The `strip` function does this for us: `" Hello World! ".strip()` returns `"Hello World!".strip()`, i.e., the same string, but with the leading and trailing space removed. If we only want to remove the spaces on the left-hand side, we use `lstrip` and if we only want to remove those on the right-hand side, we use `rstrip` instead. Therefore, `" Hello World! ".lstrip()` yields `"Hello World! "` and `" Hello World! ".rstrip()` gives us `" Hello World!"`.

In alphabet-based languages, we usually can distinguish between uppercase characters, such as "H" and "W", and lowercase, such as "e", "l", and "o". The method `lower` transforms all characters in a string to lowercase and `upper` translates them to uppercase instead. Thus `"Hello World!".lower()` returns `hello world!` whereas `"Hello World!".upper()` yields `"HELLO WORLD!"`.

As the final examples of string functions, we can use `startswith` and `endswith`, respectively, to check whether a string begins or ends with another. `"Hello World!".startswith("hello")` is `False` whereas `"Hello World!".startswith("Hello")` is `True`. `"Hello World!".endswith("Hello")` is `False`, too, but `"Hello World!".endswith("World!")` is `True`.

Of course, these were just a small selection of the many string operations available in Python. You can find more in the official documentation [552].

Listing 3.14: Specifying string literals and indexing its characters.

```
Python 3.14.10 (main, Oct 27 2026, 09:35:01) [GCC 15.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> "Hello World!"  # Both the double-quote and the single-quote can be
'Hello World!'
>>> 'Hello World!'  # used to define strings, but " is preferred.
'Hello World!'
>>> "Hello" + ' ' + "World"  # Concatenate three strings.
'Hello World'

>>> len("Hello")  # Get the length of the string "Hello".
5

>>> "Hello"[0]  # First character, "H"
'H'
>>> "Hello"[1]  # Second character, "e"
'e'
>>> "Hello"[2]  # Third character, "l"
'l'
>>> "Hello"[3]  # Fourth character, "l"
'l'
>>> "Hello"[4]  # Fifth character, "o"
'o'
>>> "Hello"[5]  # Sixth character does not exist: Error!
Traceback (most recent call last):
  File "<console>", line 1, in <module>
IndexError: string index out of range

>>> "Hello"[-1]  # Last character: "o"
'o'
>>> "Hello"[-2]  # Second-to-last character: "l"
'l'
>>> "Hello"[-3]  # Third-to-last character: "l"
'l'
>>> "Hello"[-4]  # Fourth-to-last character: "e"
'e'
>>> "Hello"[-5]  # Fifth-to-last character: "H"
'H'
>>> "Hello"[-6]  # Sixth-to-last character does not exist: Error!
Traceback (most recent call last):
  File "<console>", line 1, in <module>
IndexError: string index out of range

>>> "Hello"[0:3]  # Substring starting at index 0 and ending *before* 3.
'Hel'
>>> "Hello"[1:3]  # Substring starting at index 1 and ending *before* 3.
'el'
>>> "Hello"[2:]   # Substring starting at index 2, until end.
'llo'
>>> "Hello"[1:-2]  # Substring starting at index 1, ending before -2.
'el'
>>> "Hello"[:-2]   # Everything before the second-to-last character.
'Hel'
>>> "Hello World!"[1:8:2]  # Every 2nd character between indexes 1 and 8
'el o'
```

Listing 3.15: Some more basic string operations.

```
Python 3.14.10 (main, Oct 27 2026, 09:35:01) [GCC 15.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> "World" in "Hello World!"  # check if string is contained: yes
True
>>> "Earth" in "Hello World!"  # check if string is contained: no
False

>>> "Hello World!".find("World")  # get index of substring
6
>>> "Hello World!".find("world")  # result -1 means "not found"
-1
>>> "Hello World!".find("l")      # search from start of string
2
>>> "Hello World!".find("l", 3)   # search starting at index 3
3
>>> "Hello World!".find("l", 4)   # search starting at index 4
9
>>> "Hello World!".find("l", 10)  # starting at index 10: not found
-1

>>> "Hello World!".rfind("l")        # search backwards from end
9
>>> "Hello World!".rfind("l", 2, 9)  # search backwards in range [2, 9)
3
>>> "Hello World!".rfind("l", 0, 3)  # search backwards in range [0, 3)
2
>>> "Hello World!".rfind("l", 0, 2)  # search backwards in range [0, 2)
-1

>>> "Hello World!".replace("Hello", "Hi")             # replace all
'Hi World!'
>>> "Hello World! Hello!".replace("Hello", "Hi")      # occurrences, but
'Hi World! Hi!'
>>> "Hello World!".replace("Hello", "Hello! Hello!")  # not recursively
'Hello! Hello! World!'

>>> " Hello World! ".strip()   # Remove leading and trailing spaces.
'Hello World!'
>>> " Hello World! ".lstrip()  # Remove leading spaces.
'Hello World! '
>>> " Hello World! ".rstrip()  # Remove trailing spaces.
' Hello World!'

>>> "Hello World!".lower()  # Convert to lower case.
'hello world!'
>>> "Hello World!".upper()  # Convert to upper case.
'HELLO WORLD!'

>>> "Hello World!".startswith("hello")  # Checking if string starts with
False
>>> "Hello World!".startswith("Hello")  # text is case-sensitive.
True

>>> "Hello World!".endswith("Hello")   # Checking if string ends with
False
>>> "Hello World!".endswith("World!")  # text is case-sensitive, too.
True
```

3.6.2 The str Function and f-strings

We now have learned the basic operations that can process strings. However, we did not yet learn one essential thing: How can we convert an object, like an `int` or `float` value to a `str`? Very often, we want to perform some complicated calculation that produces a number and then print this number.

In Python, you can convert many different objects to strings by passing them to the function `str`. As Listing 3.16 shows, passing the integer number `23` to `str`, i.e., invoking `str(23)`, yields the string `"23"`. Similarly, invoking `str` with the `float` value `23.5` produces the string `"23.5"`.

We can of course concatenate the results of such computations to form more comprehensive texts: `str(1 < 5)+ " is True and "+ str(1 + 5)+ "= 6."` produces `"True is True and 6 = 6."`, because `1 < 5` evaluated to `True`, which is converted to a string, and `1 + 5` gives us `6`, which, too, is converted to a string. However, converting data to strings like this is rather tedious.

Let us therefore discuss a very powerful and much more convenient gadget in Python's string processing toolbox: format strings, or f-strings for short [79, 210, 384, 521]. An f-string is like a normal string, except that it starts with `f"` instead of `"`. And, most importantly, it can contain other data and even complete expressions inside curly braces (`{...}`) which are then embedded into the string.

In Listing 3.17, we first consider the f-string `f"{12345678901234}is a really big integer."`[8]. This is basically a normal string, except that it contains an integer value. The opening curly brace ("{") right at its beginning signifies that some Python expression will begin which must be translated to a string. The actual expression, `12345678901234` is just a really big integer. The closing curly brace ("}") signifies the end of the expression. The Python interpreter evaluates all expressions inside such curly braces inside the f-string and then turns their results to strings (and removes the curly braces). This process is called *(string) interpolation*. `f"{12345678901234}is a really big integer."` simply becomes `f"12345678901234 is a really big integer."`

This first example was not very spectacular. But f-strings offer us several interesting means to format data. For example, we can add some formatting specifiers after the expression, separated by `:`. If our expression evaluates to an `int`, then we can specify a "thousand separator" after the `:`. In western languages, it is usually to group the digits of large numbers in groups of 3. In Chinese, they tend to use groups of 4 instead. To the best of my knowledge, we can only specify thousand separators, thought. This separator will be placed every three digits in the generated text. As an example, `f"12345678901234 with thousand separator ',' is {12345678901234:,}."` turns into `"12345678901234 with thousand separator ',' is 12,345,678,901,234."`.

Back in Section 3.2.2, we learned that integers can also be represented in hexadecimal and binary format. f-strings conveniently offer this out-of-the-box, we just need to add a `:x` or a `:b` format specifier to the expression, respectively. For example, the f-string `f"{12345678901234}in hexadecimal notation is {12345678901234:x}."` becomes `"12345678901234 in hexadecimal notation is b3a73ce2ff2."` and the

[8] The code that formats my inline Python examples sometimes eats spaces after { or }. Therefore, some of the strings presented here may look a bit off. In Listing 3.17, they are printed correctly, though.

Listing 3.16: The `str` function converts objects to strings.

```
1  Python 3.14.10 (main, Oct 27 2026, 09:35:01) [GCC 15.2.0] on linux
2  Type "help", "copyright", "credits" or "license" for more information.
3  # Converting values of some datatypes to strings.
4  >>> str(23)           # Convert the int 23 to the string "23".
5  '23'
6
7  >>> str(23.5)         # Convert the float 23.5 to the string "23.5".
8  '23.5'
9
10 >>> str(4) + str(5)   # First to-string conversion, then concatenation.
11 '45'
12
13 # Multiple to-string conversions and concatenations in one line.
14 >>> str(1 < 5) + " is True and " + str(1 + 5) + " = 6."
15 'True is True and 6 = 6.'
```

Listing 3.17: Python f-strings in action.

```
Python 3.14.10 (main, Oct 27 2026, 09:35:01) [GCC 15.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> f"{12345678901234} is a really big integer."
'12345678901234 is a really big integer.'
>>> f"12345678901234 with thousand separator ',' is {12345678901234:,}."
"12345678901234 with thousand separator ',' is 12,345,678,901,234."
>>> f"{12345678901234} in hexadecimal notation is {12345678901234:x}."
'12345678901234 in hexadecimal notation is b3a73ce2ff2.'
>>> f"{12345678901234} in 0x-hexadecimal notation is {12345678901234:#x}."
'12345678901234 in 0x-hexadecimal notation is 0xb3a73ce2ff2.'
>>> f"{1234567890} in binary notation is {1234567890:b}."
'1234567890 in binary notation is 1001001100101100000001011010010.'
>>> f"{1234567890} in 0b-binary notation is {1234567890:#b}."
'1234567890 in 0b-binary notation is 0b1001001100101100000001011010010.'

>>> f"{5} + {4} = {5 + 4}"
'5 + 4 = 9'

>>> from math import pi
>>> f"pi is approximately {pi}."
'pi is approximately 3.141592653589793.'
>>> f"pi rounded to two decimals is {pi:.2f}."
'pi rounded to two decimals is 3.14.'

>>> f"1/321 as percentage with 2 decimals is {1/321:.2%}."
'1/321 as percentage with 2 decimals is 0.31%.'
>>> f"1.2345533e4 with thousand separator and 1 decimal is {1.2345533e4:,.1
    ↪ f}."
'1.2345533e4 with thousand separator and 1 decimal is 12,345.5.'

>>> from math import sin
>>> f"sin(0.25pi) is approximately {sin(0.25*pi):.5f}."
'sin(0.25pi) is approximately 0.70711.'
>>> f"{1.2359817e12} is {1.2359817e12:e} and approximately {1.2359817e12:.3
    ↪ g}."
'1235981700000.0 is 1.235982e+12 and approximately 1.24e+12.'

>>> f"Single braces without expression: {{ and }}."
'Single braces without expression: { and }.'

>>> f"{5 + 4 = }"
'5 + 4 = 9'
>>> f"{23 * sin(2 - 5) = }"
'23 * sin(2 - 5) = -3.245760185376946'

>>> f"{1 < 5} is True and {1 + 5} = 6."  # The last str function example
'True is True and 6 = 6.'
```

f-string `f"{1234567890}in binary notation is {1234567890:b}."` is turned to `"1234567890 in binary notation is 1001001100101100000001011010010."`. We can also add the `0x` and `0b` prefixes to the generated number strings by using the `:#x` and `:#b` format specifiers instead. Let us use the same examples again but instead with the #-prefixes in the format specifiers. The hexadecimal variant `f"{12345678901234}in 0x-hexadecimal notation is {12345678901234:#x}."` then becomes `"12345678901234 in 0x-hexadecimal notation is 0xb3a73ce2ff2."`. The binary formatting string `f"{1234567890}in 0b-binary notation is {1234567890:#b}."` turns into `"1234567890 in 0b-binary notation is 0b1001001100101100000001011010010."`.

But f-strings allow us to do even more. They can contain complete Python expressions. `f"{5}+ {4}= {5 + 4}"` is evaluated to `"5 + 4 = 9"`.

We can also access constants and variables from within the f-string. Let us again import the constant π from the `math` module by doing `from math import pi`. We can print it as string by typing `f"pi is approximately {pi}."` into the Python console. The result is the string `"pi is approximately 3.141592653589793."`

In many situations, we do not want to see the full `float` value with all available significant digits. We can round it to two decimals by adding the `.2f` format specifier. Three digits would be `.3f` and so on... Anyway, `f"pi rounded to two decimals is {pi:.2f}."` gives us `"pi rounded to two decimals is 3.14."`.

Sometimes, we want to present a floating point value as a percentage. For example, $\frac{1}{321} = 0.003115265 \approx 0.31\%$. How can we convert such value to a nice string? By using the `:.2%` format specifier, which gives us a percentage with two decimals. `:.4%` would yield three decimals, and so on... Therefore `f"1/321 as percentage with 2 decimals is {1/321:.2%}."` turns into `"1/321 as percentage with 2 decimals is 0.31%."`.

We can also combine thousand separators and rounding to decimals. The format specifier `,.1f` will use the comma "," as thousand separator and print a floating point number rounded to one decimal. The f-string `f"1.2345533e6 with thousand separator and 1 decimal is {1.2345533e6:,.1f}."` thus is evaluated to `"1.2345533e6 with thousand separator and 1 decimal is 1,234,553.3."`.

Let us also insert a floating point arithmetic expression in an f-string. We therefore import the sine function `sin` from the `math` module via `from math import sin`. The f-string `f"sin(0.25pi)is approximately {sin(0.25*pi):.5f}."` computes the expression `sin(0.25*pi)` and presents its result rounded to five decimals via the `:.5f` format specifier. It therefore becomes `"sin(0.25pi)is approximately 0.70711."`.

We can also use the scientific notation. The format speciier `:e` simply prints a number in scientific notation back from Section 3.3.4. `:.3g` uses scientific notation, but only presents three digits, whereas `:.4g` would present four digits, and so on... `f"{1.2359817e12}is {1.2359817e12:e}and approximately {1.2359817e12:.3g}."` prints the same number three times. First, it is simply rendered as normal `float` value, then it is rendered in scientific notation, and then it is rendered in scientific notation, but rounded to three digits. This gives us `"235981700000.0 is 1.235982e+12 and approximately 1.24e+12."`.

The keen reader may have encountered a question at this stage: *"What do I do if I want to include a curly brace, say { or } inside my f-string?"* Well, if you include a single brace, it would be interpreted as start or end of an expression. If the following text makes sense as expression, it will be interpreted. If not, an error will occur. Either way, the curly brace would disappear. The solution is simple: If you need "{", just write "{{". It will be interpreted as a single "{" brace. If you need "}", just write "}}". It will be interpreted as a single "}" brace. `f"Single braces without expression: {{and }}."` simply becomes `"Single braces without expression: {and }."`

As the final example, let us look at a very cool feature of f-strings. Often, we want to print an expression together with its result [262]. Earlier, we wrote `f"{5}+ {4}= {5 + 4}"` is evaluated to `"5 + 4 = 9"`. What we actually wanted to print was the expression `5 + 4` together with its result. This can be done much easier: We can simply write `f"{5 + 4 = }"`, which, too is evaluated to `"5 + 4 = 9"` The more complex `f"{23 * sin(2 - 5)= }"` becomes `"23 * sin(2 - 5)= -3.245760185376946"`. One cool feature of this kind of expression-to-string conversation is that you can add the other format specifiers we discussed earlier after the `=`. For example, you could write `f"{23 * sin(2 - 5)= :.2f}"` and then the `.2f` format would be applied to the result of the expression, i.e., you would get `"23 * sin(2 - 5)= -3.25"` as the result of the extrapolation.

We can also convert our last example of the `str`-function, `str(1 < 5)+ " is True and "+ str(1 + 5)+ "= 6."`, to an f-string and get `f"{1 < 5}is True and {1 + 5}= 6."`. This not just looks nicer, but it is also faster.

In the original example, Python needs to perform three string concatenation operations via the `+` operator. It first creates the result of `str(1 < 5)`, which is a string. `" is True and "` is also a string by definition. The two are concatenated, which results in a third string. Then `str(1 + 5)` is evaluated, yielding the fourth string. The third string and this one are concatenated, resulting in a fifth string, and so on. This results in the creation of seven separate string objects in total. All of them need to be explicitly created, because this is what we demanded with the command.

When using an f-string, there is no explicit need to create the intermediate concatenation results as objects. There might just be some text in a buffer somewhere under the hood that is finally returned

as a string. Furthermore, f-strings are parsed when the code is loaded and evaluated during runtime, meaning that no runtime parsing is required [218]. There even exist specified Python bytecode for f-strings [520]. They also do not require an explicit or implicit function call and they can access the symbols of the current scope directly without needing them to be passed as arguments [218]. For all of these reasons, F-strings therefore are fast and efficient [237]. Having learned about them, you are now able to swiftly and elegantly convert the results of your computations to beautiful text.

3.6.3 Converting Strings to other Datatypes

While converting the other datatypes to strings is important to produce output, converting strings to the other datatypes is also important. If our programs accept input from the console, the command line, or from text files, we need to somehow translate these input strings to whatever datatype we actually need. Luckily, the datatypes we have discussed so far conveniently provide functions for doing so, and these functions are named like the datatypes themselves.

Listing 3.18 shows that we can convert the string `"1111"` to an integer `1111` simply by passing it to the function `int`. If we want to convert hexadecimal-formatted text such as the string `"0x1111"` instead, we need to tell the `int` function that the basis for conversion is 16. Doing `int("0x1111", 16)` yields `4369`, because $1 + 16^1 + 16^2 + 16^3 = 1 + 16 + 256 + 4096 = 4369$. Similarly, if we have a string in binary annotation, we pass 2 as second parameter to `int`. Calling `int("0b1111", 2)` returns `15`, because $1 + 2 + 4 + 8 = 15$.

Listing 3.18: Converting strings to other datatypes.

```
Python 3.14.10 (main, Oct 27 2026, 09:35:01) [GCC 15.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
# Here are some functions can be used to convert strings to other types.
>>> int("1111")         # Convert decimal number 1111 to int.
1111
>>> int("0x1111", 16)   # Convert hexadecimal number 1111 to int.
4369
>>> int("0b1111", 2)    # Convert binary number 1111 to int.
15

>>> float("2.233e4")    # fractional number in scientific notation and
22330.0
>>> float("0.1123")     # normal fractional number converted to float
0.1123
>>> float("inf")        # converted to the float value for infinity
inf
>>> float("nan")        # converted to the float value not-a-number
nan

# ----> WARNING: The function `bool` works differently! <----
# It performs a standard truth test, which returns `False` if its
# argument is False, 0, and empty string or sequence, or None.
# Otherwise, it returns True.
# So it is actually True for the string "False".
# "" is an empty string, but "False" is not an empty string....
>>> bool("True")        # "True" is converted to the bool value True.
True
>>> bool("False")       # "False" is converted to the bool value True!!!
True
>>> bool("")            # The empty string is converts to False.
False
>>> bool("blabla")      # Actually, any non-empty string becomes True.
True

# We now have learned how to use these functions with string parameters.
# However, they also work with parameters of other types.
# You can, for example, also do `float(1)` or `int(3.5)`.
```

We can also convert strings to floating point numbers by using the `float` function. This works for scientific notation (`float("2.233e4")` yields `22330.0`) as well as for "normal" floating point numbers (`float("0.1123")` yields the `float` `0.1123`). The `float` function also extends the special values that a floating point value can take one. Consequently, `float("inf")` gives us `inf` and `float("nan")` returns `nan`.

But the function `bool` works *differently* from what one might expect [84]. `bool("True")` indeed yields `True`. However, `bool("False")`, too, yields `True`. One would think that `bool("False")` should return `False`, but it does not. The reason is that `bool` does not compare the string it receives as argument to the string constants that `str(True)` and `str(False)` return. Instead, it performs a *truth value testing* procedure [84, 576].

You see, in Python, we can test many objects for their truth value. `True` and `False` obviously have truth values `True` and `False`, respectively. By default, another object has truth value `True` unless it offers an explicit conversion to `bool`[9] that yields `False` or has a length and that length is zero[10]. Other objects that have a truth value of `False` are numeric types which are zero, such as `0` and `0.0` as well as empty collections (which we learn about a bit later in Chapter 5) or the empty string.

Strings by themselves do not support an explicit conversation to the type `bool`. The function `bool` thus has to figure out the truth value by itself. However, strings do have a length, as we already learned. You can get the length via the `len` function. Now, `"False"` is not an empty string. It has a length greater than 0. `len("False")` gives us 5. Therefore, `"False"` has truth value `True`, meaning that `bool("False")` actually yields `True`!

The empty string, on the other hand, has lengh zero. Therefore, it has a truth value of `False`, i.e., `bool("")` gives us `False`. Any other string, for example `"blabla"` is also none-empty and thus has truth value `True`.

It should be noted that these conversion functions also work with other datatypes. For example, `float(0)` converts the integer `0` to the `float` value `0.0` and `bool(0)` gives us `False`. Anyway, you are now also able to convert strings to data that you can use as input for your computations.

3.6.4 Escaping Characters

First Time Readers and Novices: This section tells you how to include special characters in strings that you otherwise could not include, like quotation marks. You can skip this section and circle back to it when you need it.

The last example of f-strings brought up an interesting topic: "What do we do if we want to use a character in a string which we cannot use?" For example, if our string is delimited with double quotation marks `"`, then we cannot put the character "`"`" into it, because it would then be interpreted as the end of the string. On the other hand, if our string is delimited with single quotation marks `'`, then we cannot put the character "`'`" into it, because it would then be interpreted as the end of the string. Well, you can say, if I need a double quotation mark, then I will delimit my string with single quotation marks and vice versa. This is all good as long as you do not need *both* single and double quotation marks inside the string.

The answer to this problem is *escaping* [184]. The idea is very simple: If we need a certain quotation mark, then we simply put a backslash ("\") before it. The backslash tells the Python interpreter that the next character should be considered as a normal character and not be interpreted as any special character, like a string delimiter.

In Listing 3.19, we present several strings with escape sequences. For clarity reasons, we pass them to the `print` function to output them, which means that they show up undelimited in the console. A double quotation mark can be printed as `print("\"")`, i.e., as a string which is delimited and that then contains the escape sequence \". It then shows up as `"` in the output. A single quotation mark can be printed as `print("\'")`, i.e., via the escape sequence \'. It appears as `'` in the output.

If the use the backslash character "\" to escape characters which may otherwise have some special meaning ... then what do we do if we need a backslash inside of a string? Easy: We escape it. The escape sequence "\\" is converted to a single backslash and `print("\\")` writes `\` to the output.

[9] Via the dunder method `__bool__`, see later in Section 13.6.
[10] Via the dunder method `__len__`, see later in Section 13.6.

Listing 3.19: Escaping special characters in Python strings.

```
Python 3.14.10 (main, Oct 27 2026, 09:35:01) [GCC 15.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("A single quotation mark: '")    # We can simply use the
A single quotation mark: '
>>> print('A double quotation mark: "')    # "other" quotation mark...
A double quotation mark: "

# But now we need to use escaping.
>>> print("A single (') and a double (\") quotation mark.")
A single (') and a double (") quotation mark.

>>> print("\"Hello World!\"")  # We simply put a backslash (\) before
"Hello World!"
>>> print('\'Hello World\'')    # the forbidden character.
'Hello World'
>>> print("\\")                 # Backslashes are escaped like this, too.
\
>>> print("\"\'\\\"\"")         # This becomes "'\""
"'\""

>>> print("Hello\nWorld!")      # \n is a newline.
Hello
World!
>>> print("Hello\r\nWorld!")    # On Windows systems, \r\n is common.
Hello
World!

>>> print("The horizontal tab is like a bigger space: '\t'.")
The horizontal tab is like a bigger space: '    '.

# A backslash before a line break removes the line break.
>>> print("Hello \
... World!")
Hello World!
```

Knowing these sequences, we can now try to `print("\"\'\\")`. The result printed to the output then is `"'\`.

Another situation where escape sequences are nice is when we want to have strings that span multiple lines. The newline sequence "\n" represents a "newline character" which causes the console to skip to the next line. `print("Hello\nWorld!")` will first print `Hello`, then end the current line and begin a new line, and then print `World!`. Notice that the newline character sequence "\n" is used in Linux and similar systems, whereas Microsoft Windows uses "\r\n". Under Python, both always work, regardless under which operating system you are working, and you should *always* use "\n". However, only for the sake of completeness, I include the example `print("Hello\r\nWorld!")` as well, which produces the same output as `print("Hello\nWorld!")`.

You probably have pressed the tabulator key ⇤⇥ on your keyboard at some time in the past. It produces something like a "longer space". If you want to include a horizontal tabulator in a string, the escape sequence "\t" is your friend: `print("The horizontal tab is like a bigger space: '\t'.")` yields `The horizontal tab is like a bigger space: '    '.`.

Finally, a backslash can also escape an actual newline in your string. If you have a string that is too long to write on a single line but you do not want to have a linebreak inside the actual string, you can simply put a backslash, hit ↵, and continue the typing the string. The linebreak will then be ignored entirely. Therefore, if you print `print("Hello\`, hit ↵, and then continue to write `World!")`, this produces the output `HelloWorld!`.

Escape sequences allow us to write arbitrary text in strings. We already learned the sequences "{{" and "}}" that were designed for f-strings only. The backslash-based escape sequence we discussed in

Listing 3.20: Examples of multi-line strings.

```
Python 3.14.10 (main, Oct 27 2026, 09:35:01) [GCC 15.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("""This is a multi-line string.
... I can hit enter to begin a new line.
... This linebreak is then part of the string.""")
This is a multi-line string.
I can hit enter to begin a new line.
This linebreak is then part of the string.

>>> print(f"""This is a multi-line f-string.
... It can contain expressions like {(5 ** 7) / 3123:.2f}
... or {(5 < 9) and ((5.5 / 3) < (2 * 11))}.""")
This is a multi-line f-string.
It can contain expressions like 25.02
or True.
```

this section work for both f-strings and normal strings.

3.6.5 Multi-Line Strings

Before we discussed that strings in Python are delimited either by `"` or `'` on each side. However, we can actually also delimit them with *three quotation marks* on each side, i.e., with either `"""` or `'''`. Such string delimiters are used for multi-line strings. In such strings, you can insert linebreaks by hitting ↵ completely normally. You can use the escape sequences from the previous section as well. The main use case are docstrings, which we will discuss later, see, e.g., Best Practice 23.

Best Practice 8

When defining a multi-line string literal, the double-quotation mark variant (`"""..."""`) is preferred over the single-quotation mark variant (`'''...'''`) [232, 594].

Listing 3.20 shows what happens if we print such a multi-line string. We first create the string by writing the three lines `This is a multi-line string.`, `I can hit enter to begin a new line.`, and `This linebreak is then part of the string.`. The first line begins with `"""` and the last one ends with `"""` as well. Passing this text to the `print` function, well, prints exactly this three-line string.

We can also have multi-line f-strings. These then simply start with `f"""`. The example in Listing 3.20 presents such a multi-line f-string with two expressions for (string) interpolation which spans over three lines.

3.6.6 Unicode and Character Representation

First Time Readers and Novices: This section is for readers who want to learn how text is mapped to numbers in order to store it in a computer. First-time readers can safely skip over it.

The memory of our computers basically stores chunks of bits of certain fixed sizes, say, bytes that are composed of 8 bit each. Usually, these are interpreted as integer numbers. While Python supports arbitrarily large integers, usually we deal with integers composed of 8 bytes, i.e., 64 bits. The `float` datatype in Python is also usually 8 bytes large, but these are interpreted differently in order to facilitate fractional numbers (see, e.g., Figure 3.3). But how does this work with text?

Well, by representing each character of a text as a number. A `str` is then nothing but a list of these numbers. The system then knows how to interpret these numbers as characters. Maybe the most well-known historical mapping is ASCII [7, 583], which used seven bits per character. It therefore contained only latin characters, punctuation marks, numbers, and some control characters (like the newline and tab characters we learned when discussing string escaping). Since different languages use different characters – and more than $2^7 = 128$ in total – many different mappings have historically

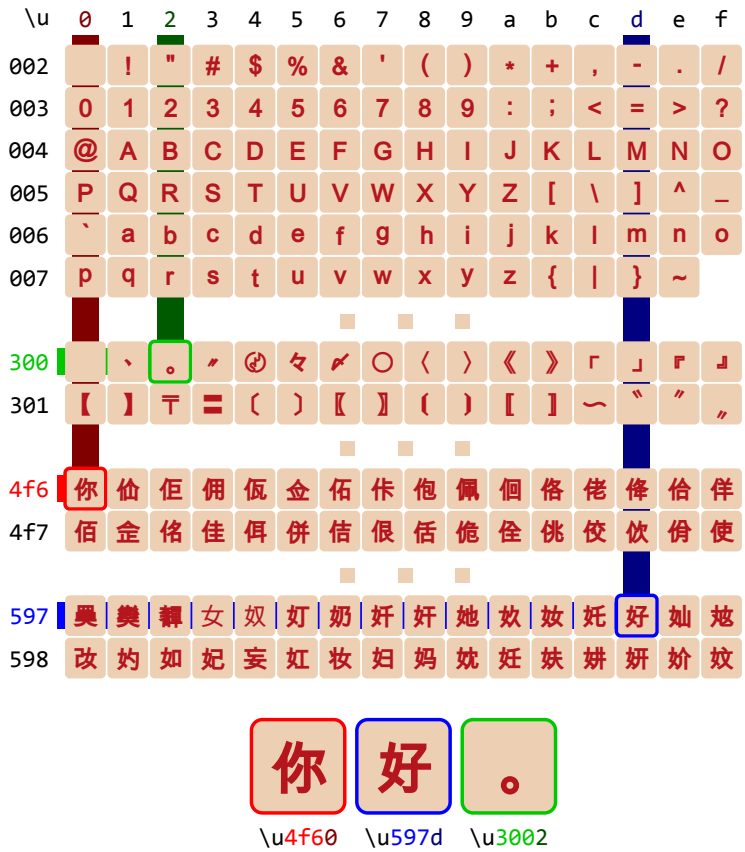


Figure 3.14: A subset of the Unicode character table including the Basic Lating characters as well as some Simplified Chinese characters (简体中文) [664].

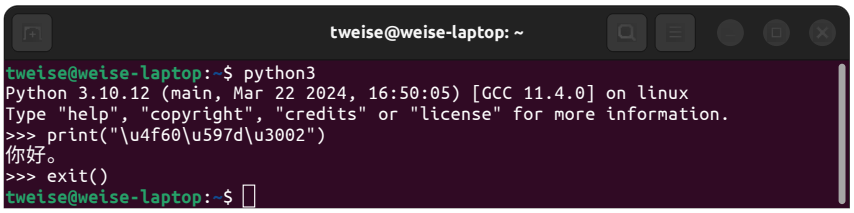


Figure 3.15: "Hello", i.e., "你好。" in Simplified Chinese (简体中文) [664] and entered via Unicode escaped string.

evolved and still exist today. In China, different mappings specialized to Chinese characters exist additionally, including the historical GB 2312 [662], GBK [665], or the newer GB 18030 [663]. Today, the vast majority of computers and systems understand one common standard that covers all languages: Unicode [285, 565, 582], the most frequently used mapping of characters to numbers. Therefore, Python uses Unicode as well.

Figure 3.14 illustrates a subset of the Unicode code table, including the Basic Latin characters, which are basically still compatible with ASCII, and some Simplified Chinese characters (简体中文) [664]. Most Unicode characters can be identified by a number represented as four hexadecimal digits (mentioned back in Section 3.2.2). The rows Figure 3.14 are annotated with the first three of these digits and the columns with fourth and last hexadecimal digits.

Python allows us to enter Unicode characters via a special escape code starting with `\u` followed by these four digits. This is very useful. Imagine that you are sharing a program file with some colleagues. Depending on how their computer encodes text, the Basic Latin characters are usually interpreted correctly. But some computers may misinterpret Unicode text as something else because the mix up the file encoding. If we use the `\u`-based escape, then we can represent *any* character as Basic Latin text sequence. It is also useful if we want to, e.g., enter Chinese text on a machine that does not have an input method editor (IME) or other corresponding tools, or text in any other kind of language where we do not have corresponding keys on the keyboard (see, e.g., Listing 4.3 later on).

Anyway, in Figure 3.15, we use the information obtained in Figure 3.14 to print the Chinese text "你好。" standing for "Hello." and pronounced as "Nǐ hǎo." as a unicode-escaped string. We found that the character for "你" has unicode number 4f60, "好" has 597d, and the big period "。" has 3002. The string `"\u4f60\u597d\u3002"` then corresponds to the correct Chinese text "你好。".

3.6.7 Summary

Strings, i.e., instances of `str`, represent text. Text is obviously an important element of any program, not just because programs are usually written as text, but because they often receive text as input and send text back as output as well. Strings are therefore an essential element of programming. They are basically lists of characters. We can index them, i.e., extract portions of text, we can search inside a string to check whether and where it contains a certain substring, and we can manipulate strings, e.g., by replacing substrings.

It is also important that we can transform data such as `int`s, `float`s, or `bool`s to strings. Because we actually *always* print them as strings to the console. The user cannot interpret the binary representation of such data, they only want text. While the function `str` can convert many different types of objects to strings, we often want to combine several different pieces of information to an output text. f-strings are the tool for that. They can render almost arbitrary data as nicely formatted strings and take care of things such as rounding or inserting thousand separators.

Converting strings to the other datatypes that we have discussed so far can, conveniently, be done by using functions of the same names: The function `int` converts its argument string to an instance of `int`. The function `float` converts its argument string to an instance of `float`. But the function `bool` works very differently! `bool("True")` and `bool("False")` both yield `True`, while `bool("")` gives us `False`. This is quite dangerous.

Sometimes we want to include characters in our strings that are dodgy. For example, if our string is delimited by `"` marking its begin and end, inserting such a `"` inside the string would be awkward. Indeed, the Python interpreter would think that it marks the end of the string and then confuse the `"` marking the actual end as the beginning of a new string. Escape sequences solve this problem: We would just write `\"` instead of `"` if we want to insert a double quotation mark inside a string. Multi-line strings solve the problem that we sometimes want to enter text as strings that, well, spans multiple lines.

Finally, we learned that strings internally are Unicode character sequences. Unicode is a standard that maps the characters of all common languages to numbers. We can look up the number corresponding to a character in a Unicode code table. Usually, these are four-digit hexadecimal numbers, which we can then use with the special `\u` escape codes in Python. This allows us to basically represent arbitrary text from arbitrary languages using only the Basic Latin characters.

3.7 None

The last simple type we talk about is the `NoneType` and its one single value: `None`. Now you already learned the type `bool` which can take on only two different values, `True` and `False`. You have also learned that the type `float` has a special value called "Not a Number" and written as `nan`. So let's approach this new type from this direction.

`None` is used in scenarios where we want to specify that something does not have any value. It is not an integer, float, string, or bool. `None` is not equivalent to `0`, it is not equivalent to `nan`, and also different from the empty string `""`. It is just nothing.

Listing 3.21 illustrates some of the things we can do with `None`. If we write `None` into the Python console, then nothing happens. In the past, we just wrote values, such as `34` and, after we hit ↵, they would be printed again. Not so `None`. If we want to print `None`, we have to force it by using the function `print`. `print(None)` then indeed prints `None`.

The value `None` has many use cases. Unfortunately, most of them we have not yet learned about, so we will have to circle back to this later. For now, simply imagine that you want to write a program that step-by-step computes data. Variables that have not yet been computed could be set to `None` to signify that. If we had a `float`, we could try to set it to `nan` instead, but `nan` could also be the result of a computation. `None` would be much clearer, as no arithmetic calculation could ever return `None`.

For this to work we need to be able to check whether a value is `None`. We can use the `==` operator for it, but its use is discouraged [594]. We do it here anyway, just for demonstration purposes, because

Listing 3.21: Examples of using the value `None`.

```
Python 3.14.10 (main, Oct 27 2026, 09:35:01) [GCC 15.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> None                          # Prints nothing.
>>> print(None)                    # Prints "None".
None

# The use the equality comparisons via == is discouraged if None may
# be involved. We do it here anyway, just to see what it does.
>>> 1 == None                      # False
False
>>> "Hello World!" == None         # also False
# Different from 'nan == nan', which is False, 'None == None' is...
False
>>> None == None                   # ...True
True

# It is encouraged to use 'is' instead, which checks object identity
# (not equality like ==).
>>> 1 is None                      # False
<console>:1: SyntaxWarning: "is" with 'int' literal. Did you mean "=="?
False
>>> "Hello World!" is None         # False
<console>:1: SyntaxWarning: "is" with 'str' literal. Did you mean "=="?
False
>>> None is None                   # True
True

>>> print(print("Hello World!"))   # Prints "Hello World!" and "None".
Hello World!
None
>>> type(None)                     # None is the only instance of NoneType
<class 'NoneType'>
```

you are already very familiar with `==`. `1 == None` gives us `False` and so does `"Hello World!"== None`. `None == None`, however, is `True`. This is a bit interesting, because `nan == nan` would give us `False`, but then again, `nan` means "undefined" and `None` means "nothing."

Best Practice 9

Comparisons to singletons like `None` should always be done with `is` or `is not`, never the equality operators `==` or `!=` [594].

As stated above, the right way to compare values if `None` may be involved is using the operator `is` [594]. `is` is a bit similar to `==`, but instead of comparing whether the contents of an object are the same, it compares whether two values reference the same object. As expected `1 is None` gives us `False` and so does `"Hello World!"is None`. `None is None`, however, is `True`.

Also, functions that do not return any result do, in fact, return `None`. We already learned some functions. The function `len`, for example, can be used to compute the length of a string and then returns this value as `int`. The function `log` from the `math` module computes the natural logarithm and returns a `float`. `print`, however, just prints its parameter to the console ... and returns *nothing*. Well, not *nothing*. It returns `None`! We can test this by doing `print(print("Hello World!))`. The inner `print("Hello World!)` will print the string `"Hello World!"`. It will return `None`. This value `None` is then passed into the outer `print` function, which thus essentially does `print(None)` and, thus, prints `None` to the console. We therefore see two lines of text appear, first `Hello World` and then `None`.

The type of `None` can be determined using the `type` function and, indeed, is `NoneType`.

A final use case for `None` is as default value for optional parameters of functions. But we will learn this much later. And thus, we conclude our short section about the value `None` at this point.

3.8 Summary

This section was far more exhausting than what I initially anticipated. I admit that. But I think we now have a solid understanding of the simple datatypes that `Python` offers to us and what we can do with them. We learned about integer numbers and how we can do arithmetic with them. We learned about floating numbers, which can represent fractions and which are limited in their precision. Boolean expressions, such as comparisons, can either be `True` or `False`. Strings are used to represent text and we learned how to convert the other types to and from them. Finally, `None` represents the absense of any value. Equipped with this knowledge, we now can embark to learn how to write programs that compute with these datatypes.

Chapter 4

Variables

We already learned different simple types in Python as well as basic expressions, such as mathematical formulas or how to work with strings. We are still relatively far from writing programs, though. Basically, all we can do is expressions that fit on a single line.

For more complicated computations, we want to store and modify values. For this purpose, variables exist.

4.1 Defining and Assigning Variables

As in mathematics, a variable in Python is basically a name with a value assigned to it. You can define a variable and assign its value by writing `name = value`. Here, `name` is the name of the variable and `value` be the value that we want to assign to that name. If we want to access the value that was stored, we can just use the `name` instead. You can write `name` in an arbitrary expression and Python then just uses `value` instead. Indeed, you have seen this before, when we used the variables `pi`, `e`, `inf`, and `nan` that we imported from the module `math`. You can also change which value is stored under a given name by assigning another value to it, e.g., by doing `name = new_value`.

With this, we can now store intermediate results and use them in later computation steps. This allows us, for the first time, to write programs that perform computations in multiple steps and that consist of multiple lines of code. Python programs are stored as text files with the suffix `.py`, e.g., `hello.py`.

In the very moment we begin to create such files, two things happen: First, our code will immediately become much more complex. Second, our code can be reused, i.e., executed multiple times. It can be reused by us, now or in ten years, or shared with others. This changes the quality of programming entirely. Until now, the Python interpreter was basically a fancy calculator. Now our programs become tools that could be used hundreds of times. They also could become building blocks or bricks of giant architectures.

Therefore, from the very start when we actually begin to write program files, it becomes important that we clearly document what we do and why. For this purpose, we will never just write code – we will always write comments giving explanations of what our code does. From the very beginning we must train ourselves to exercise proper discipline.

Best Practice 10

Comments help to explain what the code in programs does and are a very important part of the *documentation* of code. Comments begin with a `#` character, after which all text is ignored by the Python interpreter until the end of the current line. Comments can either occupy a complete line or we insert two spaces after the last code character in the line and then start the comment [594].

So we now learn two things together: Using variable assignment and commenting our code. Because variable assignment is the most fundamental step in writing programs and because programs without comments are *wrong*. By the way, this is how we do it in this book: We learn new programming concepts, but try to always intersperse important best practices.

Listing 4.1: A Python program showing some examples of variable assignments. (stored in file assignment.py; output in Listing 4.2)

```
# We define a variable named "int_var" and assign the int value 1 to it.
int_var = 1

# We can use the variable int_var in computations like any other value.
print(2 + int_var)  # This should print 2 + int_var = 2 + 1 = 3.

# We can also use the variable in f-strings.
print(f"int_var has value {int_var}.")  # prints 'int_var has value 1.'

# We can also change the value of the variable.
int_var = (3 * int_var) + 1  # int_var = (3 * 1) + 1 = 4
print(f"int_var is now {int_var}.")  # prints 'int_var is now 4.'

float_var = 3.5  # Ofcourse we can also use floating point numbers.
print(f"float_var has value {float_var}.")  # 'float_var has value 3.5.'

new_var = float_var * int_var  # new_var = 3.5 * 4 = 14.0 <- a float!
print(f"{new_var = }.")
```

↓ `python3 assignment.py` ↓

Listing 4.2: The stdout of the program assignment.py given in Listing 4.1.

```
3
int_var has value 1.
int_var is now 4.
float_var has value 3.5.
new_var = 14.0.
```

4.1.1 A Simple Example of Variable Assignment and Comments in the Code

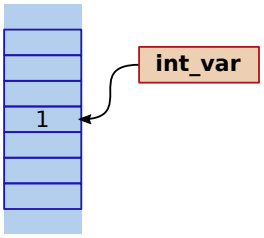
So let's get to the subject of variable assignments with some examples. Listing 4.1 shows the source code of our very first commented program. This program does not do anything useful, but it illustrates how variables can be used. It begins by assigning the `int` value `1` to a variable named `int_var`. We could have chosen any other name for the variable as well, e.g., `my_value`, `cow`, `race_car`, as long as it does not contain special characters like spaces or line breaks. But we chose `int_var`. The `=` assigns the value `1` to `int_value`. As Figure 4.1.1 illustrates, the value `1` will now be stored somewhere in memory and `int_var` is a name that points to this memory location.

We can use `int_var` just like any other value. For example, we can compute `2 + int_var` and pass the result to the `print` function. This will then print the text `3` to the `stdout` stream of our program. We can also use `int_var` in f-strings about which we learned back in Section 3.6.2. `f"int_var has value {int_var}."` will be interpolated to `"int_var has value 1."`.

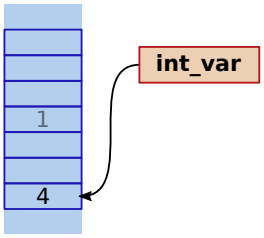
Variables are called *variables* and not *constants* because we can change their value. Hence, we can update `int_var` and give it a new value. For example, we can do `int_var = (3 * int_var )+ 1`. This will update `int_var` to now hold the result of the computation `(3 * int_var)+ 1`. In this computation, the current (old) value of `int_var` is used. It therefore corresponds to computing `(3 * 1)+ 1`, which equals `4`. This value is stored somewhere in memory and `int_var` points to it, as sketched in Figure 4.1.2. The value `1` is now no longer referenced. Eventually, the Python interpreter could free the corresponding memory to use it for something else. Doing `print(f"int_var is now {int_var}.")` will print `int_var is now 4.` to the `stdout` stream.

Of course, we can have multiple variables. The command `float_var = 3.5` creates a variable named `float_var`. It also allocates a piece of memory, writes the floating point value `3.5` into it, and lets `float_var` point to that piece of memory, as illustrated in Figure 4.1.3. We can use this variable in an f-string as well: `print(f"float_var has value {float_var}.")` is interpolated to `"float_var has value 3.5."`.

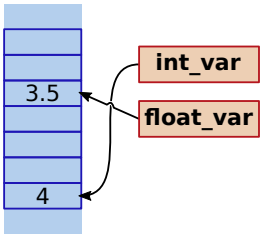
In a final step, we create a third variable with the name `new_var` by computing



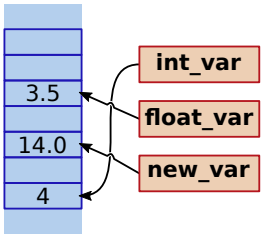
(4.1.1) `int_var = 1`



(4.1.2) `int_var = (3 * int_var)+ 1`



(4.1.3) `float_var = 3.5`



(4.1.4) `new_var = float_var * int_var`

Figure 4.1: Illustrations of the variable assignments in Listing 4.1 in the same order in which they appear in the program: Variables are basically names that point to objects which are located somewhere in memory.

`new_var = float_var * int_var`. The result is `3.5 * 4`, i.e., the `float` value `14.0`. Figure 4.1.3 illustrates this variable assignment step. Finally, `print(f"new_var = {new_var}.")` then prints `new_var = 14.0.`. (Do you remember a method, to get this output even more easily?)

This first program is stored in a file named `assignment.py`. To execute it, you have two choices: You can do this in the terminal or using PyCharm.

Under Ubuntu Linux, you open the terminal by pressing Ctrl + Alt + T, under Microsoft Windows you instead press ⊞ + R, type in `cmd`, and hit ↵. Then you enter the folder where the program `assignment.py` is stored using the command `cd`. Then you would execute the command `python3 assignment.py` to run the Python interpreter, as illustrated in Figure 4.2.4.

Alternatively, you can open the program file in PyCharm IDE, as sketched in Figure 4.2.1. You would then right-click on the file `assignment.py` in the project tree view. In the popup-menu that opens, you would left-click on Run 'assignment' as shown in Figure 4.2.2. As a shortcut, you can also simply press Ctrl + ⇧ + F10. Either way, PyCharm will run the program and the output appears in Figure 4.2.3. As you see, the full standard output stream (`stdout`) of the complete program given in Listing 4.2 is identical to what we get from the manual execution in either the terminal or PyCharm.

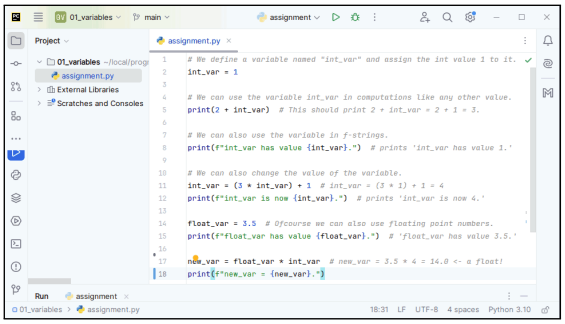
We are not completely done yet, though. Notice that we wrote the names of variables in a certain style, in lowercase letters. This is the *de facto* standard way to name variables and functions in Python programming.

Best Practice 11

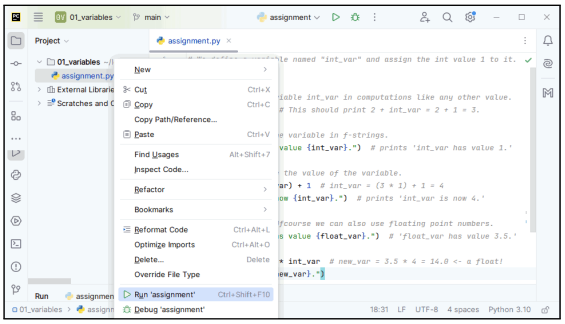
Variable names should be lowercase, with words separated by underscores [594]. This style is called *snake case* or *snake_case* [656].

This seems like a strange thing to introduce right at the beginning when learning programming. Matter of fact, we now have seen some best practices on styling our code, e.g., Best Practice 8, 10 and 11, and many more such best practices will follow. Before continuing further, let us therefore revisit the deeper meaning behind them. Why is it important to style our code in a consistent way? Why can't we just write things down in any way that pleases us?

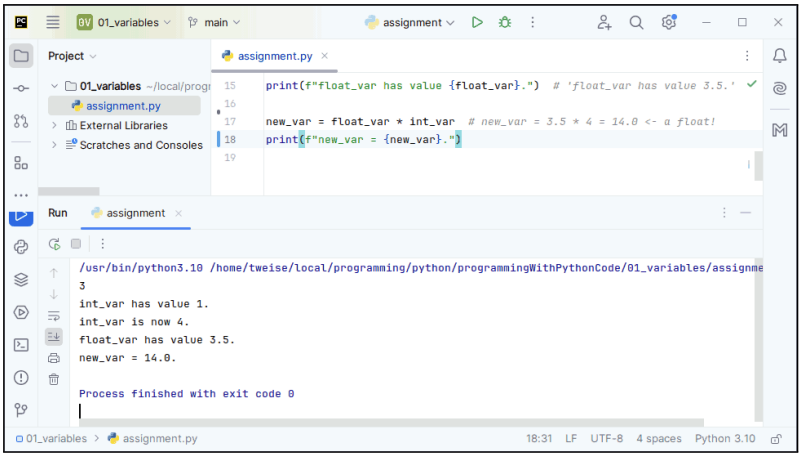
Well, because following best practices is nothing that can be done "later." You will never have the time to revisit and improve the style of your old code. It is also nothing that you can just switch over to. If you have learned doing a certain thing in a certain way, it will always be hard to switch over to a different way. If an apprentice in a kitchen is not taught to wash their hands before beginning to



(4.2.1) The file `assignment.py` opened in PyCharm.



(4.2.2) Left-clicking on Run 'assignment' in the pop-up menu after right-clicking on `assignment.py`, or directly pressing Ctrl + ⇧ + F10, to run the program.



(4.2.3) The output of the program `assignment.py` in PyCharm.



(4.2.4) The output of the program `assignment.py` in the Ubuntu terminal (which you can open via Ctrl + Alt + T).

Figure 4.2: Running the program `assignment.py` from Listing 4.1 in PyCharm (Figures 4.2.1 to 4.2.3) or the Ubuntu terminal (Figure 4.2.4).

prepare meals, then they will not simply begin doing that consistently after being told to do it once after they join a new restaurant. Following style guides and best practices is a habit. And we need to nurture this habit right from the start.

Best Practice 12

Regardless which programming language you are using, it is important to write code and scripts in a consistent style, to use a consistent naming scheme for all things that can be named, and to follow the generally established best practices and norms for that language.

For many programming languages, there exist comprehensive and clear style guides. Since we usually work collaboratively on larger projects, writing code in a consistent style is very important. Ideally, all collaborators can open a source code file and easily read and understand our code. If everybody writes code in different styles, maybe using different indentations or different naming conventions, reading code can become harder and even confusing. Therefore, style guides often tell us how to name things and how to structure code consistently.

Best Practice 13

The most important style guide for the Python programming language is PEP8: *Style Guide for Python Code* [594] at https://peps.python.org/pep-0008. Python code violating PEP8 is invalid Pythoncode.

4.1.2 LIU Hui's Method for the Approximation of π

Let us now come to a more serious example. I am not good at mathematics, but I still really like mathematics anyway, so we will go with a mathematics example: approximating π. The number π is the ratio of the circumference of a circle and its diameter. A we already mentioned before in Section 3.3, it is irrational, a never-ending and never-repeating sequence of digits. It is also transcendental, i.e., not the root of any polynomial with rational coefficients. We can compute it to a certain precision, e.g., as the `float` constant `pi` with value `3.141592653589793`. But we can never really write it down in its entirety.

Well, when I say "we can compute it", then the question "How?" immediately arises. One particularly ingenious answer was given by the Chinese mathematician LIU Hui (刘徽, who may have looked as sketched Figure 4.3) somewhere in the third century Common Era (CE) [420, 649] in his commentary to the famous Chinese mathematics book *Jiu Zhang Suanshu* (九章算术) [140, 144, 311, 420, 540]. In Figure 4.4, we show how π can be approximated based on the idea of LIU Hui (刘徽): By inscribing regular e-gons with an increasing number e of edges into a circle such that the corners of the e-gons lie on the circle.

We start with a hexagon ($e = 6$) where the radius r is equal to the radius of the circle. Since it is a regular hexagon, it can be divided into six triangles. These are isosceles triangle, since two sides of each triangle have length r, since two of its edges are on the cirlce and the third edge is the circle center. The apex angle between these two eges is 60°, as $360°/6 = 60°$. Therefore, the triangles are



Figure 4.3: Modern rendition of LIU Hui (刘徽) from [649]. Not under the Creative Commons license, copyright © is with CAST / Official WeChat account of VOC.

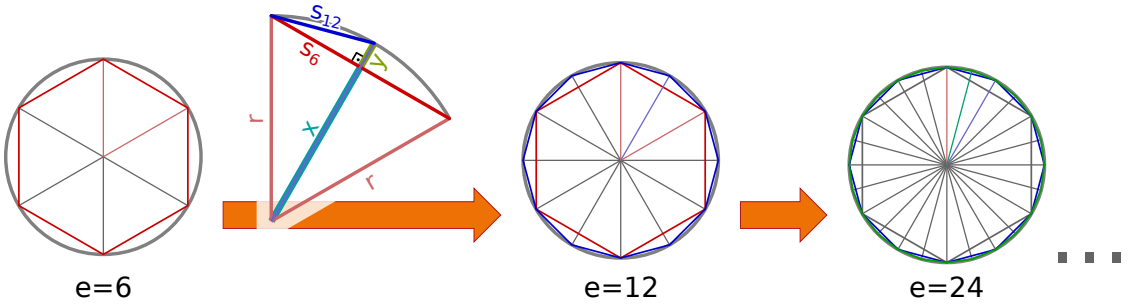


Figure 4.4: Approximating the ratio of the circumference and the diameter of a circle, i.e., π, by inscribing regular $3 * 2^n$-gons.

equilateral, meaning that the third side also has length r.

This means that all the e edges s_6 of this hexagon then have length r as well. It is easy to see that the circumference of the hexagon is $U = e * s_6 = 6 * r$. The diameter of the circle is $D = 2r$. Assuming that the circumference of the hexagon is an approximation of the circumference of the circle, we could approximate π as $\pi \approx \frac{U}{D}$. For $e = 6$ edges, this gives us $\pi_6 = \frac{6r}{2r} = 3$.

Now this is a very coarse approximation of π. We can get closer to the actual ratio if we would use more edges, i.e., higher values of e. The ingenious idea of LIU Hui (刘徽) is to use e-gons with $e = 3 * 2^n$. For $n = 1$, we get the hexagon with $e = 6$. For $n = 2$, we double the edges and have a dodecagon with $e = 12$ edges. But how do we get the edge length s_{12} of this dodecagon?

We can get it from the edge length s_6 and radius r of the hexagon. If we use the same six corners for the hexagon and dodecagon and connect the newly added six corners with the center of the circle, then these connections will separate each edge of the hexagon exactly in half and do so at a 90° angle, as shown again in Figure 4.4. Here, the new side length s_{12} is the hypotenuse of a right-angled triangle with base $\frac{s_6}{2}$ and height y. To get the height y, we can use that $r = x + y$ and the fact that there is a second right-angled triangle here, namely the one with base x, height $\frac{s_6}{2}$, and hypotenuse r. This gives us $x^2 + \left(\frac{s_6}{2}\right)^2 = r^2$. Let's make things easier by choosing $r = 1$.

We get $x^2 = 1 - \left(\frac{s_6}{2}\right)^2 = 1 - \frac{{s_6}^2}{4}$ and, hence, $y = 1 - \sqrt{1 - \frac{{s_6}^2}{4}}$. With this we can move on to ${s_{12}}^2 = y^2 + \left(\frac{s_6}{2}\right)^2$, which we can resolve to ${s_{12}}^2 = \left(1 - \sqrt{1 - \frac{{s_6}^2}{4}}\right)^2 + \frac{{s_6}^2}{4}$. Using $(a-b)^2 = a^2 - 2ab + b^2$ and applying it to the first term, we get ${s_{12}}^2 = 1 - 2\sqrt{1 - \frac{{s_6}^2}{4}} + \left(1 - \frac{{s_6}^2}{4}\right) + \frac{{s_6}^2}{4}$. This then gives us ${s_{12}}^2 = 2 - 2\sqrt{1 - \frac{{s_6}^2}{4}} - \frac{{s_6}^2}{4} + \frac{{s_6}^2}{4}$, which we can further refine to ${s_{12}}^2 = 2 - 2\sqrt{1 - \frac{{s_6}^2}{4}}$. We can pull the 2 from outside the root into the root by multiplying everything inside by $2^2 = 4$ and get ${s_{12}}^2 = 2 - \sqrt{4 - {s_6}^2}$. Thus, we have the really elegant $s_{12} = \sqrt{2 - \sqrt{4 - {s_6}^2}}$.

As new approximation of π_{12}, we now have $\frac{12 * s_{12}}{2r} = 6 * s_{12} = 6\sqrt{2 - \sqrt{4 - {s_6}^2}} = 6\sqrt{2 - \sqrt{4 - 1}} = 6\sqrt{2 - \sqrt{3}} \approx 3.105828539$. This is already quite nice. We can actually repeat this step to get to s_{24}. And we could continue this process by again doubling the number the edges. Repeating the above calculations and observing Figure 4.4, we get the equation:

$$s_{2e} = \sqrt{2 - \sqrt{4 - s_e^2}} \tag{4.1}$$

$$\pi_{2e} = \frac{e}{2} s_{2e} \tag{4.2}$$

Now that we have learned some programming, we do no longer need to type the numbers and computation steps into a calculator. We also do not need to use `Python` as calculator. Instead, we can enter all the commands needed for the computation into a program file. We will call it `pi_liu_hui.py`, as illustrated in Listing 4.3.

We begin by setting the initial number of edges `e = 6` and the initial side length of the e-gon to `s = 1`, because we still keep with the choice of $r = 1$. In each iteration of the approximation, we simply set `e *= 2`, which is equivalent to `e = e * 2`, to double the number of edges. We compute `s = sqrt(2 - sqrt(4 - (s ** 2)))` having imported the `sqrt` function from the `math` module. We print the approximated value of π as `e * s / 2`. Notice how elegantly we use the Unicode characters π and $\approx$ via the escape sequences `\u03c0` and `\u2248`, respectively, from back in Section 3.6.6 (and how nicely it indeed prints the greek character π in the `stdout` in Listing 4.4). Either way, since Equations 4.1 and 4.2 are always the same, we can simply copy-paste the lines of code for updating `s`, `e`, and printing the approximated value of π several times.

Listing 4.4 shows the standard output stream (`stdout`) produced by this program. Indeed, each new approximation comes closer to π. For 192 edges, we get the approximation `3.1414524722853443`. Given that the constant `pi` from the `math` module is `3.141592653589793`, we find that the first four digits are correct and that the number is only off by only 0.0045%!

For your convenience, we also showed the results when executing the program in `PyCharm` or the `Ubuntu` terminal in Figure 4.5. To open a terminal under `Ubuntu` `Linux`, you would press Ctrl + Alt + T, whereas under Microsoft `Windows`, you press ⊞ + R, type in `cmd`, and hit ↵. With the command `cd`, you would enter the directory where our program `pi_liu_hui.py` is located. You would

Listing 4.3: A Python program showing several steps of the approximation of π using the method of LIU Hui (刘徽). (stored in file pi_liu_hui.py; output in Listing 4.4)

```
from math import pi, sqrt   # We need sqrt. pi is for comparison.

# We use f-strings with Unicode escapes to print the current result.
# "\u03c0" is the Unicode escape for the Greek letter pi.
# "\u2248" is the Unicode escape for the "approximately equal" sign.
print(f"We use Liu Hui's Method to Approximate \u03c0\u2248{pi}.")
e = 6    # the number of edges: We start with a hexagon, i.e., e=6.
s = 1.0  # the side length: Initially 1, meaning the radius is also 1.
print(f"{e} edges, side length={s} give \u03c0\u2248{e * s / 2}.")

e *= 2   # We double the number of edges... ...now there are 12.
s = sqrt(2 - sqrt(4 - (s ** 2)))  # ...and recompute the side length.
print(f"{e} edges, side length={s} give \u03c0\u2248{e * s / 2}.")

e *= 2  # We double the number of edges... ...now there are 24.
s = sqrt(2 - sqrt(4 - (s ** 2)))  # ...and recompute the side length.
print(f"{e} edges, side length={s} give \u03c0\u2248{e * s / 2}.")

e *= 2  # We double the number of edges... ...now there are 48.
s = sqrt(2 - sqrt(4 - (s ** 2)))  # ...and recompute the side length.
print(f"{e} edges, side length={s} give \u03c0\u2248{e * s / 2}.")

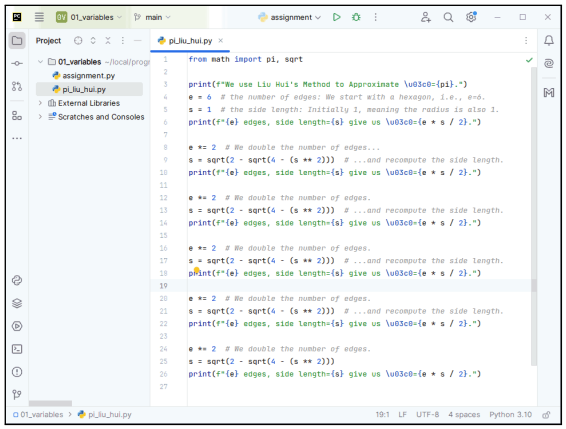
e *= 2  # We double the number of edges... ...now there are 96.
s = sqrt(2 - sqrt(4 - (s ** 2)))  # ...and recompute the side length.
print(f"{e} edges, side length={s} give \u03c0\u2248{e * s / 2}.")

e *= 2  # We double the number of edges... ...now there are 192.
s = sqrt(2 - sqrt(4 - (s ** 2)))
print(f"{e} edges, side length={s} give \u03c0\u2248{e * s / 2}.")
```

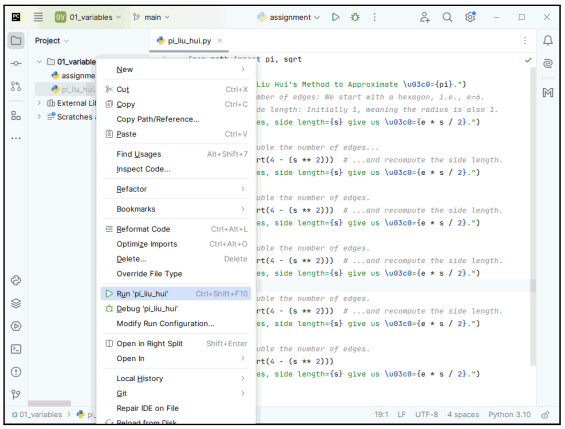
↓ `python3 pi_liu_hui.py` ↓

Listing 4.4: The stdout of the program pi_liu_hui.py given in Listing 4.3.

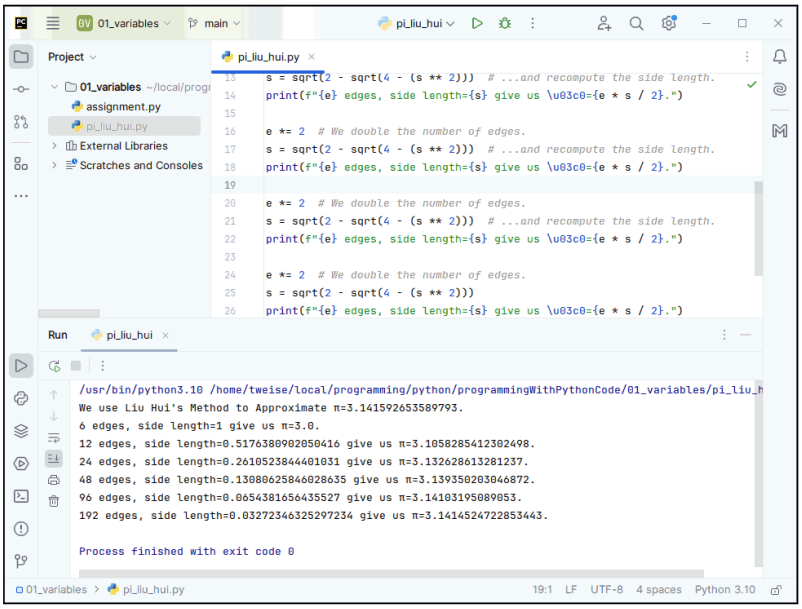
```
We use Liu Hui's Method to Approximate π≈3.141592653589793.
6 edges, side length=1.0 give π≈3.0.
12 edges, side length=0.5176380902050416 give π≈3.1058285412302498.
24 edges, side length=0.2610523844401031 give π≈3.132628613281237.
48 edges, side length=0.13080625846028635 give π≈3.139350203046872.
96 edges, side length=0.0654381656435527 give π≈3.14103195089053.
192 edges, side length=0.03272346325297234 give π≈3.1414524722853443.
```



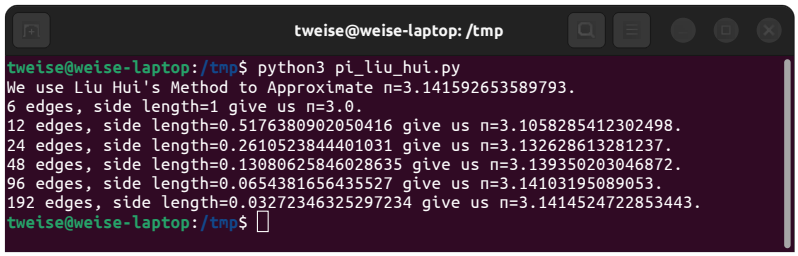
(4.5.1) The file `pi_liu_hui.py` opened in PyCharm.



(4.5.2) Left-clicking on Run 'pi_liu_hui' in the pop-up menu after right-clicking on `pi_liu_hui.py`, or directly pressing Ctrl+⇧+F10, to run the program.



(4.5.3) The output of the program `pi_liu_hui.py` in PyCharm.



(4.5.4) The output of the program `pi_liu_hui.py` in the Ubuntu terminal (which you can open via Ctrl+Alt+T).

Figure 4.5: Running the program `pi_liu_hui.py` from Listing 4.1 in PyCharm (Figures 4.5.1 to 4.5.3) or the Ubuntu terminal (Figure 4.5.4).

then type in `python3 pi_liu_hui.py` and hit ↵. As you can see in Figure 4.5.4, you will get the same output as given in Listing 4.4.

Alternatively, if you are using PyCharm, you can open the program file `pi_liu_hui.py` as shown in Figure 4.5.1. You can right-click on this program in the project tree view and, in the pop-up menu that appears, left-click on Run 'pi_liu_hui', as sketched in Figure 4.5.2. You can instead also press Ctrl + ⇧ + F10 in the editor window. Either way, the program will be executed and its output appears (see Figure 4.5.3). And again it is identical to what we have shown in Listing 4.4. Therefore, in the future, we will only very sporadically add such screenshots. Instead, we will usually only print code and output pairs like Listings 4.3 and 4.4.

4.1.3 Variables can be modified

First Time Readers and Novices: This section tells you about some weird stuff possible in Python but not in many other programming languages. It may be confusing, so you can just as well skip it (unless you are curious...).

At the beginning of this section, I wrote *"...ou have seen this before, when we used the variables `pi`, `e`, `inf`, and `nan` that we imported from the module `math`."* Now, this may sound kind of weird, because we would normally consider things like π or e as *constants* and not as variables. Calling them *variables* sort of implies that we can change them, which would obviously not make any sense.

While you cannot change the mathematical constant π, the name `pi` in module `math` itself just identifies a variable storing this constant. As the output Listing 4.6 of the program given in Listing 4.5 shows, we can actually change the values of these variables. As this could wreak havoc in any other code or library that also accesses and uses these constants, there seem to be few use cases for this skill outside of nefarious purposes. Still, I think knowing this counter-intuitive information may be an interesting complement to our basic programming knowledge.

Best Practice 14

Do never change the values of variables that are not supposed to be changed.

Listing 4.5: `inf`, `nan`, `pi`, and `e` are actually *variables* in the module `math`, so technically, we can change them. This has no impact on how the other mathematical functions work. But it may create all sorts of unpredictable effects in other code or libraries using these variables... (stored in file variables_of_math.py; output in Listing 4.6)

```
# NEVER NEVER NEVER EVER DO THIS.
# Python does not know constants. Hence, inf, nan, pi, e are variables.

import math                     # Let's grab the math module.
from math import sin            # Import the sin function.

print(math.pi)                  # This prints 3.141592653589793
print(sin(math.pi))             # 1.2246467991473532e-16, so basically 0.

math.pi = 3                     # We change the value of the variable pi.
print(math.pi)                  # This now prints 3.

from math import pi             # And import the variable into our scope.
print(math.sin(pi))             # We changed the variable pi, but the
                                # mathematical constant is still the same.
                                # So here we get 0.1411200080598672.

math.nan = 5                    # Now the variable "nan" becomes 5, but 5 is
                                # still a number, i.e., not a (not a number)!
print(math.isnan(math.nan))     # Therefore, this prints False.

math.inf = 10000                # 10000 is fairly large, right?
from math import inf            # Let's get the value `inf` into our scope.
print(50000 > inf)              # This now prints True.
```

↓ `python3 variables_of_math.py` ↓

Listing 4.6: The stdout of the program variables_of_math.py given in Listing 4.5.

```
3.141592653589793
1.2246467991473532e-16
3
0.1411200080598672
False
True
```

4.1.4 Summary

We now have learned how we can store values in variables. We also have learned that we can then use the variables instead of the values. We further have learned that we can assign new values to variables and do so multiple times.

This means that we can now, for the first time, create meaningful programs consisting of multiple lines of code. Programs, in which the intermediate results computed by one line of code are transmitted and used in a later line of code. This is a huge jump.

Now our code can become much more complex. This means that we must now begin to consider issues such as code quality. We need to follow style guides to make the code readable. We need to write comments into our code, so that we (and others) can later understand what we were thinking when writing the programs.

But most importantly: We can already do some cool stuff!

4.2 Interlude: Finding Errors in your Code with the IDE and Exceptions

Before we depart from PyCharm screenshots, however, we will visit one absolutely crucial functionality that modern IDEs provide: They help us to find errors in the code. Errors are common. They happen all the time. All programmers make mistakes. Sometimes it's a typo. Sometimes we accidentally switch

Listing 4.7: A variant of program assignment.py from Listing 4.1 with an error: int_var is accidentally spelled as intvar in one location. (stored in file assignment_wrong.py; output in Listing 4.8)

```
# We define a variable named "int_var" and assign the int value 1 to it.
int_var = 1

# We can use the variable int_var in computations like any other value.
print(2 + int_var)  # This should print 2 + int_var = 2 + 1 = 3.

# We can also use the variable in f-strings.
print(f"int_var has value {int_var}.")  # prints 'int_var has value 1.'

# We can also change the value of the variable.
int_var = (3 * int_var) + 1  # int_var = (3 * 1) + 1 = 4
print(f"int_var is now {intvar}.")  # prints 'int_var is now 4.'

float_var = 3.5  # Ofcourse we can also use floating point numbers.
print(f"float_var has value {float_var}.")  # 'float_var has value 3.5.'

new_var = float_var * int_var  # new_var = 3.5 * 4 = 14.0 <- a float!
print(f"{new_var = }.")
```

↓ python3 assignment_wrong.py ↓

Listing 4.8: The stdout, standard error stream (stderr), and exit code of the program assignment_wrong.py given in Listing 4.7.

```
3
int_var has value 1.
Traceback (most recent call last):
  File "{...}/variables/assignment_wrong.py", line 12, in <module>
    print(f"int_var is now {intvar}.")  # prints 'int_var is now 4.'
                            ^^^^^^
NameError: name 'intvar' is not defined. Did you mean: 'int_var'?
# 'python3 assignment_wrong.py' failed with exit code 1.
```

the order of parameters of a function. Sometimes we mix up floats and ints. Sometimes we make logical errors. Sometimes we misunderstood what the function that we are calling does. Sometimes we misunderstood the algorithm that we try to implement. Some errors are obvious and easy to fix. Some errors can be discovered based on the error output of the program. Some errors require more serious debugging to even spot them (see Section 13.4). In some simple cases like typos, however, our IDE can already point us to the mistake.

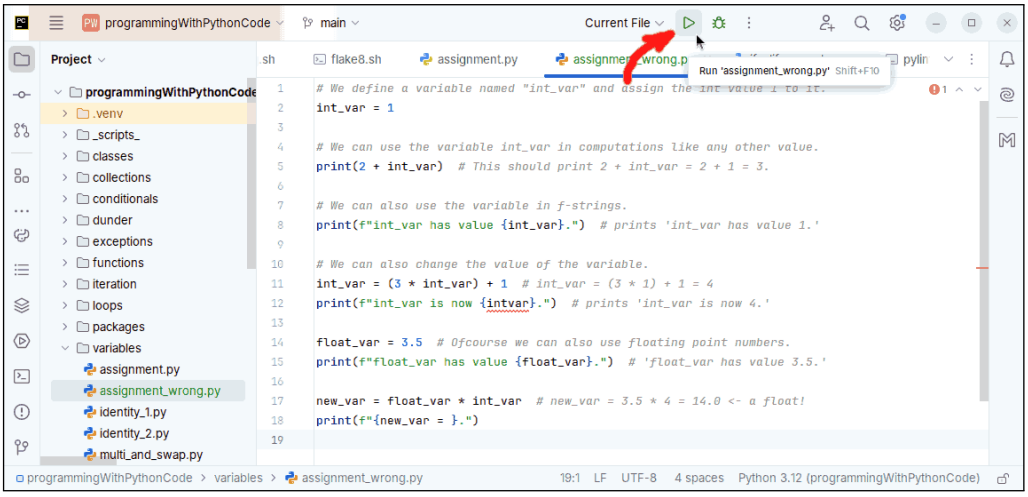
4.2.1 Finding the Error by using the Program Output and Exceptions

In Listing 4.7, we prepared program assignment_wrong.py, a variant of assignment.py (Listing 4.1) with an error. For the sake of the example, let us assume that the programmer made a typo in line 12 of the program: They misspelled int_var as intvar. Executing the program with the error leads to the output given in Listing 4.8.

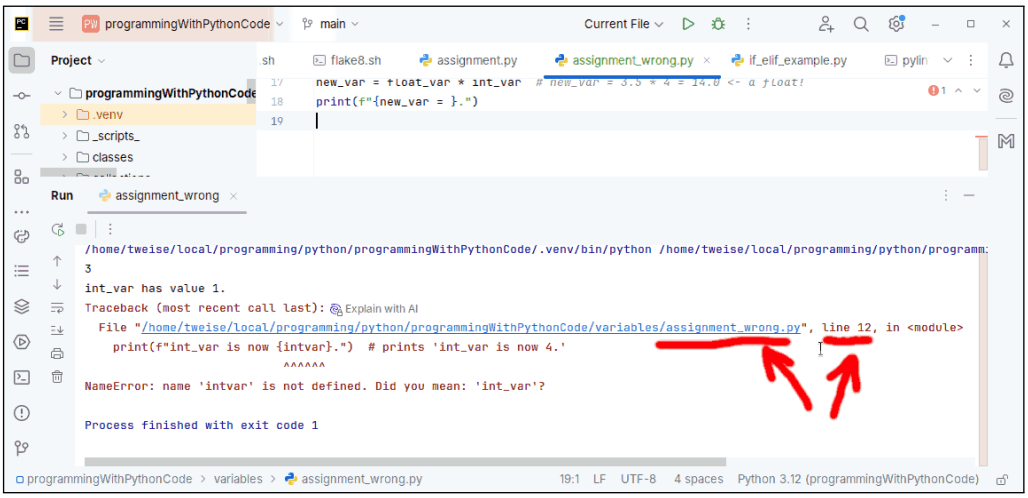
We can also open the program assignment_wrong.py given in the PyCharm IDE. We execute this program manually by clicking on the ▷ button or by pressing ⇧+F10 in Figure 4.6.1. As you can see, the output in the run window is the same as given in Listing 4.8 (Figure 4.6.2).

If a program crashes, then *carefully* reading its output is always the *first* way to find out what went wrong. Indeed, in our example, the text that appeared tells us what went wrong and even suggests how to fix it. It says: *"NameError: name 'intvar' is not defined. Did you mean: 'int_var'?"* This is already pretty clear. We accessed some variable (a name), intvar, which has not been defined or assigned. It simply does not exist.

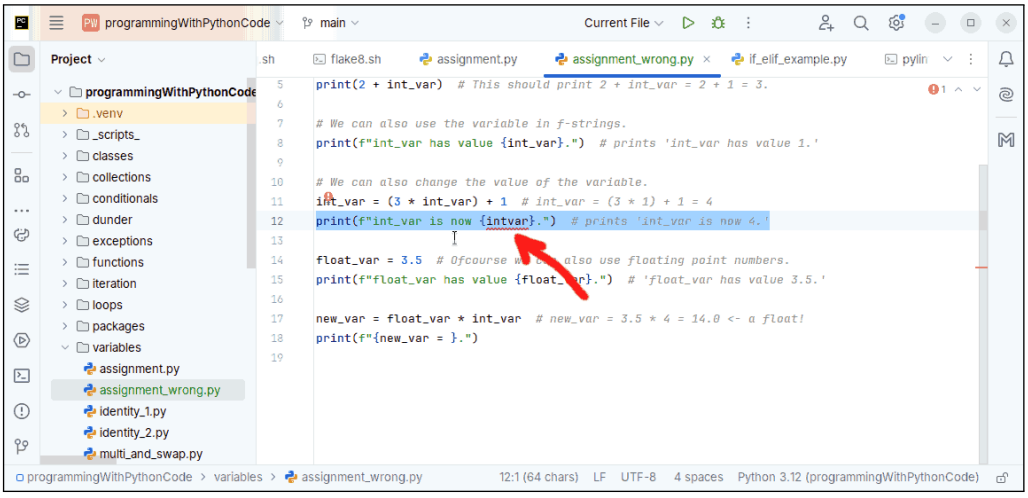
The Python interpreter then checks whether some similar name exists. It found that there is a variable named int_var. Even more, it also tells us the exact file and line where the error occurred, namely in line 12 of file assignment_wrong.py! With this information, we have a good chance of



(4.6.1) We run the program `assignment_wrong.py` given in Listing 4.7 in the `PyCharm` IDE by clicking on the ▷ button or by pressing ⇧+F10.

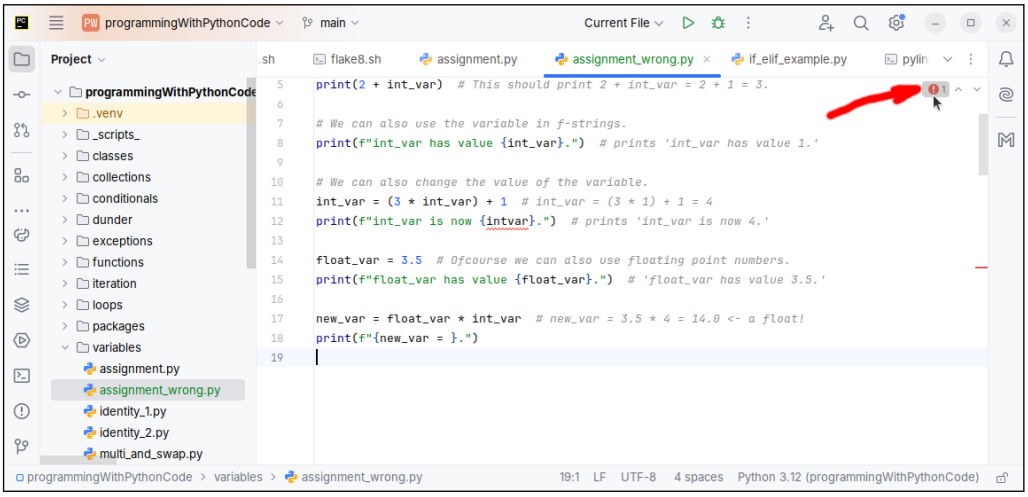


(4.6.2) The output in the run window is the same as given in Listing 4.8. It is the *first* way to find out what went wrong. It tells us what went wrong and even suggests how to fix it: *`NameError`: name '`intvar`' is not defined. Did you mean: '`int_var`'?* It also tells us the exact file and line where the error occurred, namely in line 12 of file `assignment_wrong.py`.

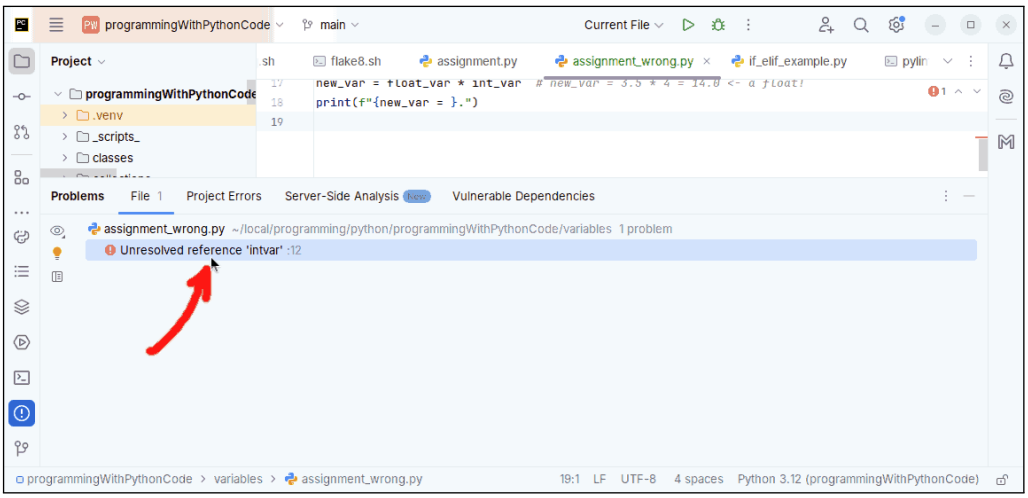


(4.6.3) If we click on the linked file location, it takes us to where the error occurred. The incorrectly typed word is (and always was) underlined with red color. Looking for underlined words is the *second* method to find errors in code!This should have told us already that something is fishy without the need to even run the program in the first place.

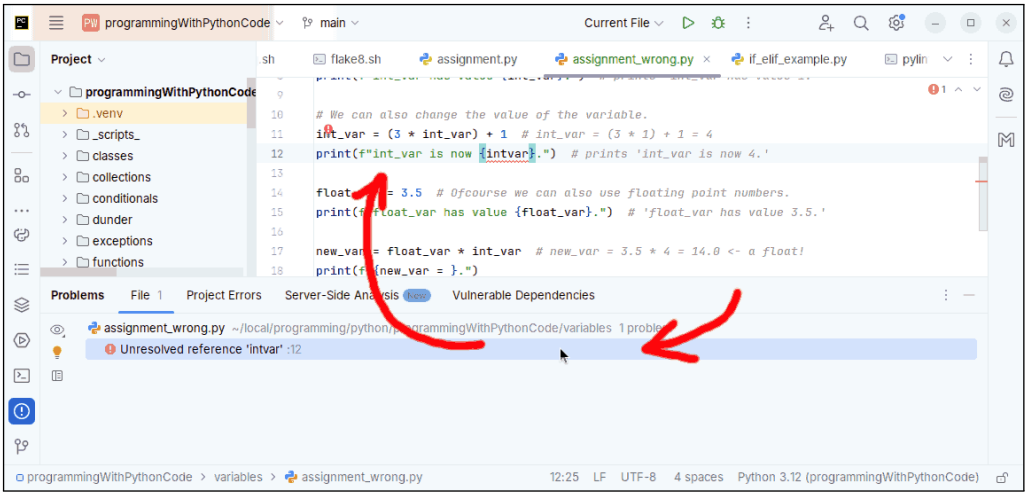
Figure 4.6: How the IDE can help us finding errors.



(4.6.4) The IDE also informs us that something is wrong by displaying the small red ❗ icon in the top-right corner. This is the *third* way to find errors. We click on it...

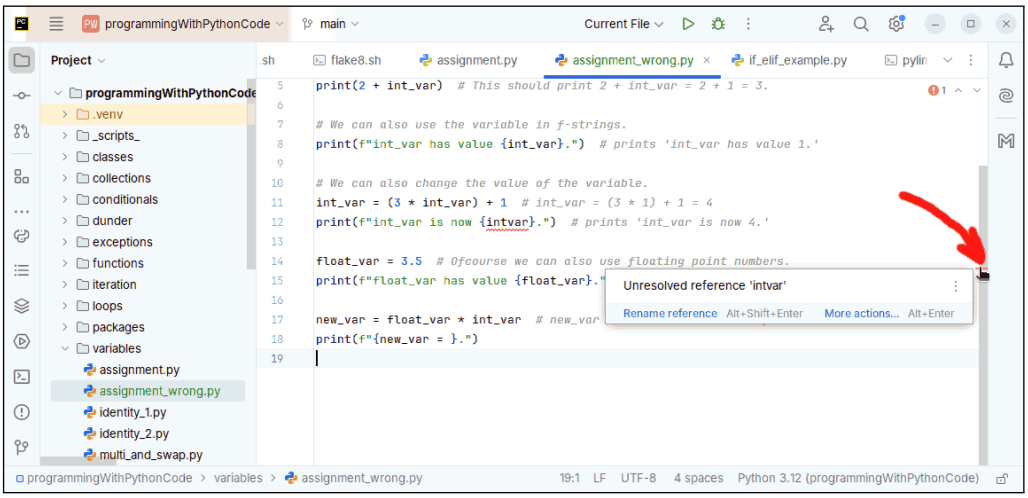


(4.6.5) ...and it takes us to the list of potential errors that it has detected. Here, it tells us that there is an *Unresolved reference 'intvar'* at line 12 of the file. We click on this note...

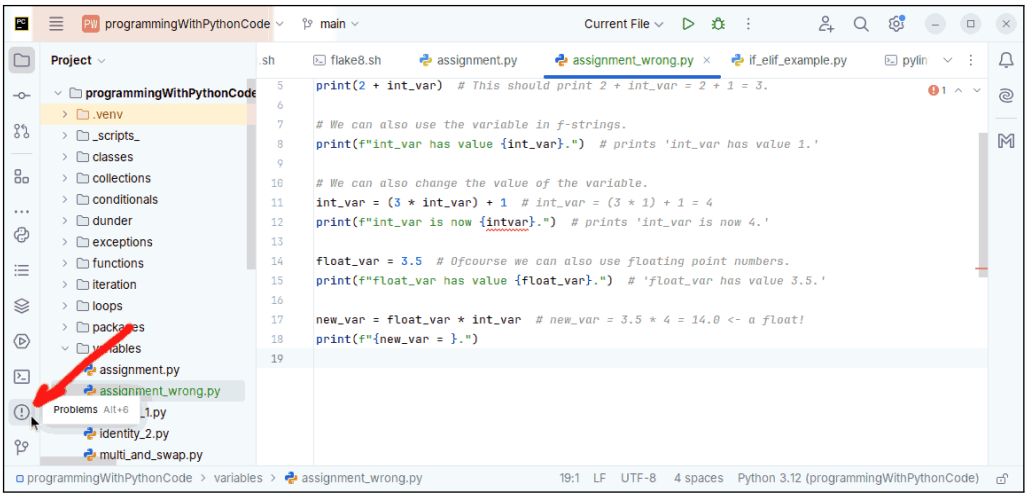


(4.6.6) ...and it takes us again to the dodgy line.

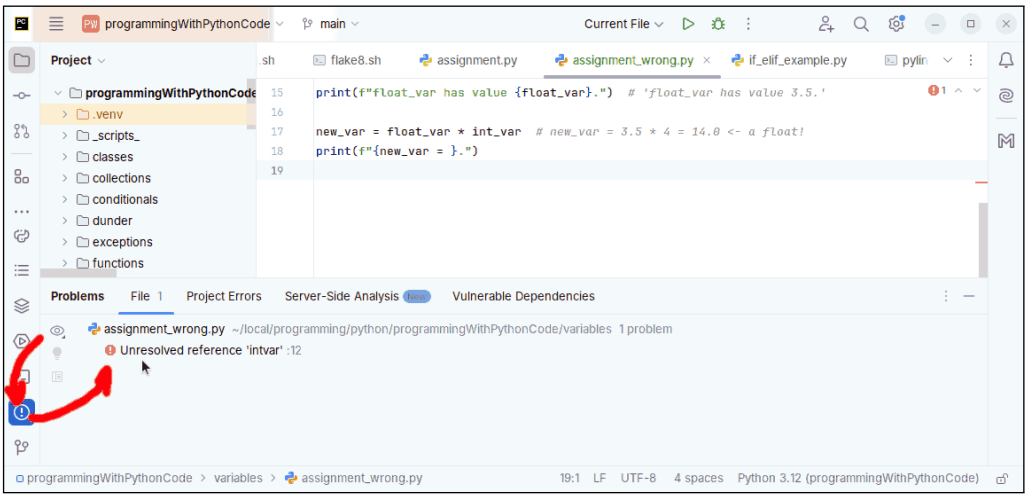
Figure 4.6: How the IDE can help us finding errors.



(4.6.7) The *fourth* way in which the `PyCharm` IDE can help us to discover errors are small red marks at the right-hand side. Holding the mouse cursor over these lines will open a small view with the suggested error message.



(4.6.8) The *fifth* way to get a list of potential errors in `PyCharm` is to click on the ⓘ button in the side menu on the left-hand side or to press Alt + 6 .



(4.6.9) This again takes us to the list of potential errors.

Figure 4.6: How the IDE can help us finding errors.

finding the mistake.

The so-called `Exception` stack trace that it prints to `stderr` thus not just tells us that there was an error. It even tells us a probable cause and the most-likely line of code that caused the error. We will discuss this topic in-depth later in Chapter 9, but even at this stage, the message here is pretty clear.

Best Practice 15

Always carefully *read* error messages. They often provide you very crucial information where to look for the mistake. Not reading error messages is wrong.

4.2.2 Finding the Error by using the IDE

We found the error by executing the program, seeing that it crashed, and then reading the output. In the run console of `PyCharm` that presents the program output, we can even click on the linked file location. This takes us to where the error occurred in Figure 4.6.3.

The question is: Could we have found this error even without executing the program? Actually, when looking at the line to which the click took us, we notice that the incorrectly typed word is (and always was) underline with red color. This should have told us already that something is fishy without the need to even run the program in the first place.

Best Practice 16

When writing code, we should always check whether the IDE notifies us about potential errors. In the case of `PyCharm`, these are often underlined in red or yellow color. We should always check all such marks!

So we already know two ways in which we can find errors in our code with the help of our IDE. But there are even more ways.

The IDE also informs us that something is wrong by displaying the small red icon in the top-right corner, as shown in Figure 4.6.4. Clicking on this symbol is the third way to find errors. This will take us to the list of potential errors that it has detected in Figure 4.6.5. Here, `PyCharm` tells us that there is an *"Unresolved reference 'intvar'"* at line 12 of the file. We can also click on that note, and it takes us again to the dodgy line in Figure 4.6.6

The fourth method in which the `PyCharm` IDE can help us to discover errors are small red marks at the right-hand side of our editor window, shown in Figure 4.6.7. Holding the mouse cursor over these marks will open a small view with the suggested error message.

The fifth way to get a list of potential errors in `PyCharm` is to click on the button in the side menu on the left-hand side or to press Alt + 6, as illustrated in Figure 4.6.8. This again takes us to the list of potential errors in Figure 4.6.9.

Useful Tool 3

The IDE and the error messages (`Exception` stack traces) are your most important tools to find errors. Read error messages. If your IDE – regardless whether it is `PyCharm` or something else – annotates your code with some marks, then you should check every single one of them.

4.2.3 Summary

Programmers make errors. We make errors. You are going to make lots of errors. Later and now. Even now, in very simple programs, we are going to make errors. This cannot be prevented. The question can only be how we deal with that. How can we minimize the number of errors that we are going to make? How can we maximize that chance to discover and fix errors?

The answer is simple: *By using all the tools at our disposal.* We can use the output of programs to find errors after running the programs. And we can use our IDE to find errors based on the hints it provides to us even before executing the programs. Both tools make it much much easier to find errors.

You can guess the importance of error-finding features also by the fact that PyCharm implements the error hints in several different ways, in the hope to get you to click and investigate its list of proposed errors and warnings. As mentioned in Best Practice 15 and 16, using the IDE features for error discovery and detection is incredibly important.

Even if your program executes as expected, there still might be hidden errors in the code. Sometimes, you can easily tell whether the output of a program is correct. Sometimes, you cannot. In some cases, an output that looks OK might still be wrong.

Sometimes, there might be some incorrect instructions in your program that just weren't used in your last execution. So even correct program output does not guarantee that the program itself is correct. Therefore, always checking each and every piece of code that your IDE marks as dodgy is very important. Make sure that you fully understand all error and warning messages.

Warnings can be important, too. They can indicate possible errors, potential problems with variable types, or missing required packages (see, e.g., Section 15.1.1). Always fix errors and warnings wherever possible (obviously). Even where you deem the warnings as false-positives, try to fix them anyway. They could result from non-standard code formatting that still "works", but may be confusing for readers of your code. You should always try to produce warning-free code. For every warning or error that you deem as not a problem, remember: On the one hand, *you* might be wrong. On the other hand, having fewer warnings and false-positive suspected errors makes it easier to find the actual problem if an actual problem happens.

4.3 Multiple Assignments and Value Swapping

Let us return to the topic of variables and assignments. In Python, we can assign values to multiple variables at once (of course, always exactly one value to one variable). In this case, we separate both the variable names and the values with commas. The first line `a, b = 5, 10` program multi_and_swap.py given as Listing 4.9 is equivalent to the two lines `a = 5` and `b = 10`. It assigns the value `5` to the variable `a` and the value `10` to the variable `b`. After this assignment step, `a == 5` and `b == 10` holds. `print(f"{a = }, {b = }")` therefore prints $a = 5$, $b = 10$.

This method can also be used to *swap* the values of two variables. Writing `a, b = b, a` looks a bit strange but it basically means "the *new* value of `a` will be the *present* value of `b` and the *new* value of `b` will be the *present* value of `a`." The line is therefore basically equivalent to `t = a`, `a = b`, and `b = t`. Without the double-assignment, we would first have to store the value of `a` in some temporary variable `t`, then overwrite `a` with `b`, and finally we would copy the value of `t` to `b`. But with the double-assignment, we can accomplish this in a single line of code instead of three. And we do not

Listing 4.9: A Python program assigning multiple values to multiple variables and using the same method to swap variable values. (stored in file multi_and_swap.py; output in Listing 4.10)

```
a, b = 5, 10
print(f"{a = }, {b = }")

a, b = b, a
print(f"{a = }, {b = }")

z, y, x = 1, 2, 3
print(f"{x = }, {y = }, {z = }")

x, y, z = z, y, x
print(f"{x = }, {y = }, {z = }")
```

↓ `python3 multi_and_swap.py` ↓

Listing 4.10: The stdout of the program multi_and_swap.py given in Listing 4.9.

```
a = 5, b = 10
a = 10, b = 5
x = 3, y = 2, z = 1
x = 1, y = 2, z = 3
```

need a temporary variable either. `print(f"{a = }, {b = }")` thus now prints *a=10, b=5*.

Best Practice 17

Swapping of variable values can best be done with a multi-assignment statement, e.g., `a, b = b, a`.

The same concept of multiple assignments works for arbitrarily many variables. `z, y, x = 1, 2, 3` assigns, respectively, `1` to `z`, `2` to `y`, and `3` to `x`. `print(f"{x = }, {y = }, {z = }")` thus yields `x = 3, y = 2, z = 1`.

We can also swap multiple values. `x, y, z = z, y, x` assigns the present value of `z` to become the new value of `x`, the present value of `y` to also be the new value of `y`, and the present value of `x` to become the new value of `z`. `print(f"{x = }, {y = }, {z = }")` now gives us `x = 1, y = 2, z = 3`.

4.4 Variable Types and Type Hints

4.4.1 Variable Types

A variable is basically a name pointing to an object. Each object has a type and we already learned about several of these datatypes (`int`, `bool`, `str`, ...) in Chapter 3. We can obtain the type of an object stored in variable `var` by invoking `type(var)`. Listing 4.11 shows a program `variable_types.py` that does just that and the output of that program is given in Listing 4.12. It is obvious that the type of a variable that holds an integer value is `int`, the type of a variable that holds a floating point number is `float`, and so on. There really is not much to say about that.

4.4.2 Types and Confusion

Listing 4.11: An example of the types of variables. (stored in file `variable_types.py`; output in Listing 4.12)

```
1  int_var = 1 + 7  # Create an integer variable holding an integer number.
2  print(type(int_var))  # This prints "<class 'int'>".
3
4  float_var = 3.0  # 3.0 is a float and it is stored in float_var.
5  print(type(float_var))  # This prints "<class 'float'>".
6
7  str_var = f"f{float_var = }"  # Render an f-string into str_var.
8  print(type(str_var))  # This prints "<class 'str'>".
9
10 bool_var = (1 == 0)  # 1 == 0 is False, so a bool is stored in bool_var.
11 print(type(bool_var))  # This prints "<class 'bool'>".
12
13 none_var = None  # We create none_var which, well, holds None.
14 print(type(none_var))  # This prints "<class 'NoneType'>".
```

↓ `python3 variable_types.py` ↓

Listing 4.12: The stdout of the program `variable_types.py` given in Listing 4.11.

```
1 <class 'int'>
2 <class 'float'>
3 <class 'str'>
4 <class 'bool'>
5 <class 'NoneType'>
```

Well, actually, there is. You see, when you declare a variable in a language like C, you have to specify its *type*. You are then only permitted to assign values that have exactly this type to the variable. In Python, you do not need to specify a type for variables. You can assign whatever value you want to a variable. You can even first store a text string in a variable and later store a Boolean value in it, if you want. Clearly, this means that different programming languages have different approaches to handle datatypes. Let us look at the basic type-related properties of programming languages [64] to better understand this situation.

Definition 4.1: Strongly-Typed Language

In a strongly-typed programming language, every *value* has a *type* that cannot change.

Such a language does not allow interactions between values of incompatible types, such as `"str" + 5`. Examples are Python, Java, and C.

Definition 4.2: Weakly-Typed Language

In a weakly-typed programming language, the *type* of a *value* can automatically be converted based on the current needs of a computation.

Weakly-typed languages may implicitly convert the types of incompatible expressions to enable interactions. Here, the `5` in `"str" + 5` may automatically and implicitly be converted to a text string, possibly resulting in `"str5"`. A common example is JavaScript.

Definition 4.3: Statically-Typed Language

In a statically-typed language, each *variable* has a *fixed type* and can only refer to values of that type.

Definition 4.4: Dynamically-Typed Language

In a dynamically-typed language, a *variable* has *no explicit type*. We can store values of different types in a variable.

Python as well as the Bash shell script language would be an examples of a dynamically-typed language. C and Java fall on the statically-typed side.

The original mixture of strongly-typed and dynamically-typed had some advantages for Python. The code is shorter, because you do not need to write the types of variables or parameters. The code looks more elegant and programming becomes easier. Well. At first glance. However, there are also problems.

Listing 4.13: An example of the confusing variable types. (stored in file variable_types_wrong.py; output in Listing 4.14)

```
1 int_var = 1 + 7  # Create an integer variable holding an integer number.
2 print(type(int_var))  # This prints "<class 'int'>".
3
4 int_var = int_var / 3  # / returns a float, but we may expect an int?
5 print(type(int_var))  # This now prints "<class 'float'>".
```

↓ `python3 variable_types_wrong.py` ↓

Listing 4.14: The stdout of the program variable_types_wrong.py given in Listing 4.13.

```
1 <class 'int'>
2 <class 'float'>
```

Let's take a look at program `variable_types_wrong.py` given as Listing 4.13. We declare a variable named `int_var` and store the integer `8` in it. Then we update `int_var` by computing `int_var = int_var / 3`. Back in Section 3.2, you learned that the `//` operator performs an integer division with an `int` result, whereas the division using the `/` operator always returns a `float` [651]. This means that our variable `int_var` now contains a `float`, which is also visible in the output in Listing 4.14.

From the perspective of Python, this is totally fine. The *value* of the variable always has a fixed type, as the language is strongly-typed. The type of the initial value of the variable is `int`. The type of the next value of the variable is `float`. The *variable itself* does not have a fixed type, because the language is dynamically-typed. Therefore storing a `float` in a variable that presently contains an `int` is OK. The program executes and the output appears without error.

However, from the perspective of programming, Listing 4.13 is just plain *wrong*. Imagine that this was not just some random example without meaning. Imagine that this was a part of a really useful program. Imagine that you got this program from some source and try to understand it. If you read this program, then you are likely to overlook that a variable named `int_var` contains a `float`. Even if you notice it, it must strike you as odd.

If you do actually notice it, you may think: "Hm. Why is there a `float` in the variable `int_var`?" Normally, we would expect code to make sense and names for things should be reasonable and reflect the nature of things. Thus, there are at least two possible explanations for this situation: On the one hand, maybe, the original author of this code mistakenly mistook the `/` operator and the `//` operator (see Best Practice 2). Maybe they wanted to do an integer division and accidentally did a floating point division. Depending on what the code later on (in our imaginary larger program) does, it could be very hard to find such an error.

On the other hand, maybe, the author fully well wanted to do a floating point division and expected a `float` to be stored in `int_var`, but chose a misleading name. It could be that they started with code where they originally did an integer division and had only `int`s stored in the variable. Later, they discovered that doing a floating point division would be a better choice, changed the operator, but forgot or simply did not bother to change the name of the variable. In keeping the original name, however, they created dangers: What if another programmer continues to work on this code and, based on the variable's name, expects it to contain an `int` whereas it actually contains a `float`. This could again lead to all sorts of strange errors later on in her code. For example, maybe she tries to do something like `"abcdefg"[int_var]`, i.e., use `int_var` as index in a string. This would work fine if `int_var` was an `int`, but crashes the program now because `int_var` is a `float`.

Best Practice 18

The names we use in program code should clearly reflect our intentions.

If the programmer had followed Best Practice 18, then the only way a `float` could be in a variable named `int_var` would be a bug in the code. The most likely reason would be the mix-up between the `/` operator and the `//` operator. Either way, you will certainly agree that something is wrong with this program. And most certainly, it was just a small oversight, maybe even just a typo.

Unfortunately, since you are not the author of the program, you do not know what is wrong. This code will cause some problem down the line. Many such problems exist in many software projects and they indeed are hard to find [314]. So here, the leniency of Python of allowing us to not specify types comes back to bite us. This whole family of problems results from the fact that Python is dynamically-typed.

Everyone of us makes such mistakes. It is impossible to completely avoid them. Luckily, there are two things that we can do to prevent such situations from passing our "quality control":

1. Use static type checking tools to find such potential errors in our code.
2. Use type hints to annotate variables with their types to *a)* make our intention clearer and *b)* support type checking tools.

And if you are in one of my classes, you better do both. And now we will learn how to do that.

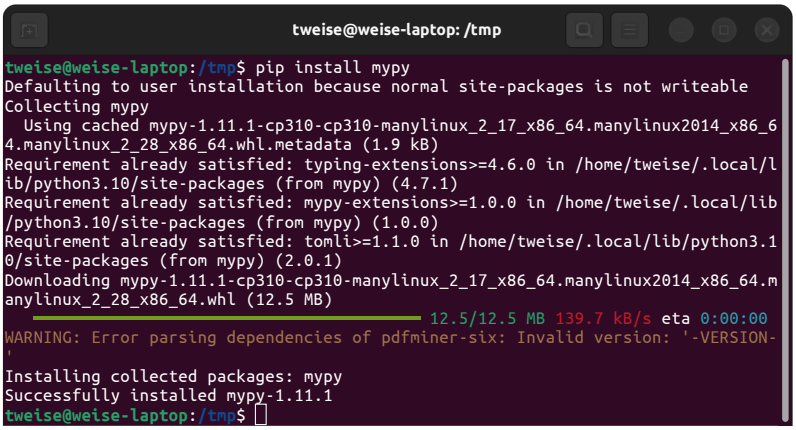


Figure 4.7: Installing Mypy in a Ubuntu terminal via pip (see Section 14.1 for a discussion of how packages can be installed).

Listing 4.15: The results of static type checking with Mypy of the program variable_types_wrong.py given in Listing 4.13. (This is actually output generated by the script Listing 16.1 on page 438.)

```
1 $ mypy variable_types_wrong.py --no-strict-optional --check-untyped-defs
2 variable_types_wrong.py:4: error: Incompatible types in assignment (
    ↪ expression has type "float", variable has type "int")  [assignment]
3 Found 1 error in 1 file (checked 1 source file)
4 # mypy 2.4.1 failed with exit code 1.
```

Listing 4.16: The results of static type checking with Mypy of the program variable_types.py given in Listing 4.11. (This is actually output generated by the script Listing 16.1 on page 438.)

```
1 $ mypy variable_types.py --no-strict-optional --check-untyped-defs
2 Success: no issues found in 1 source file
3 # mypy 2.4.1 succeeded with exit code 0.
```

4.4.3 Static Type Checking

A first step to avoiding any type-related errors in programs is, of course, careful programming. The second step is to use tools that check whether your program code contains ambiguities or errors. In languages like C and Java, the compiler will take care of that for you to a some extend, because these languages are statically and strongly typed.

In Python, which allows for dynamic typing and is an interpreted language, we will use a tool like Mypy [347]. You can install this tool by opening a terminal. Under Ubuntu, you therefore press Ctrl+Alt+T, and under Microsoft Windows, you press ⊞+R, type in cmd, and hit ↵. Then type in pip install mypy and hit ↵. Usually, you will do that in a virtual environment, as discussed later in Section 14.1. The Mypy tool will be installed as illustrated in Figure 4.7.

We can now apply the tool to the program variable_types_wrong.py from Listing 4.13. All we have to do is invoke it in the terminal, giving the program to be checked as argument as well as some additional parameters. In Listing 4.15, we invoke mypy variable_types_wrong.py --no-strict-optional --check-untyped-defs, where variable_types_wrong.py is the (very fitting) name of the program to check. (In some setups, this does not work, and we have to write python3 -m mypy variable_types_wrong.py instead of mypy variable_types_wrong.py.) Indeed, Mypy tells us that something dodgy is going on in the fourth line of that program, i.e., int_var = int_var / 3. It will fail with an exit code of 1. Programs usually return 0 as exit code if everything went well and some non-zero value if something went wrong. And something went wrong, because Mypy found the error.

Listing 4.17: The results of static type checking the program `assignment_wrong.py` with `Mypy` given in Listing 4.7.

```
1 $ mypy assignment_wrong.py --no-strict-optional --check-untyped-defs
2 assignment_wrong.py:12: error: Name "intvar" is not defined; did you mean "
    ↪ int_var"?  [name-defined]
3 Found 1 error in 1 file (checked 1 source file)
4 # mypy 2.4.1 failed with exit code 1.
```

Useful Tool 4

`Mypy` [347] is a static type checking tool for `Python`. This tool can warn you if you, e.g., assign values to a variable that have a different type than the values previously stored in the variable, which often indicates a potential programming error. It can be installed via `pip install mypy`, as illustrated in Figure 4.7 on page 103. You can then apply `Mypy` using the command `mypy fileToScan.py`, where `fileToScan.py` is the name of the file to check (you can also specify a whole directory). We use the `Bash` script given in Listing 16.1 on page 438 to apply `Mypy` to the example programs in this book.

If we instead apply `Mypy` to the completely fine (albeit useless) program `variable_types.py` in Listing 4.11, it will tell us that there is no error, as illustrated in Listing 4.16. So we now have one tool at our hands with which we can check our source code for type-related problems. Notice that this program just checks the source code. It does not change the code and it does not execute our program. It just reads in the code and looks for type-related errors. This will obviously have no impact on the programs performance or speed. It also cannot fix the errors, as it cannot know what the programmer actually intended to do. But knowing that line 4 in Listing 4.13 is probably wrong will help the programmer to fix that error or oversight before passing the program on to someone else.

Best Practice 19

Every program should pass static type checking with tools such as `Mypy` (see Useful Tool 4). Any issue found by the tools should be fixed. In other words, type check the program. If there is an error, fix the error and *type check it again*. Repeat this until no errors are found anymore.

For the sake of completeness, we also apply `Mypy` to the program `assignment_wrong.py` given in Listing 4.7 that we used to illustrate the use of the `PyCharm` IDE in finding bugs. The output given in Listing 4.17 informs us about the same error we encountered back in Section 4.2: *"Name 'intvar' is not defined."* With the IDE and `Mypy`, we now have independent tools that can help us to discover errors in our code. The more such tools we have *and actively use*, the more likely it is that we can produce error-free programs. When reading the text, you may think: *"`Mypy` is additional software. If I want to use it, I need to install it and I will have to learn its command line parameters. Everytime I do use it and apply it to some `Python` code, I need to open the terminal, go into the directory where my program code is located, run the `Mypy` program, and read the output. This is, like, lots of work. Why do I need to bother?"* There are two answers, the first one is *"This will improve the quality of your code."* We elaborate this answer more later. But there also is a second answer:

Best Practice 20

A professional software engineer or programmer or, actually, any professional computer scientist knows many tools and is always keen to learn new tools.

In your professional life, you will learn dozens if not hundreds of tools on different OSes. The more tools you know, the easier it gets to learn and dig into other software if need be. Today, automated builds in Continuous Integration (CI), for example, alone may involve more than a dozen programs. Being able to learning new tools and become proficient with them is thus an essential skill of a programmer.

Listing 4.18: Listing 4.13, but with the variable explicitly hinted as `int`. (stored in file variable_types_wrong_hints_1.py; output in Listing 4.19)

```
1 int_var: int = 1 + 7  # Create a variable hinted as integer.
2 print(type(int_var))  # This prints "<class 'int'>".
3
4 int_var = int_var / 3  # / returns a float, but we may expect an int?
5 print(type(int_var))  # This now prints "<class 'float'>".
```

↓ `python3 variable_types_wrong_hints_1.py` ↓

Listing 4.19: The stdout of the program variable_types_wrong_hints_1.py given in Listing 4.18.

```
1 <class 'int'>
2 <class 'float'>
```

4.4.4 Type Hints

When we discussed the program variable_types_wrong.py in Listing 4.13, we stated that there could be two reasons for the error in the code: Either, the author accidentally mixed-up the two operators `/` and `//` or they chose a misleading name for their variable `int_var`. The problem that any type checking tool faces is that it cannot know the intention of the programmer. It can discover that line 4 is probably wrong, because the variable `int_var`, which former contained an `int`, now gets a `float` value assigned to it. But it cannot find out why that happened.

Oddly enough, the problem of guessing the intention of the programmer exists only to much smaller degree in a statically typed language like C. Here, we *need* to define the type of every variable before assigning a value to it. Therefore, if the programmer would have wanted `int_var` to strictly be an integer, they would have declared it as an integer variable. If they wanted to store `floats` in it, they would have declared it as a `float` variable. The compiler would have seen any malpractice right away. It could tell us that either line 4 is wrong or our initial assignment of an `int` to the variable in line 1 would be converted to a `float`[1].

As already said, this is where the dynamic typing of Python comes back to bite us. It is very convenient for small projects, but as soon as the projects get bigger, it creates a mess. Remember that "real programs" are much more complex than Listing 4.13. Imagine wading through thousands of lines of code to figure out what type a variable has, and, while doing so, remember that Python permits overwriting the contents of a variable with objects of an entirely different type whenever it pleases us.

Realizing that dynamic typing can be a blessing but also a problem, *optional* type hints were introduced into the Python language [482, 592]. In other words, Python *can* support static typing as well [563]. We can now declare the type of a variable *if we want to do so*. This solves the above problem basically entirely and allows us to state our intentions to type checking tools.

When we declare and assign a variable, `my_var = 1` and want to define it as an integer variable, for example, we simply write `my_var: int = 1`. Of course, a type checker will already see that `my_var` should be an integer variable when we assign the integer `1` to it. However, by writing `: int` after its name, we also clearly establish our intention. Then, `my_var` is a variable in which only integers should be stored. And this makes a difference.

If the author of Listing 4.13 had used type hints, they could have written their program differently, as illustrated in Listings 4.18 and 4.20. Both programs as well as the original one produce exactly the same output if we execute them with the Python interpreter. The Python interpreter ignores type hints. It does not care whether we violate them or not in our code. However, if we type-check them with Mypy, we get two different results, namely Listings 4.22 and 4.23, respectively.

In program variable_types_wrong_hints_1.py given as Listing 4.18, we annotate `int_var` as an integer variable by writing `int_var: int`. While the name of the variable already indicates that we want it to hold integers, we now have formally specified this intention. The Mypy type checker tells us in Listing 4.22 basically the same as it stated in Listing 4.15, namely that assigning a `float` to this variable is wrong.

[1] This is one exception that proves the rule in many strongly-typed languages: Integers can be interpreted as floating point numbers where necessary.

Listing 4.20: Listing 4.13, but with the variable explicitly hinted as either `int` or `float` and named appropriately. (stored in file variable_types_wrong_hints_2.py; output in Listing 4.21)

```
my_var: int | float = 1 + 7  # A variable hinted as either int or float.
print(type(my_var))  # This prints "<class 'int'>".

my_var = my_var / 3  # / returns a float, which is OK.
print(type(my_var))  # This now prints "<class 'float'>".
```

↓ `python3 variable_types_wrong_hints_2.py` ↓

Listing 4.21: The stdout of the program variable_types_wrong_hints_2.py given in Listing 4.20.

```
<class 'int'>
<class 'float'>
```

Listing 4.22: The results of static type checking the program variable_types_wrong_hints_1.py with Mypy of the program given in Listing 4.18.

```
$ mypy variable_types_wrong_hints_1.py --no-strict-optional --check-untyped
    ↪ -defs
variable_types_wrong_hints_1.py:4: error: Incompatible types in assignment
    ↪ (expression has type "float", variable has type "int")  [assignment]
Found 1 error in 1 file (checked 1 source file)
# mypy 2.4.1 failed with exit code 1.
```

Listing 4.23: The results of static type checking the program variable_types_wrong_hints_2.py with Mypy of the program given in Listing 4.20.

```
$ mypy variable_types_wrong_hints_2.py --no-strict-optional --check-untyped
    ↪ -defs
Success: no issues found in 1 source file
# mypy 2.4.1 succeeded with exit code 0.
```

However, if the intent of the author was that the variable could hold either an `int` or a `float`, they could have annotated it with the type hint `int | float`. The `|` here is not the "bit-wise or" we learned back in Section 3.2.2, but instead indicates that both `int`s and `float`s are acceptable values for a variable [453].[2] This annotation is used in program variable_types_wrong_hints_2.py in Listing 4.20. A side effect of using this type is that, when writing this code, its author would have realized that the name `int_var` would not be an appropriate name. They would probably have used something like `my_var` instead. Applying Mypy to this program yields the output Listing 4.23, which indicates that everything is OK.

Using type hints together with type checking would have prevented the problems in Listing 4.13 from the start. Either, its author would have discovered the mistake of computing a `float` value instead of an `int`. Or they would have more discovered that the variable name was misleading in the process of communicating their intention by marking the variable to either hold an `int` or a `float`. In both cases, the problem could have been fixed, by using another division operator or by renaming the variable, respectively. And this comes at no cost at all when executing the program, because type hints are only checked by programmers and tools, but not by the interpreter.

Best Practice 21

Always use type hints.

[2] Technically speaking, the type hint specification in PEP 484 [592] allows all variables annotated with `float` to also accept `int` values in section The Numeric Tower. Thus annotating the variable only with `: float` would have been sufficient. However, since `int` is actually *not* a subclass of `float` in Python, this can be confusing. Writing `int | float` gives me the opportunity to introduce the `|` operator ... so this is how we do it.

Listing 4.24: A variant of Listing 4.11 which has been improved by adding type annotations. (src)

```
int_var: int = 1 + 7  # Create an integer variable holding an integer.
print(type(int_var))  # This prints "<class 'int'>".

float_var: float = 3.0  # 3.0 is a float and it is stored in float_var.
print(type(float_var))  # This prints "<class 'float'>".

str_var: str = f"{float_var = }"  # Render an f-string.
print(type(str_var))  # This prints "<class 'str'>".

bool_var: bool = (1 == 0)  # 1 == 0 is False, so a bool is stored.
print(type(bool_var))  # This prints "<class 'bool'>".

none_var: None = None  # We create none_var which, well, holds None.
print(type(none_var))  # This prints "<class 'NoneType'>".
```

Listing 4.25: The results of static type checking with Mypy of the program variable_types_hints.py given in Listing 4.24.

```
$ mypy variable_types_hints.py --no-strict-optional --check-untyped-defs
Success: no issues found in 1 source file
# mypy 2.4.1 succeeded with exit code 0.
```

There are good reasons for always using type hints. While not specifying variable types is very convenient, it comes at a high cost:

> Python's only serious drawbacks are (and thus leaving room for competition) its lack of performance and *that most errors occur [at] run-time*.
>
> — *Paul Jansen [296], 2025*

Type-related errors are one category of mistakes that will become visible only at runtime. At the same time, these are errors that can relatively easily be discovered during static code analysis. They are *much* easier to detect than logical flaws in the program code. Compilers in C or Java would never permit assignments of mismatched datatypes. With type hints and static type checkers like Mypy, we can bring this functionality to Python.

We now annotate our good old program variable_types.py from Listing 4.11 with type hints as a small exercise. The variable `int_var`, in which we want to store the integer value `8`, will be annotated with `: int`. The variable `float_var`, in which we want to store the floating point number `3.0`, will be annotated with `: float`. The variable `str_var`, in which we want to store the string `"float_var = 3.0"`, wird mit will be annotated with `: str`. The variable `bool_var`, in which we want to store the value `False`, will be annotated with `: bool`. And, finally, the variable `none_var`, in which we want to store `None`, will be annotated with `: None`. This produces Listing 4.24, which we can check with Mypy and obtain Listing 4.25.

It is very obvious at this point that including the type of the variable in the variable name is not needed. It is not helpful for any tool. If we want to convey our intention about the type to another programmer who may be reading our code, then type hints are a much better choice. Different from variable names, they also can be interpreted by type checkers.

With type hints, we have brought the advantages of a statically typed programming language to Python. Since they are optional, a programmer can choose whether and where to use them. However, if you are a student attending one of my courses, consider them as mandatory. Their advantage is that they allow us to find many (though by far not all) logical errors. They make the code easier to read and easier to understand. Therefore, from my perspective, Best Practice 21 *always* applies.

You may ask why we emphasize that type hints are important and good for a programming language, which originally did not offer any such thing. We may look at the Python ecosystem at large to get an impression of what much more experienced developers think. First, there are several important tools, like the psycopg [599], the PostgreSQL Python adapter, that are fully annotated with type hints based

on the PEP 484 [592] specification. Second, many other libraries use these annotations at least partially and/or try to ensure that code which is newly contributed to them is annotated, e.g., `Matplotlib` [280], `NumPy` [419], and `Pandas` [431]. The fact that many popular tools use it only *partially* instead of being completely type-hinted is that they simply are older than PEP 484 [592], which is from 2014. Third, some packages like `Scikit-learn` and `SciPy`, for instance, to the best of our knowledge, do not adopt static typing at the time of this writing. Interestingly, in the case of these examples, as reason it is given that using type hints would be very complicated with their existing codebases [6, 101]. The lesson we should learn from this is that

Best Practice 22

It is important to integrate type hints from the very start of each project. The idea to first write code and later annotate it with type hints is wrong.

Finally, it is worth noting that using static type-checkers can even have a positive influence on security aspects of your code, as you can learn in our *Databases* class [611]. Injection attacks such as SQL injection attacks (SQLi attacks) have been an application security concern for decades. Such attacks can be prevented if the queries to DBs are never dynamically constructed by the likes of f-strings but instead are always defined as string literals. `Python` supports the type `LiteralString` for string literals [528]. Implementations of the `Python` DB Application Programming Interface (API), such as `psycopg` [599], can be annotated to only accept such strings. Hence, a type checker could detect and complain if you would try to dynamically construct queries, thus preventing SQLi attacks – but only if you use it... At the time of this writing, `Mypy` does not yet support this functionality, though [599, 660]. Currently, the feature is still too new. We are at the bleeding edge of software development.

4.4.5 Summary

`Python` is a strongly- and dynamically-typed programming language. This gives us much freedom to write code conveniently, but it also allows for a number of possible bugs in our code that is easier to avoid in statically-typed languages. What we want is to write clean, safe, and secure code.

And in this section, you have learned two more building blocks to do just that, in the dynamically-typed environment of `Python`. Type checking and type hints. We add them to proper commenting of code, following style conventions, carefully checking your IDE for warnings, and properly reading any output of crashed programs to locate the erros. Putting all of these things together, we have a solid foundation ensuring that our code is clean. Writing these things so early in the book does feel a bit strange. We have just learned how to use variables and yet, I am already bothering you with things that may look very nit-picky and bean-counter-ish. In many other classes, these topics may appear much later, as afterthoughts, if at all. But I do want that this book provides you an introduction to real-world software development with `Python`. I do want to explain the things that are important if you want to actually and productively write programs.

During my career, I have met many students who can write down a complex algorithm in `Python`. But many of them, many of these bright and excellent students, were unable to find and fix errors in their code. Because they never learned how to do that. Additionally, a large fraction of the student-produced code that I have seen did not have a clear structure. Many program files were hard to read, which even added to the problem of finding errors. They did not know about any tools that can help them.

From my perspective, it is much less important to discuss all the specific features of a programming language. You can easily learn them by yourself, once you understand the basics. I think it's much more important to help you, from the very start, to write *good* code. To understand what makes code *good*. So these topics come early in my book.

4.5 Object Equality and Identity

We use variables to references objects in memory. When we compare two variables, we actually compare the objects they reference. It is clear that two variables can reference objects that are either equal or not equal. They can also reference the same object. These two concepts – equality and identity – must be distinguished.

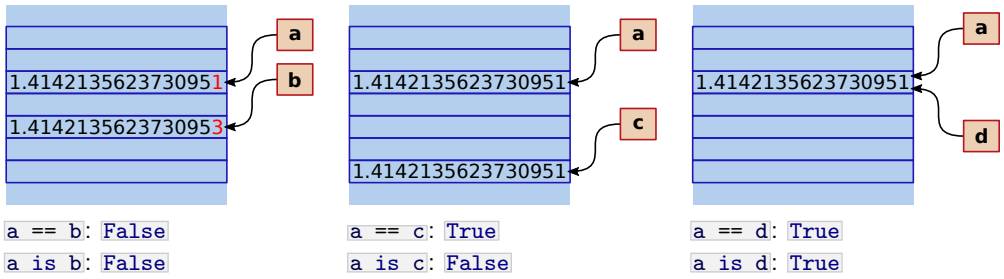


Figure 4.8: A illustration of the concepts of equality (`==`) and identity (`is`) from the perspective of variables.

Listing 4.26: An example of the difference between equality and identity. (stored in file `identity_1.py`; output in Listing 4.27)

```
# We define sqrt_2 to be a constant with the value of square root of 2.
sqrt_2: float = 1.4142135623730951  # We set the value of the variable.
print(f"sqrt_2 = {sqrt_2}")  # We print the value of the variable

# We can also compute the square root using the `sqrt` function.
from math import sqrt  # Import the root function from the math module.
sqrt_2_computed: float = sqrt(2.0)  # Compute the square root of 2.0.
print(f"{sqrt_2_computed = }")  # Print the value.

# Let's compare the computed and the constant value:
print(f"are they equal: {sqrt_2 == sqrt_2_computed}")
print(f"are they the same object: {sqrt_2 is sqrt_2_computed}")
```

↓ `python3 identity_1.py` ↓

Listing 4.27: The stdout of the program `identity_1.py` given in Listing 4.26.

```
sqrt_2 = 1.4142135623730951
sqrt_2_computed = 1.4142135623730951
are they equal: True
are they the same object: False
```

And this can be done rather easily. Imagine that you buy a green jacket, let's call it A. On the next day, you buy another green jacket, which looks exactly like the first one. You call the second jacket B. Now A and B are equal, but they are still two distinct objects. Therefore, it holds that `A == B` but *not* `A is B` (i.e., it holds that `A is not B`) A and B are two names for two distinct but equal objects. Then again, you can put jacket A into your wardrobe and when you take it out tomorrow, start calling it C. Then A and C would be two names that refer to very same jacket, to identical objects. Now both `A == C` and `A is B` are `True`.

Figure 4.8 illlustrates this issue. If we have two variables `a=1.4142135623730951` and `b=1.4142135623730953`, then they have different values and reference different objects (namely, the two different memory cells holding the two different `float` values). In this case, `a == b` and `a is b` will both return `False`.

Now we can also have another variable `c`, which may reference a `float` object that holds the same value as the one referenced by `a`. This value would be stored somewhere else in memory. Then, `a == b` is `True`, because the variables have equal values. However, `a is b` would be `False`, because they still reference different objects.

If I would declare a variable `d = a`, then `d` would point to exactly the same `float` object as `a`. Now, both `a == b` and `a is b` are `True`.

A simple example of the difference between equality and identity is given in program `identity_1.py` as Listing 4.26. Here, we declare a `float` variable and store in it the value of $\sqrt{2}$ (as exactly as it can be represented by a Python `float`, that is), which just so happens to be `1.4142135623730951`. We then import the function `sqrt` from the `math` module and compute `sqrt(2.0)`. We store the result

Listing 4.28: An second example of the difference between equality and identity. (stored in file identity_2.py; output in Listing 4.29)

```
# String and integer literals and identity.
a: str = "Hello World!"
b: str = "Hello World!"
print(f"Are 'a' and 'b' the same object: {a is b}")

c: str = "Hello " + "World!"
print(f"Are 'a' and 'c' the same object: {a is c}")

d: str = "Hello"
d = d + " World!"
print(f"Are 'a' and 'd' the same object: {a is d}")
print(f"Are 'a' and 'd' equal objects: {a == d}")

e: int = 10
mul: int = 5
f: int = (e * mul) // mul
print(f"Are 'e' and 'f' the same object: {e is f}")

g: int = 1_000_000_000_000_000_000
h: int = (g * mul) // mul
print(f"Are 'g' and 'h' the same object: {g is h}")
print(f"Are 'g' and 'h' equal objects: {g == h}")
```

↓ `python3 identity_2.py` ↓

Listing 4.29: The stdout of the program identity_2.py given in Listing 4.28.

```
Are 'a' and 'b' the same object: True
Are 'a' and 'c' the same object: True
Are 'a' and 'd' the same object: False
Are 'a' and 'd' equal objects: True
Are 'e' and 'f' the same object: True
Are 'g' and 'h' the same object: False
Are 'g' and 'h' equal objects: True
```

of this in a second variable, namely `sqrt_2_computed`. We find that `sqrt_2 == sqrt_2_computed` is `True`, whereas `sqrt_2 is sqrt_2_computed` is `False`. The two values are clearly the same, however, they are stored at two different memory locations. The interpreter did not know that the same value as the result of `sqrt(2.0)` was already stored somewhere in memory. Therefore, it allocated the necessary memory for the result of the `sqrt` operation and stored it there. We now have two variables that reference two objects which both hold the same value.

First Time Readers and Novices: In the rest of this subsection, we discuss (albeit superficially) how objects are cached and reused. First-time readers are welcome to skip over the rest of this subsection.

A slightly more involved example of the issue of object equality and identity is given in program identity_2.py as Listing 4.28. You see, when the Python interpreter parses the program code file, it will allocate the memory and store the values of constants and literals. Listing 4.28, we declare `a = "Hello World!"` and then `b = "Hello World!"`. In other words, `a` and `b` will receive the same value. As the output in Listing 4.29 shows, the Python interpreter is clever enough not to store this value twice in memory. Instead, `a` and `b` will point to the same object, i.e., `a is b` holds. Interestingly, even if we subsequently do `c = "Hello "+ "World!"`, the interpreter is still clever enough to recognize that this constant expression yields the same result already stored in `a`. It basically computes the result of the concatenation of the two constant strings and discovers this identity. Therefore, `a is c` will also be `True`.

If we use a variable in such an expression, it does no longer work, though: First setting `d = "Hello"` and then setting `d = d + "World!"` will yield an equal result (`a == d` is `True`), but it will be stored

somewhere else in memory (`a is d` is `False`). The interpreter, at least in its current version, cannot infer value identities of results of computations that involve values that are not constants.

Well, sometimes it can. It seems that Python, like Java, caches a set of small integers.[3] It is obvious that we need small integer numbers again and again and again in our programs. Allocating a new object whenever we use the values `1` or `0` would be wasteful. In Listing 4.28, we calculate the result of the computation `f = (e * mul)// mul`, where `e` is a variable with value `10` and `mul` is a variable with value `5`. The result is obviously `10` as well, i.e., a small integer. And it is identical to `e`, i.e., `e is f` is `True` – despite begin the result of a computation including only variables.

We now replace `10` with a large value, say we set `g = 1_000_000_000_000_000_000`. We compute `h = (g * mul)// mul`. Of course, `g == h` is `True`. However, `g is h` is `False`. Because this integer value is too large to be cached and a new object is generated.

4.5.1 Summary

We can distinguish *equality* from *identity*. Equality is tested with the operator `==` and inequality is tested with operator `"!="`. Identity is tested with operator `is` and un-identity with operator `is not`. Two distinct objects can either be equal or unequal, but never identical. Two identical objects are always equal, i.e., `A is B`⇒`A == B`.

4.6 Summary

Variables allow us to store and access data. After gaining this ability, we finally can begin to write "real" programs. We do no longer just execute single commands in the Python console. Instead, we can write Python program files (with the name suffix `.py`) that perform computations in multiple steps. Our implementation of the ideas of LIU Hui (刘徽) to approximate the ratio of the circumference of a circle to its diameter, i.e., the number π, Listing 4.3, was a first example. Here, we refined the approximation in several steps and ended up with some pretty good estimate.

The elegant syntax of Python allows us to assign and reassign variables. We can swap the values of two variables `a` and `b` by writing `a, b = b, a`.

However, the elegant and simply syntax also has drawbacks. It does not force us to define the type of a variable. As a result, it does not prevent us from first storing an integer inside a variable and then storing a string into it. This can lead to problems and confusion. The Python interpreter does not care about such potential issues and will happy obey any command we give to it. Therefore, we need additional tools: Type checkers like Mypy help us to detect such potential problems. They can tell us if we try to store something in a variable which is of a different type than the previous usage of the variable. These programs are applied to the program files, read them, and look for errors.

These type checkers cannot guess whether the previous usage was wrong or our attempt to overwrite the value with one of a different type. Or maybe all is good and we intended from the beginning to allow the variable to store different kinds of objects. To allow us to create some clarity, type hints were introduced. We can annotate variables with type hints that, basically, specify the type of objects we want to be stored in the variables. This gives the dynamically typed Python language the same expressiveness of statically typed languages like Java. The Python interpreter does not care about this and, again, happily ignores all such type hints and obeys all of our commands. Static type checking tools, like Mypy, however, now can see our intend and provide better error messages when they detect an issue.

As the final topic in this section, we tackled the issue of equality versus identity. Two variables `x` and `y` are identical if they point to the same object. Then, `x is y` and `y is x` will be `True`. Even if this is not the case, i.e., `x is y` is `False`, then the variables do not point to one and the same object. However, they could still point to two objects which have the same value. Nothing can prevent me from storing `"Hello!"` multiple times at multiple different locations in memory. I can let `x` and `y` point to two different locations both storing the same string `"Hello!"`. Then, `x is y` may be `False`, but `x == y` could be `True`.

Having discussed all of these issues, we can now depart from the interesting topic of variables.

[3] Some sources say that all integers between `-5` and `256` are cached. However, this is highly implementation and configuration specific and may be entirely different on your machine.

Chapter 5

Collections

We already learned about simple datatypes, like integer and floating point numbers, strings, and Boolean values. We also learned how we can use a variable to store one instance (object) of such a datatype. However, in many cases, we do not just want to store a single object. Often, we want to store and process *collections* of objects [87, 126, 142]. Python offers us four basic types of collections:

The first one, *lists*, are mutable sequences of objects. We can create a `list` variable `my_list` composed of the three strings `"ax"`, `"by"`, and `"cz"` by writing `my_list = ["ax", "by", "cz"]`. Like the characters in strings, we can access the elements of lists using square brackets. For instance, `my_list[0]` would return the first element of `my_list`, namely `"ax"`. Different from strings in Python and arrays in other programming languages, lists support all kinds of modification operations, including the change, insertion, appending, and deletion of elements. In Section 5.1 we learn more about lists and we also get to know a new static code analysis tool that can help us to detect programming issues.

The second type of collections is the *tuple*. Tuples are similar to lists, with two main differences: First, tuples are *immutable*, which means once a tuple is created, it cannot be changed anymore. Second, the semantics of tuples allows for them to contain objects of different types, whereas lists should only contain elements of a single type. Tuples are created using parentheses, so `my_tuple = (1, 2, "z")` creates an instance of the datatype `tuple` with the name `my_tuple` that contains the two integers `1` and `2` as well as the string `z`. The elements can be accessed using square brackets, so `my_tuple[1]` gives us the second element, namely `2`. We learn more about tuples in Section 5.3.

The third type of collection in Python is the datatype `set`, which implements the mathematical notion of *sets*. A set can contain each element at most once. The methods for modifying sets and for checking whether elements are contained in them are particularly fast. Sets are created using curly braces, i.e., `my_set = {1.2, 2.3, 4.5, 2.3}` would create the set `my_set` with the three numbers `1.2`, `2.3`, and `4.5`. Notice that `2.3` would appear only once in the set. Sets cannot be indexed, but like lists and tuples they support the `in` operator and `1.2 in my_set` returns `True` while `1.3 in my_set` is `False`. We will discuss sets in Section 5.4.

The fourth and final type of collection that we will discuss here are *dictionaries*. Represented by the datatype `dict`, they are mappings between keys and values, similar to hash tables in other programming languages. They, too, are created using curly braces which, however, contain key-value pairs. For example, `my_dict = {"pi": 3.1416, "e": 2.7183, "phi": 1.618}` creates a dictionary which maps the strings `"pi"`, `"e"`, and `"phi"` to the values `3.1416`, `2.7183`, and `1.618`, respectively. The value of a key can be retrieved using the square-bracket indexing, i.e., `my_dict["e"]` would return `2.7183`. They are discussed in Section 5.5.

5.1 Lists

A `list` is a mutable sequence of objects which can be accessed via their index [500]. They work very similar to the strings we already discussed in Section 3.6, but instead of (only) characters, they can contain any kind of objects and they can be modified.

With program `lists_1.py` in Listing 5.1, we provide some first examples of using lists. A list can be defined by simply writing its contents, separated by `,` inside square brackets `[...]`. `["apple", "pear", "orange"]` creates a list with three elements, namely the strings `"apple"`, `"pear"`, and `"orange"`. If we want to store a list in a variable, then we can use the type

Listing 5.1: A first example of using lists in Python: creating, indexing, printing of and appending elements and other lists to lists. (stored in file lists_1.py; output in Listing 5.2)

```
"""An example of creating, indexing, and printing lists."""

fruits: list[str] = ["apple", "pear", "orange"]  # Create List.
print(f"We got {len(fruits)} fruits: {fruits}")  # Print length and list

fruits.append("cherry")  # Append one element at the end of a list.
print(f"There now are {len(fruits)} fruits: {fruits}")

vegetables: list[str] = ["onion", "potato", "leek"]  # Create list.
print(f"The vegetables are: {vegetables}.")  # Print the list.

food: list[str] = []  # Create an empty list.
food.extend(fruits)  # Append all elements of `fruits` to `food`.
food.extend(vegetables)  # Append all elements of `vegetables` to `food`
print(f"Fruits and vegetables: {food}")  # Print the new list.
print(f"len(food) = {len(food)}")  # Print the length of list `food`.
print(f"{food[0] = }")  # Print the first element of `food`.
print(f"{food[1] = }")  # Print the second element of `food`.
print(f"{food[2] = }")  # Print the third element of `food`.
print(f"{food[-1] = }")  # Print the last element of `food`.
print(f"{food[-2] = }")  # Print the second-to-last element.
print(f"{food[-3] = }")  # Print the third-to-last element.

del food[1]  # Delete the element at index 1 from list `food`.
print(f"Food is now: {food}.")  # Print the list again.
```

↓ python3 lists_1.py ↓

Listing 5.2: The stdout of the program lists_1.py given in Listing 5.1.

```
We got 3 fruits: ['apple', 'pear', 'orange']
There now are 4 fruits: ['apple', 'pear', 'orange', 'cherry']
The vegetables are: ['onion', 'potato', 'leek'].
Fruits and vegetables: ['apple', 'pear', 'orange', 'cherry', 'onion', '
    ↪ potato', 'leek']
len(food) = 7
food[0] = 'apple'
food[1] = 'pear'
food[2] = 'orange'
food[-1] = 'leek'
food[-2] = 'potato'
food[-3] = 'onion'
Food is now: ['apple', 'orange', 'cherry', 'onion', 'potato', 'leek'].
```

hint `list[elementType]` where `elementType` is to be replaced with the type of the list elements [340]. `fruits: list[str] = ["apple", "pear", "orange"]` therefore creates the list `fruits` with the contents listed above. It also tells any automated type checking tool and other programmers that we intend that only `str` values should be stored inside the list.

The length of a list can be obtained using the `len` function. `len(fruits)` will therefore return the value `3`. We can use lists in f-strings just like any other datatype. The string representation of `fruits` which then would be used is simply `"['apple', 'pear', 'orange']"`.

We can add single elements to a list by using the `append` method. Invoking `fruits.append("cherry")` will append the string `"cherry"` to the list `fruits`. The list then equals `["apple", "pear", "orange", "cherry"]` and has `len(fruits)== 4`.

Of course we can have multiple lists in a program. In Listing 5.1, we now create the second list `vegetables` with the three elements `"onion"`, `"potato"`, and `"leek"`.

An empty list is created with expression `[]`, which consists of just the square brackets with no

Listing 5.3: A second example of using lists in Python: inserting and deleting elements, sorting and reversing lists. (stored in file lists_2.py; output in Listing 5.4)

```
"""An example of creating, modifying, sorting, and copying lists."""

numbers: list[int] = [1, 7, 56, 2, 4]  # Create the list.
print(f"The numbers are: {numbers}.")  # Print the list.

print(f"is 7 in the list: {7 in numbers}")  # Check if 7 is in the list.
print(f"is 2 NOT in the list: {2 not in numbers}")  # the opposite check
print(f"7 ist at index {numbers.index(7)}.")  # Search for number 7.
print(f"2 ist at index {numbers.index(2)}.")  # Search for number 2.

numbers.insert(2, 12)  # Insert the number 12 at index 2...
print(f"After inserting 12, the numbers are: {numbers}.")  # and print.

numbers.remove(56)  # Remove the number 56 from the list.
print(f"After removing 56, numbers are: {numbers}.")  # Print the list.

numbers.sort()  # Sort the list `numbers` in place.
print(f"The sorted numbers are: {numbers}.")  # Print the list.

numbers.reverse()  # Reverse the order of the list elements.
print(f"The reversed numbers are: {numbers}.")  # And print the list.

cpy: list[int] = list(numbers)  # Create a copy of the list `numbers`.
print(f"cpy == numbers: {cpy == numbers}.")  # Indeed, `cpy == numbers`.
print(f"cpy is numbers: {cpy is numbers}.")  # No, `cpy is not numbers`.

del cpy[0]  # We change `cpy`, but `numbers` remains unchanged.
print(f"cpy == numbers: {cpy == numbers}.")  # Now, `cpy != numbers`.
print(f"cpy is numbers: {cpy is numbers}.")  # And `cpy is not numbers`.
print(f"cpy is not numbers: {cpy is not numbers}.")  # indeed, it is not
```

↓ `python3 lists_2.py` ↓

Listing 5.4: The stdout of the program lists_2.py given in Listing 5.3.

```
The numbers are: [1, 7, 56, 2, 4].
is 7 in the list: True
is 2 NOT in the list: False
7 ist at index 1.
2 ist at index 3.
After inserting 12, the numbers are: [1, 7, 12, 56, 2, 4].
After removing 56, numbers are: [1, 7, 12, 2, 4].
The sorted numbers are: [1, 2, 4, 7, 12].
The reversed numbers are: [12, 7, 4, 2, 1].
cpy == numbers: True.
cpy is numbers: False.
cpy == numbers: False.
cpy is numbers: False.
cpy is not numbers: True.
```

contents inside. We can append *all* the elements of one collection to a list by using the `extend` method. We start with the empty list `food` and then invoke `food.extend(fruits)`. Now all the contents of the list `fruits` are appended to `food`. We then invoke `food.extend(vegetables)`, which will add all the elements from the list `vegetables` to `food` as well. `fruits` and `vegetables` remain unchanged during this procedure, but `food` now contains all of their elements as well. It contains all seven fruits and vegetables and its `len(food)` is therefore `7`.

We can access the elements of a list by their index, again in the same way we access the characters in a string. `food[0]` returns the first element of the list `food`, which is `"apple"`. `food[1]` returns the second element of the list `food`, which is `"pear"`, and so on. We can also access the elements using

Listing 5.5: A third example of using lists in Python: slicing, adding, and multiplying lists. (stored in file lists_3.py; output in Listing 5.6)

```
"""An example of more operations with lists."""

lst1: list[int] = [1, 2, 3, 4]  # create first list
lst2: list[int] = [5, 6, 7]  # create second list
lst3: list[int] = lst1 + lst2  # lst3 = concatenation of lst1 and lst2.
print(f"lst3 = lst1 + lst2 == {lst3}")  # [1, 2, 3, 4, 5, 6, 7]

lst4: list[int] = lst2 * 3  # lst4 = lst2, repeated three times.
print(f"lst4 = lst2 * 3 == {lst4}")  # [5, 6, 7, 5, 6, 7, 5, 6, 7]

lst5: list[int] = lst4[2:-2]  # lst5 = lst4 from index 2 to 3rd from end
print(f"lst5 = lst4[2:-2] == {lst5}")  # [7, 5, 6, 7, 5]

lst6: list[int] = lst4[1::2]  # start at index 1, take every 2nd element
print(f"lst6 = lst4[1::2] == {lst6}")  # [6, 5, 7, 6]

# Start copying lst4 at last element, move backwards take every 2nd
# element, and stop right before index=3.
lst7: list[int] = lst4[-1:3:-2]
print(f"lst7 = lst4[-1:3:-2] == {lst7}")  # [7, 5, 6]

lst7[1] = 12  # Modify the slice lst7 originally from lst4.
print(f"{lst4 = }, {lst7 = }")  # Shows that lst4 remains unchanged.

a, b, c = lst2  # store the three elements of lst2 into variables
print(f"{a = }, {b = }, {c = }")  # a=5, b=6, c=7
```

↓ python3 lists_3.py ↓

Listing 5.6: The stdout of the program lists_3.py given in Listing 5.5.

```
lst3 = lst1 + lst2 == [1, 2, 3, 4, 5, 6, 7]
lst4 = lst2 * 3 == [5, 6, 7, 5, 6, 7, 5, 6, 7]
lst5 = lst4[2:-2] == [7, 5, 6, 7, 5]
lst6 = lst4[1::2] == [6, 5, 7, 6]
lst7 = lst4[-1:3:-2] == [7, 5, 6]
lst4 = [5, 6, 7, 5, 6, 7, 5, 6, 7], lst7 = [7, 12, 6]
a = 5, b = 6, c = 7
```

the end of the list as reference: `food[-1]` returns the last element of the list `food`, which is `"leek"`. `food[-2]` returns the second-to-last element of the list `food`, which is `"potato"`, and so on.

Finally, elements can also be deleted from the list by their index. `del food[1]` deletes the second element from the list `food`. The second element is `pear` and if we print `food` again, it has indeed disappeared.

In program lists_2.py in Listing 5.3, we illustrate some more operations on lists. We begin again by creating a list, this time of numbers: `numbers: list[int] = [1, 7, 56, 2, 4]` creates (and type-hints) a list of five integers. If we want to know whether an element is included in a list, we can use the `in` operator. `7 in numbers` returns `True` if `7` is located somewhere inside the list `numbers` and `False` otherwise. The `not in` operator inquires the exact opposite: `2 not in numbers` becomes `True` if `2` is *not* in the list `numbers` and is `False` if it is in the list. If we want to know at which index a certain element in the list is located, we can use the `index` method. `numbers.index(7)` will search where the number `7` is located inside `numbers`. Since it is the second element and indices start at 0, it returns `1`. Similarly, `numbers.index(2)` returns `3`, because `numbers[3] == 2`.

The `insert` method allows us to insert an element at a specific index. The elements which are currently at that location or at higher indices are all shifted up one slot. `numbers.insert(2, 12)` will insert the number `12` at index `2` into the list `numbers`. The element `56` which currently occupies this spot is moved to index `3`, which means that the `2` located at this place is moved to index `4`, which

means that the value `4` which right now is stored in this location will move to index `5`. The list `numbers` now looks like this: `[1, 7, 12, 56, 2, 4]`.

If we want to remove a specific element from the list without knowing its location, the `remove` method will do the trick. `numbers.remove(56)` searches through the list `numbers` for the element `56` and, once it finds it, deletes it. The list becomes `[1, 7, 12, 2, 4]`. Notice that `remove` only deletes the single first occurrence of the element. If there had been two `56`es in our list, only the first one would have been removed.

We can sort a list in place by using the `sort` method. `numbers.sort()` sorts the list `numbers`, which then becomes `[1, 2, 4, 7, 12]`. Similarly, we can reverse a list, i.e., make the last element become the first, the second-to-last element the second, and so on, by using the method `numbers.reverse`. Reversing the list `numbers` after we sorted it will turn it into `[12, 7, 4, 2, 1]`.

If we want to create a list copy of an existing sequence, we can just invoke the constructor `list` directly. `cpy: list[int] = list(numbers)` creates the new list `cpy` which has the same contents as `numbers`. This means that `cpy == numbers` will be `True`, because `cpy` is an exact copy of `numbers`. `cpy is numbers`, however, is `False`. They are not the same object.

We can change `cpy` by deleting its first element via `del cpy[0]`. `numbers` will be unaffected by this and stays unchanged. Now, both `cpy == numbers` and `cpy is numbers` will be `False`. By the way, in the same manner as `not in` is the opposite of the `in` operator, `not is` is the opposite of `is`: `cpy is not numbers` yields `True` because `cpy` is not the same object as `numbers`.

Program `lists_3.py` given in Listing 5.5 continues our journey through the magical land of `Python lists`. You can add two lists `a` and `b` using `+`, i.e., do `c = a + b`. The result is a new list which contains all elements of the first list `a` followed by all the elements from the second list `b`. Therefore, the expression `[1, 2, 3, 4] + [5, 6, 7]` results in `[1, 2, 3, 4, 5, 6, 7]`. Similarly, if you multiply a list by an integer `n`, you get a new list which equals the old list `n` times concatenated. `[5, 6, 7] * 3` therefore yields `[5, 6, 7, 5, 6, 7, 5, 6, 7]`. You would get the same result doing `3 * [5, 6, 7]`.

In Section 3.6.1, we discussed string slicing. Lists can be sliced in pretty much the same way [500]. When slicing a list `l` or a string, you can provide either two or three values in the square brackets, i.e., either do `l[i:j]` or `l[i:j:k]`. If `j < 0`, then it is replaced with `len(l) - j`. In both the two and three indices case, `i` is the inclusive start index and `j` is the exclusive end index, i.e., all elements with index `m` such that `i <= m < j`. In other words, the slice will contain elements from `l` whose index is between `i` and `j`, including the element at index `i` but *not* including the element at index `j`. If a third index `k` is provided, then it is the step length. `l[i:j:k]` selects all items at indices `m` where `m = i + n*k`, `n >= 0` and `i <= m < j`. If `i` is omitted, i.e., if `:j:k` is provided, then `i=0` is assumed. If `j` is omitted, i.e., if `i::k` is provided, then `j=len(l)` is assumed.

If you have a list `lst4 = [5, 6, 7, 5, 6, 7, 5, 6, 7]`, then the slice `lst4[2:-2]` will return a new list which contains all the elements of `lst4` starting from index `2` and up to (excluding) the second-to-last element. The slice `lst4[1::2]` starts at index `1`, continues until the end of the list, and adds every second element. This results in `[6, 5, 7, 6]`. As a final example, consider the slice `lst4[-1:3:-2]`. It will begin creating the new list at the last element. The step-length is `-2`, so it will move backwards and add every second element to the new list. It stops adding elements before reaching index `3`. Therefore, the result will be the new list `lst7 = [7, 5, 6]`.

Notice that the slices we create are independent copies of ranges of the original lists. The list `lst7` is a slice from the list `lst4`. If we modify it, e.g., set `lst7[1] = 12`, then we set the second element of `lst7` to `12`. `lst7` becomes `[7, 12, 6]`. Now, the second element of `lst7` originally is the seventh element of `lst4`, namely the `5` located at index `6`, which is equivalent to index `-3`. You may wonder whether this element now also has changed. It did not. `lst4` remains unchanged by any operation on the independent copied slice `lst7`.

An interesting functionality is also list unpacking. In Section 3.6.1, the list `lst2` contains the three elements `[5, 6, 7]`. If we know the number of elements in the list in our program, then we can assign them to exactly the same number of variables. `a, b, c = lst2` creates and assigns values to three variables `a=5`, `b=6`, and `c=7` by unpacking the list `lst2`. Of course, this only works if list `lst2` has exactly length 3.

Listing 5.7: A program processing lists which exhibits some subtle errors and inefficiencies. (stored in file `lists_error.py`; output in Listing 5.8)

```
my_list: list[str] = list([1, 2, 3])
print(my_list)
```

↓ `python3 lists_error.py` ↓

Listing 5.8: The stdout of the program `lists_error.py` given in Listing 5.7.

```
[1, 2, 3]
```

Listing 5.9: The results of static type checking with Mypy of the program given in Listing 5.7.

```
$ mypy lists_error.py --no-strict-optional --check-untyped-defs
lists_error.py:1: error: List item 0 has incompatible type "int"; expected
    ↪ "str"  [list-item]
lists_error.py:1: error: List item 1 has incompatible type "int"; expected
    ↪ "str"  [list-item]
lists_error.py:1: error: List item 2 has incompatible type "int"; expected
    ↪ "str"  [list-item]
Found 3 errors in 1 file (checked 1 source file)
# mypy 2.4.1 failed with exit code 1.
```



Figure 5.1: Installing Ruff in a Ubuntu terminal via pip (see Section 14.1 for a discussion of how packages can be installed).

5.2 Interlude: The Linter Ruff

Recently, we learned that static code analysis tools can help us to discover subtle problems in our programs. Obviously, when dealing with more complex data structures like lists, there are other potential problems, more mistakes that one could make. Let us look at the very short example program `lists_error.py` in Listing 5.7. The program consists of only two lines, `my_list: list[str] = list([1, 2, 3])` and `print(my_list)`. It does not have any *error* in the strict sense. We can execute it just fine and it will produce the output `[1, 2, 3]` as shown in Listing 5.8.

However, upon closer inspection, we discover some issues. In a first step, we would apply Mypy (as Useful Tool 4) to check for problems with the types of variables. And indeed, Listing 5.9 shows us *three* errors! We defined `my_list` as a list of strings by using the type hint `list[str]`. However, we then set its value to be a list of three integer numbers (hence, three errors). This is clearly wrong.

But is this all that's wrong? Mypy is a tool that looks for type-related errors and *only* for type-related errors. But did we commit *only* type-related errors? Are there tools that help us to find *other* errors? Linters are tools for analyzing program code to identify bugs, problems, vulnerabilities, and inconsistent code styles [304, 488]. As promised in the title of this section, we will also use another tool to analyze this program: the linter Ruff.

Listing 5.10: The results of linting with Ruff of the program lists_error.py given in Listing 5.7. (We used the script given in Listing 16.2 on page 440 to apply Ruff.)

```
$ ruff check --target-version py312 --select=A,AIR,ANN,ASYNC,B,BLE,C,C4,COM
    ↪ ,D,DJ,DTZ,E,ERA,EXE,F,FA,FIX,FLY,FURB,G,I,ICN,INP,ISC,INT,LOG,N,NPY,
    ↪ PERF,PIE,PLC,PLE,PLR,PLW,PT,PYI,Q,RET,RSE,RUF,S,SIM,T,T10,TD,TID,TRY,
    ↪ UP,W,YTT --ignore=A005,ANN001,ANN002,ANN003,ANN204,ANN401,B008,B009,
    ↪ B010,C901,D203,D208,D212,D401,D407,D413,INP001,N801,PLC2801,PLR0904,
    ↪ PLR0911,PLR0912,PLR0913,PLR0914,PLR0915,PLR0916,PLR0917,PLR1702,
    ↪ PLR2004,PLR6301,PT011,PT012,PT013,PYI041,RUF100,S,T201,TRY003,UP035,W
    ↪  --line-length 79 lists_error.py
D100 Missing docstring in public module
--> lists_error.py:1:1
C410 Unnecessary list literal passed to `list()` (remove the outer call to
    ↪ `list()`)
 --> lists_error.py:1:22
1 | my_list: list[str] = list([1, 2, 3])
  |                      ^^^^^^^^^^^^^^
2 | print(my_list)
help: Remove outer `list()` call
  - my_list: list[str] = list([1, 2, 3])
1 + my_list: list[str] = [1, 2, 3]
2 | print(my_list)
note: This is an unsafe fix and may change runtime behavior
Found 2 errors.
1 unsafe fix available with `--fix --unsafe-fixes`.
# ruff 0.17.3 failed with exit code 1.
```

Useful Tool 5

Ruff is a very fast Python linter that checks the code for all kinds of problems, ranging from formatting and style issues over missing documentation to performance problems and potential errors [378]. It can be installed via `pip install ruff` as shown in Figure 5.1 on page 117. You can then apply Ruff using the command `ruff check fileToScan.py`. We provide a script for using Ruff with a reasonable default configuration in Listing 16.2 on page 440.

Let us apply Ruff to the program lists_error.py given in Listing 5.7. This can be done by executing the command `ruff check myfile.py`, where `myfile.py` is the file to be checked (you can also specify a directory). Ruff has many additional parameters, which are explained at [377, 378]. For example, if we want to analyze our code with respect to version 3.12 of Python, we would specify the command line argument `--target-version py312`. With the optional `--select=...` argument, we can list additional groups of rules to be checked. As you can see, we select quite a few! Check [378] under `https://docs.astral.sh/ruff` in the *Rules* section for the complete list. Sometimes, we wish to ignore some rules which do not apply to our programming scenario. This can be done using the optional `--ignore=...` argument. You can find the complete command that we used to execute Ruff in the top part of Listing 5.10. Because it is a bit long, you will probably want to create your own shell script for running Ruff to avoid always typing this lengthy command.

As Ruff does not check for type-related errors at the time of this writing. But – as you can see in Listing 5.10 – it does find two other issues with our program file: First, it complains that any python file should start with a string specifying the purpose of the file. The use of such docstrings makes it easier for other programmers to understand what is done by which file in projects that are composed of multiple Python scripts.

Listing 5.11: The corrected version of Listing 5.7, taking into account the information given by Mypy in Listing 5.9 and Ruff in Listing 5.10. (stored in file lists_fixed.py; output in Listing 5.12)

```
"""
A fixed version of the original, erroneous program.

The original program was only two lines, namely:

> my_list: list[str] = list([1, 2, 3])
> print(my_list)

There were three errors:

1. mypy will detect that we store integers in a list of str.
2. ruff finds the missing docstring at the program head.
3. ruff finds that writing [1, 2, 3] is better than list([1, 2, 3]).

We now fix it here.
"""
my_list: list[int] = [1, 2, 3]
print(my_list)
```

↓ python3 lists_fixed.py ↓

Listing 5.12: The stdout of the program lists_fixed.py given in Listing 5.11.

```
[1, 2, 3]
```

Listing 5.13: The results of static type checking with Mypy of the program given in Listing 5.11.

```
$ mypy lists_fixed.py --no-strict-optional --check-untyped-defs
Success: no issues found in 1 source file
# mypy 2.4.1 succeeded with exit code 0.
```

Best Practice 23

Each Python file should start with a string describing its purpose [232]. This can either be a single line, like a headline, or a longer text. In the second case, the first line must be a headline, followed by an empty line, followed by the rest of the text. Either way, it must be a string delimited by `"""..."""` [232, 594].

Additionally, Ruff finds that writing `list([1, 2, 3])` is actually useless waste of speed and memory: It basically creates a list as literal via `[1, 2, 3]`. Then it immediately makes a copy of it via the `list` function wrapped around the list specification. We can leave this outer call to `list` away.

In Listing 5.11 we implement the three recommendations from the two tools as program lists_fixed.py. We change the type hint of the list to `list[int]`, which solves the type confusion that Mypy discovered. We remove the useless copying of the list as Ruff recommended. Finally, we add a proper docstring at the top of the file, in which we even document the changes we applied. The output Listing 5.12 of the new program remains the same. But now, both tools are satisfied, as shown in Listings 5.13 and 5.14. And our program is much clearer and faster.

Well, this was only a two-line program. But ask yourself: Did you spot the incorrect type hint when you read the program? Maybe. But did you see that we actually created a list and then copied it instead of using it directly? (The docstring I give you, no chance of seeing that as we did not mention it before.)

Now imagine that your job would be to work on a program with thousands of lines that was developed by a colleague. And remember that our program could be executed flawlessly, despite the errors. The errors would like become problems later, though. Wouldn't you love it if that colleague had thoroughly documented and type-hinted and checked their code? Be that colleague.

Listing 5.14: The results of static type checking with `Ruff` of the program given in Listing 5.11.

```
$ ruff check --target-version py312 --select=A,AIR,ANN,ASYNC,B,BLE,C,C4,COM
    ↪ ,D,DJ,DTZ,E,ERA,EXE,F,FA,FIX,FLY,FURB,G,I,ICN,INP,ISC,INT,LOG,N,NPY,
    ↪ PERF,PIE,PLC,PLE,PLR,PLW,PT,PYI,Q,RET,RSE,RUF,S,SIM,T,T10,TD,TID,TRY,
    ↪ UP,W,YTT --ignore=A005,ANN001,ANN002,ANN003,ANN204,ANN401,B008,B009,
    ↪ B010,C901,D203,D208,D212,D401,D407,D413,INP001,N801,PLC2801,PLR0904,
    ↪ PLR0911,PLR0912,PLR0913,PLR0914,PLR0915,PLR0916,PLR0917,PLR1702,
    ↪ PLR2004,PLR6301,PT011,PT012,PT013,PYI041,RUF100,S,T201,TRY003,UP035,W
    ↪  --line-length 79 lists_fixed.py
All checks passed!
# ruff 0.17.3 succeeded with exit code 0.
```

Best Practice 24

Use many static code analysis tools and use them always. They can discover a wide variety of issues, problems, or potential improvements. They can help you to keep your code clean and to enforce a good programming style. Do not just apply them, but also *implement* their suggestions wherever possible.

5.3 Tuples

Tuples are very similar to lists, with three differences: First, they are immutable. You cannot add, delete, or change the elements of a tuple. Second, tuples are defined using parentheses instead of

Listing 5.15: A first example of using tuples in `Python`: creating, indexing, and printing of tuples. (stored in file `tuples_1.py`; output in Listing 5.16)

```
"""An example of creating, indexing, and type-hinting of tuples."""

fruits: tuple[str, str, str] = ("apple", "pear", "orange")
print(f"We got {len(fruits)} fruits: {fruits}.")

veggies: tuple[str, ...] = ("onion", "potato", "leek", "garlic")
print(f"The vegetables are: {veggies}.")  # Print the tuple.

print(f"{veggies[0] = }")   # first element of `veggies`.
print(f"{veggies[1] = }")   # second element of `veggies`.
print(f"{veggies[-1] = }")  # last element of `veggies`.
print(f"{veggies[-2] = }")  # second-to-last element of `veggies`.

print(f"is pear in fruits:  {'pear' in fruits}")
print(f"is pear in veggies: {'pear' in veggies}")
print(f"apple is at index {fruits.index('apple')} in fruits.")
```

↓ `python3 tuples_1.py` ↓

Listing 5.16: The stdout of the program `tuples_1.py` given in Listing 5.15.

```
We got 3 fruits: ('apple', 'pear', 'orange').
The vegetables are: ('onion', 'potato', 'leek', 'garlic').
veggies[0] = 'onion'
veggies[1] = 'potato'
veggies[-1] = 'garlic'
veggies[-2] = 'leek'
is pear in fruits:  True
is pear in veggies: False
apple is at index 0 in fruits.
```

Listing 5.17: A second example of using tuples in Python: tuples with elements of different types and tuple unpacking. (stored in file tuples_2.py; output in Listing 5.18)

```
"""An example of creating tuples of mixed types."""

mixed: tuple[str, int, float] = ("apple", 12, 1e25)  # mixed types
print(f"The mixed tuple is {mixed}.")  # Print the tuple.

other: tuple[str, int, float] = ("pear", 1, 1.2)  # second such tuple
print(f"the other tuple: {other}.")    # Print it as well.

tuples: list[tuple[str, int, float]] = [  # Create a list of 4 tuples.
    mixed, ("pear", -2, 4.5), other, ("pear", -2, 3.3)]
print(f"tuples list: {tuples}.")       # Print that list.

tuples.sort()  # We sort the list of tuples.
print(f"sorted tuples list: {tuples}.")

a, b, c = mixed  # We unpack the tuple of length 3 into 3 variables.
print(f"{a = }, {b = }, {c = }")

mixed = "x", 4, 4.5  # Declare a tuple without parentheses.
print(f"mixed is now: {mixed}")
```

↓ `python3 tuples_2.py` ↓

Listing 5.18: The stdout of the program tuples_2.py given in Listing 5.17.

```
The mixed tuple is ('apple', 12, 1e+25).
the other tuple: ('pear', 1, 1.2).
tuples list: [('apple', 12, 1e+25), ('pear', -2, 4.5), ('pear', 1, 1.2), ('
    ↪ pear', -2, 3.3)].
sorted tuples list: [('apple', 12, 1e+25), ('pear', -2, 3.3), ('pear', -2,
    ↪ 4.5), ('pear', 1, 1.2)].
a = 'apple', b = 12, c = 1e+25
mixed is now: ('x', 4, 4.5)
```

square brackets, i.e., with `(...)`.

Third, on a semantic level, lists are intended to hold objects of the same type, while tuples can contain elements of different types. The type hint `list[int]` indicates that we want something to be a list of integer numbers. While the Python interpreter permits us to ignore this and still store arbitrary objects in that list, this would violate the idea behind lists. Tuples, on the other hand, are designed to contain elements of different types.[1] Since they cannot be changed, it will always be clear which element of which type is at which location.

Best Practice 25

When you need to use an indexable sequence of objects, use a `list` only if you intent to modify this sequence. If you do not intend to change the sequence, use a `tuple`.

Listing 5.15 shows us some of the things we can do with tuples. We create a tuple `fruits` to hold three strings. The proper type hint for this is `tuple[str, str, str]` [340]. The line `fruits: tuple[str, str, str] = ("apple", "pear", "orange")` thus creates a tuple `fruits` which contains three strings, namely `"apple"`, `"pear"`, and `"orange"` We also annotated the variable with a type hint informing any static code analysis tool that we indeed intend to store three strings in the tuple. Notice that the tuple is defined using parentheses, whereas a list with the same content would have been defined using square brackets, e.g., as `fruits: list[str] = ["apple", "pear", "orange"]`.

[1] This is only true for tuples that are defined using a type hint *without* ellipsis `...`, but more about this later.

Listing 5.19: A third example of using tuples in Python: testing the immutability property. (stored in file tuples_3.py; output in Listing 5.20)

```
"""An example of testing the immutability of tuples."""

# Create a tuple consisting of an immutable object (the integer `1`) and
# a mutable object (the list [2]).
mt: tuple[int, list[int]] = (1, [2])
print(f"{mt = }")  # This prints mt == (1, [2]).

mt[1].append(2)  # We can actually change the list inside the tuple.
print(f"{mt = }")  # This prints mt == (1, [2, 2]).

mt[1] = [3, 4]  # However, this will fail with an TypeError exception.
print(f"{mt = }")  # ...and we never reach this part.
```

↓ python3 tuples_3.py ↓

Listing 5.20: The stdout, stderr, and exit code of the program tuples_3.py given in Listing 5.19.

```
mt = (1, [2])
mt = (1, [2, 2])
Traceback (most recent call last):
  File "{...}/collections/tuples_3.py", line 11, in <module>
    mt[1] = [3, 4]  # However, this will fail with an TypeError exception.
    ~~~~^
TypeError: 'tuple' object does not support item assignment
# 'python3 tuples_3.py' failed with exit code 1.
```

Listing 5.21: The results of static type checking with Mypy of the program tuples_3.py given in Listing 5.19.

```
$ mypy tuples_3.py --no-strict-optional --check-untyped-defs
tuples_3.py:11: error: Unsupported target for indexed assignment ("tuple[
    ↪ int, list[int]]")  [index]
Found 1 error in 1 file (checked 1 source file)
# mypy 2.4.1 failed with exit code 1.
```

If we are not sure about the actual number of elements that we will put in a tuple but know that they are all of the same type, we can use the ellipsis `...` in the type hint. `tuple[str, ...]` denotes a tuple that can receive an arbitrary amount of strings. The line `veggies: tuple[str, ...] = ("onion", "potato", "leek", "garlic")` is therefore correct and defined a tuple `veggies` consisting of four strings. This makes sense especially if we convert a `list[str]` to a tuple. Notice that using the ellipsis `...` in the `tuple` type hint is *only* permitted for homogeneous tuples, i.e., tuples where all elements are of the same type [592, 593].

The elements of tuples can be accessed using the normal square brackets, exactly like list elements. `veggies[0]` gives us the first element of the tuple `veggies`, namely `"onion"`. `veggies[1]` gives us the second element of the tuple `veggies`, namely `"potato"`. Negative indices also work and index the tuple from the end. `veggies[-1]` gives us the last element of the tuple `veggies`, namely `"garlic"`. `veggies[-2]` gives us the second-to-last element of the tuple `veggies`, namely `"leek"`.

Tuples support many of the same non-modifying operators as lists. The `in` and `not in` operators both work. We quickly test only the former by asking `'pear' in fruits`, which evaluates to `True` and `'pear' in veggies`, which is `False`. The `index`, too, is available and `fruits.index('apple')` returns `0` because `"apple"` is the very first element of `fruits`.

With program tuples_2.py in Listing 5.17, we explore tuples containing elements of multiple different types. The type hint `tuple[str, int, float]` states that we want to define a tuple where the first element is a string, the second element is an integer number, and the third element is a floating point number. The line `mixed: tuple[str, int, float] = ("apple", 12, 1e25)` stores such a tuple in the variable `mixed`. The first element of the tuple is the string `"apple"`. The second element

Listing 5.22: The results of the `Ruff` linter applied to the program `tuples_3.py` given in Listing 5.19.

```
$ ruff check --target-version py312 --select=A,AIR,ANN,ASYNC,B,BLE,C,C4,COM
    ↪ ,D,DJ,DTZ,E,ERA,EXE,F,FA,FIX,FLY,FURB,G,I,ICN,INP,ISC,INT,LOG,N,NPY,
    ↪ PERF,PIE,PLC,PLE,PLR,PLW,PT,PYI,Q,RET,RSE,RUF,S,SIM,T,T10,TD,TID,TRY,
    ↪ UP,W,YTT --ignore=A005,ANN001,ANN002,ANN003,ANN204,ANN401,B008,B009,
    ↪ B010,C901,D203,D208,D212,D401,D407,D413,INP001,N801,PLC2801,PLR0904,
    ↪ PLR0911,PLR0912,PLR0913,PLR0914,PLR0915,PLR0916,PLR0917,PLR1702,
    ↪ PLR2004,PLR6301,PT011,PT012,PT013,PYI041,RUF100,S,T201,TRY003,UP035,W
    ↪  --line-length 79 tuples_3.py
All checks passed!
# ruff 0.17.3 succeeded with exit code 0.
```

is the integer `12` and the third element is the floating point number `1e25`, i.e., 10^{25}. With the line `other: tuple[str, int, float] = ("pear", 1, 1.2)` we create another such tuple.

We now want to create a list of such tuples. This list should, of course, also be annotated with the proper type hint. This is easy: The type hint for our kind of tuples is `tuple[str, int, float]`. The type hint for a list is `list[elementType]`, where `elementType` is the type of the elements. If we want to create a list containing tuples of our kind, then the proper type hint would be `list[tuple[str, int, float]]`. Having realized this, we can now create the list `tuples`. We use the square bracket syntax for this. As first element, we put the tuple stored in the variable `mixed`. Then we simply define another tuple `("pear", -2, 4.5)` inline. The third element is the tuple stored in variable `other`. As last element, we again define a tuple `("pear", -2, 3.3)`. Listing 5.18 shows that we can print this list of tuples just like any other list (using an f-string).

There is an order defined on tuples (and lists, too), meaning that they support all six comparison operators `<`, `<=`, `==`, `>=`, `>`, and `!=`. Two tuples `x` and `y` are compared elementwise lexicographically. If the first element of `x` is less than the first element of `y`, then `x < y`. If the first element of `x` is greater than the first element of `y`, then `x > y`. If the first elements of both tuples are equal, the comparison continues at the second element, if both tuples have at least length 2 (otherwise, the shorter tuple is considered as smaller), and so on. This means that we can sort our list of tuples by writing `tuples.sort()`. As a result, the tuple with the string `"apples"` remains at the first position. All other tuples have `"pear"` as first element and thus are "greater." Two of them have the integer `-2` as the second element and thus are located before the tuple with `1` as second element. Among these two tuples, the one with the smallest third element comes first.

Finally, this example program shows us that we can unpack tuples just like lists. `a, b, c = mixed` stores `"apple"` in `a`, `12` in `b`, and `1e25` in `c`. Interestingly, we can also create tuples the other way around. We can omit the parentheses when creating tuples in cases where no confusion is possible. `mixed = "x", 4, 4.5` has the same effect as `mixed = ("x", 4, 4.5)`.

Let us finally explore the immutability of tuples in Listing 5.17. As stated before, we are not permitted to add, remove, insert, delete, or overwrite any element of a tuple. What happens if we try create a tuple that contains a list? The line `mt: tuple[int, list[int]] = (1, [2])` does just this. It creates a new tuple `mt` whose first element is the integer number `1` and whose second element is a list of integers, namely `[2]`. We can access this list via `mt[1]`. While the tuple cannot be changed, this list can! `mt[1].append(2)` will append the number `2` to the list, which now is `[2, 2]`. Printing `f"mt == mt"` therefore yields `mt == (1, [2, 2])`.

Best Practice 26

Only put immutable objects into tuples if you want to use the tuples as immutable objects. Mutable objects inside tuples makes the tuples modifiable as well, while other programmers may assume that they are immutable. Violating this assumption can lead to strange errors down the line.

While we should usually put only immutable objects into tuples, there is one notable exception: Later we will learn how to program functions (in Chapter 8). Every function can return a single value as result, like, e.g., `sin` or `sqrt` do. Sometimes, if we want to return multiple values, we can return them

in a tuple. This has the advantage that the value types are clear. Of course, in that case, these values do not need to be immutable. (Because functions may return lists and other mutable objects...)

However, if we use tuples as collections of data, we often want to make sure that they can semantically be considered as constants. To achieve this, their values should ideally also be immutable.

What we are never permitted to do is to change the contents of the tuple itself. Trying to do `mt[1] = [3, 4]` will raise a `TypeError`. This is an exception which, unless properly caught and handled, will terminate the program immediately. Listing 5.20 shows us this. While the first two `print` commands still succeed, the program crashes when we lay our hands on the tuple directly. The `Python` interpreter is kind enough to print the error as well as where it happened (see also later in Chapter 9). This would help the programmer to find the problem.

Had the programmer used `Mypy` to check the code before running it, however, they would have found the error already. The output of `Mypy` given in Listing 5.21 shows us that a tuple is an *Unsupported target for indexed assignment*. And indeed it is. `Ruff`, however, at the time of this writing, does not spot the error (Listing 5.22). This is because it is a type-related error and `Ruff` does not actually check for those. It is always useful to check our code with multiple tools.In summary, tuples are like lists, just immutable and they are semantically conceptualized to contain objects of different types. Careful with immutability, though, as it can only be guaranteed if the tuple itself consists of only immutable objects.

5.4 Sets

Lists and tuples are classical sequence-based data structures. Another classical data structure is a *set*. Lists and tuples can contain an element arbitrarily often. Sets implement the mathematical notation of sets. Different from tuples and lists, they can contain each element at most once. To ensure this property, sets should only contain immutable objects. If you could change the object, then there would be no way for the set to ensure that no two of its objects can be the same.

Best Practice 27

Only put immutable objects into sets.

Different from tuples, sets themselves are not immutable. Furthermore, sets are *unordered* collections [501]. Therefore, also different from tuples and lists, they cannot be indexed, i.e., the square-bracket notation does not work with sets.

Best Practice 28

Sets are unordered. Never expect anything about how the objects you put into a set are actually stored there.

At this point, you may ask: *What do I need sets for?* Adding and removing objects and checking whether an object is contained in a collection ... lists can do all of that as well. Plus lists are ordered and indexable. So what is the advantage of sets?

Sets are fast. In `Python`, sets are implemented based on the concept of hash tables and thus inherit the computational performance of these data structures [134, 322, 513]. The operation for checking whether one object is contained in a set can be done in $\mathcal{O}(1)$ on average, whereas lists need $\mathcal{O}(n)$ on average, with n being the list length [9, 276, 413]. This means that checking whether an element is contained in a set costs approximately a constant number of CPU cycles, whereas doing the same for a list takes a number of cycles roughly proportional to the length of the list. Lists thus can be preferred if we either only have very few elements to check or if we need to access elements using integer indices. If we have more than just very few elements and either need to perform efficient lookups or set operations such as those discussed in the following text, sets are the way to go.

In program `sets_1.py` given as Listing 5.23, we provide a first example of creating and working with sets. Set literals can be created by using curly braces, i.e, `{...}`. They can be type-hinted using the notation `set[elementType]` where `elementType` is the type of elements to be stored in the set [340]. The line `upper: set[str] = {"A", "G", "B", "T", "V"}` creates the variable `upper`, which points to

Listing 5.23: A first example of using sets in Python: creating, modifying, and converting sets. Since sets are unordered, printing them can yield a different result each time a program is executed (see Best Practice 28). (stored in file sets_1.py; output in Listing 5.24)

```
"""An example of creating, modifying, and converting sets."""

upper: set[str] = {"A", "G", "B", "T", "V"}  # Some uppercase letters...
print(f"some uppercase letters are: {upper}")  # Print the set.

upper.add("Z")  # Add the letter "Z" to the set.
upper.add("A")  # The letter "A" is already in the set.
upper.add("Z")  # The letter "Z" is already in the set.
print(f"some more uppercase letters are: {upper}")  # Print the set.

upper.update(["K", "G", "W", "Q", "W"])  # Try to add 5 letters.
print(f"even more uppercase letters are: {upper}")  # Print the set.

lower_tuple: tuple[str, ...] = ("b", "i", "j", "c", "t", "i")
lower: set[str] = set(lower_tuple)  # Convert a tuple to a set.
print(f"some lowercase letters are: {lower}")  # Print the set 'lower'.
lower.remove("b")  # Delete letter b from the set of lower case letters.
print(f"lowercase letters after deleting 'b': {lower}")  # Print the set

letters: set[str] = set(lower)  # Copy the set of lowercase characters.
letters.update(upper)  # Add all uppercase characters.
print(f"some letters are: {letters}")  # Print the set 'letters'.

# Create a sorted list containing all elements of the set letters.
# Warning: Strings are sorted such that uppercase characters come before
# lowercase characters.
letters_list: list[str] = sorted(letters)
print(f"the sorted list of letters is: {letters_list}")
```

↓ `python3 sets_1.py` ↓

Listing 5.24: The stdout of the program sets_1.py given in Listing 5.23.

```
some uppercase letters are: {'A', 'G', 'B', 'V', 'T'}
some more uppercase letters are: {'Z', 'A', 'G', 'B', 'V', 'T'}
even more uppercase letters are: {'A', 'W', 'V', 'Z', 'G', 'B', 'T', 'K', '
    ↪ Q'}
some lowercase letters are: {'b', 'i', 'j', 'c', 't'}
lowercase letters after deleting 'b': {'i', 'j', 'c', 't'}
some letters are: {'K', 'A', 'i', 'W', 'T', 'V', 'j', 'Z', 'Q', 'G', 'B', '
    ↪ c', 't'}
the sorted list of letters is: ['A', 'B', 'G', 'K', 'Q', 'T', 'V', 'W', 'Z
    ↪ ', 'c', 'i', 'j', 't']
```

a set of the five uppercase latin characters `"A"`, `"G"`, `"B"`, `"T"`, and `"V"`. The type hint `set[str]` states that this is a set that shall contain only strings.[2]

With the method `add`, we can add new elements to a set. Invoking `upper.add("Z")` will add the string `"Z"` to the set `upper`. The following call to `upper.add("A")` does nothing, because the string `"A"` is already contained in the set. Similarly, invoking `upper.add("Z")` again also does nothing, since `"Z"` is now already a set member as well.

We can also add a batch of elements to the set using the `update` method. `upper.update(["K", "G", "W", "Q", "W"])` attempts to add all five elements in the list `["K", "G", "W", "Q", "W"]` to the set `upper`. However, `"W"` appears twice and hence, is only added once. `"G"` is already a member of the set and therefore not added.

We can also create a set by passing any *sequence* (e.g., a list or tuple) into the constructor `set`. To

[2] The type hint `set[int|str]` would denote a set that can contain both strings and integers.

Listing 5.25: A second example of using sets in Python: creating sets and set operations (as illustrated in Figure 5.2). Since sets are unordered, printing them can yield a different result each time a program is executed (see Best Practice 28). (stored in file sets_2.py; output in Listing 5.26)

```
"""An example of creating sets and set operations."""

odd: set[int] = {1, 3, 5, 7, 9, 11, 13, 15}  # a subset of odd numbers
print(f"some odd numbers are: {odd}")  # Print the set.
print(f"is 3 \u2208 odd: {3 in odd}")  # Check if 3 is in the set odd.
print(f"is 2 \u2209 odd: {2 not in odd}")  # Check if 2 is NOT in odd.

prime: set[int] = {2, 3, 5, 7, 11, 13}  # a subset of the prime numbers
print(f"some prime numbers are: {prime}")  # Print the set.

set_or: set[int] = odd.union(prime)  # Create a new set as union of both
print(f"{len(set_or)} numbers are in odd \u222A prime: {set_or},")

set_and: set[int] = odd.intersection(prime)  # Compute the intersection.
print(f"{len(set_and)} are in odd \u2229 prime: {set_and},")

only_prime: set[int] = prime.difference(odd)  # Prime but not odd
print(f"{len(only_prime)} are in prime \u2216 odd: {only_prime},")

# Get the numbers that are in one and only one of the two sets.
set_xor: set[int] = odd.symmetric_difference(prime)
print(f"{len(set_xor)} are in (odd \u222A prime) "
      f"\u2216 (odd \u2229 prime): {set_xor}, and")

only_odd: set[int] = odd.difference(prime)  # Odd but not prime
print(f"{len(only_odd)} are in odd \u2216 prime: {only_odd},")
odd.difference_update(prime)  # delete all prime numbers from odd
print(f"after deleting all primes from odd, we get {odd}")
```

↓ python3 sets_2.py ↓

Listing 5.26: The stdout of the program sets_2.py given in Listing 5.25.

```
some odd numbers are: {1, 3, 5, 7, 9, 11, 13, 15}
is 3 ∈ odd: True
is 2 ∉ odd: True
some prime numbers are: {2, 3, 5, 7, 11, 13}
9 numbers are in odd ∪ prime: {1, 2, 3, 5, 7, 9, 11, 13, 15},
5 are in odd ∩ prime: {3, 5, 7, 11, 13},
1 are in prime \ odd: {2},
4 are in (odd ∪ prime) \ (odd ∩ prime): {1, 2, 9, 15}, and
3 are in odd \ prime: {1, 9, 15},
after deleting all primes from odd, we get {1, 9, 15}
```

demonstrate this, we first create the tuple `lower_tuple` to hold the lowercase characters `("b", "i", "j", "c", "t", "i")`. The command `lower: set[str] = set(lower_tuple)` creates the set `lower`, type hints it as set of strings, and lets it contain all the elements of `lower_tuple`. `lower_tuple` contains the letter `"i"` twice, but it will appear only once in the set that we have created. Notice: Just calling `set()` without argument (or with an empty collection inside) creates an empty set. We can delete an element of a set using the `remove` function: `lower.remove("b")` deletes the element `"b"` from the set `lower`.

We now copy the set `upper` by invoking `set(upper)` and obtain the new set `letters`. We add all the elements of set `lower` to the set `letters` by calling `letters.update(lower)`.

We can also convert the set `letters` to a list or tuple. `list(letters)` will achieve the former, `tuple(letters)` the latter. However, since sets are unordered [501, 503] and the order in which the elements would appear in the created list or tuple is not defined. The elements of the set thus could appear in a strange or unexpected order in the new collection. Here we therefore use the function `sorted`, which accepts an arbitrary collection of objects as input and returns a sorted list.

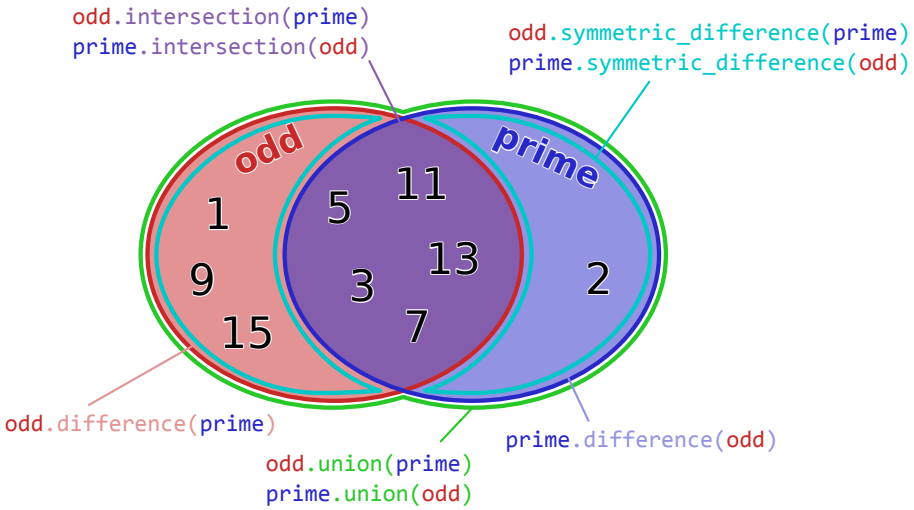


Figure 5.2: An illustration of the set operations in Python as used in Listing 5.25.

`sorted(letters)` therefore creates a list where all the elements of letters appear in a sorted fashion. Notice that the natural order of letters is alphabetically, but uppercase letters come before lowercase letters (which sometimes may be confusing as well). Either way, we obtain `letters_list`, which contains all the strings in `letters` in this default sorting order.

Best Practice 29

Careful when sorting or comparing strings: The default order is *uppercase characters before lowercase characters*, i.e., `"A" < "a"`. If you want that upper- and lowercase characters are treated the same (e.g., that `"A"` is considered as equal to `"a"`), as is the case in dictionary ordering, i.e., if you want to sort a collection `my_text` of strings in a case-insensitive manner, use `sorted(my_text, key=str.casefold)`.

In program `sets_2.py` provided as Listing 5.25, we illustrate several basic operations with sets, some of which are visualized in Figure 5.2. This time, we begin by defining a set `odd` containing the first eight odd integers. Sets, like lists and tuples, support both the `in` and `not in` operators. In the context of sets, they are equivalent to the $\in$ and $\notin$ operations from, well, set theory. We illustrate this in our small example using the unicode escape sequences `\u2208` and `\u2209` for both characters (see back in Section 3.6.6). Both `3 in odd` and `2 not in odd` are `True`.

We then define a set `prime` containing the six smallest prime numbers. We can compute the joint set of `prime` and `odd`, i.e., `prime` $\cup$ `odd` by using the `union` method. Both `prime.union(odd)` and `odd.union(prime)` will create a new set containing all the elements from both sets, namely the first eight odd integers and the number `2`. Similarly, the `intersection` method computes the intersection of two sets. Both `prime.intersection(odd)` and `odd.intersection(prime)` compute `odd` $\cap$ `prime`, which consists of the five numbers `3, 5, 7, 11, 13`. If you want to get the set of numbers that are either in `odd` or in `prime` but not in both sets, the method `symmetric_difference` does the job. Both `prime.symmetric_difference(odd)` and `odd.symmetric_difference(prime)` return `{1, 2, 9, 15}`. The elements of `prime` that are *not* in `odd` can be computed using `prime.difference(odd)`. The result, `prime` $\setminus$ `odd`, contains a single element, namely the number `2`. Similarly, we can create the new set `only_odd` containing all the elements of `odd` that are *not* in `prime` by using `odd.difference(prime)`. The result, `odd` $\setminus$ `prime`, contains a the three elements `1`, `9`, and `15`.

Notice that all of these operations create new and independent sets. If we want to delete all elements that occur in some container from an existing set, i.e., modify the set, then we use the method `difference_update`. `odd.difference_update(prime)` modifies the set `odd` by deleting all elements from `prime` from it. This operation did not create a new set, but change the existing one. `odd == only_odd` would now hold. It is similar to the `update` method, but deletes elements instead of adding them.

We will discuss a bit more about the mechanism that allows us to store and retrieve objects from sets in Section 13.2.

Listing 5.27: An example of using dictionaries in Python. (stored in file dicts_1.py; output in Listing 5.28)

```
"""An example of dictionaries."""

num_str: dict[int, str] = {  # create and type hint a dictionary
    2: "two", 1: "one", 3: "three", 4: "four"}  # the elements
print(f"num_str has {len(num_str)} elements: {num_str}")  # print dict
print(f"I got {num_str[2]} shoes and {num_str[1]} hat.")  # get element

print(f"the keys are: {list(num_str.keys())}")  # print the keys
print(f"the values are: {list(num_str.values())}")  # print the values
print(f"the items are: {list(num_str.items())}")  # the key-value pairs

num_str[5] = "fivv"  # insert (or update) the value of a key
print(f"after adding key 1 num_str is now {num_str}")
num_str[5] = "five"  # update the value of a key
print(f"after updating key 1 num_str is now {num_str}")

del num_str[4]  # delete a key
print(f"after deleting key 4 num_str is now {num_str}")
# get the value of a key, then delete it
print(f"popping key 5 gets us {num_str.pop(5)}")

str_num: dict[str, int] = {}  # create empty dictionary
str_num.update({"one": 1, "three": 3, "two": 2, "four": 4})
print(f"{num_str[1]} + {num_str[2]} = {str_num['three']}")
```

↓ `python3 dicts_1.py` ↓

Listing 5.28: The stdout of the program dicts_1.py given in Listing 5.27.

```
num_str has 4 elements: {2: 'two', 1: 'one', 3: 'three', 4: 'four'}
I got two shoes and one hat.
the keys are: [2, 1, 3, 4]
the values are: ['two', 'one', 'three', 'four']
the items are: [(2, 'two'), (1, 'one'), (3, 'three'), (4, 'four')]
after adding key 1 num_str is now {2: 'two', 1: 'one', 3: 'three', 4: 'four
    ↪ ', 5: 'fivv'}
after updating key 1 num_str is now {2: 'two', 1: 'one', 3: 'three', 4: '
    ↪ four', 5: 'five'}
after deleting key 4 num_str is now {2: 'two', 1: 'one', 3: 'three', 5: '
    ↪ five'}
popping key 5 gets us five
one + two = 3
```

5.5 Dictionaries

Dictionaries in Python are collections that store key-value pairs. The value associated with a given key is accessed via the `[...]`-based indexing. The keys are unique and can be of an arbitrary type, as long as they are immutable. The concept of dictionary is also known as hash table or hash map in other languages or domains. Sets and dictionaries in Python are implemented using similar data structures [9], namely hash tables [134, 322, 513], and thus exhibit similar performance.

Listing 5.27 shows program dicts_1.py with some examples of how we can use dictionaries. A dictionary can be created again using the `{...}` syntax, however, this time, we place comma-separated `key-value` pairs within the curly braces. The type hints for dictionaries is `dict[keyType][valueType]`, where `keyType` is the type for the keys and `valueType` is the type for the values [340]. The line `num_str: dict[int, str] = {2: "two", 1: "one", 3: "three", 4: "four"}` therefore creates a new dictionary and stores it in the variable `num_str`. The type-hint of the variable states that it can point only to dictionaries that have integers as keys and strings as values. The contents of the dictionary are the key-value pairs `2: "two"`, `1: "one"`, `3: "three"`, and `4: "four"`. This means that the value

string `"one"` is stored under the key `1`, the value string `"two"` is stored under the key `2`, and so on.

We can use dictionaries in f-strings like any other Python object. The function `len` can give us the number of elements in a dictionary. For our dictionary `num_str` with four key-value pairs, it returns `4`. The elements of the dictionary can be accessed by their keys using square brackets. Therefore `num_str[2]` will return the string `"two"` associated with key `2` and `num_str[1]` returns the string `"one"`.

We can access the keys, values, and key-value pairs of a dictionary as collections using the `keys`, `values`, and `items` functions, respectively. In the example Listing 5.27, we convert them to lists: `list(num_str.keys())` gives us all the keys in the dictionary `num_str` as a `list`, which thus equals `[2, 1, 3, 4]`. Similarly, `list(num_str.values())` gives us all the values in the dictionary `num_str` as a `list`, which thus equals `["two", "one", "three", "four"]`. Finally, `num_str.items()` returns the key-value pairs a sequence of tuples and wrapping this into a list yields `[(2, "two"), (1, "one"), (3, "three"), (4, "four")]`. You may notice that all of these sequences have the same order of elements that was used when we created the dictionary. They key-value pair `2: "two"` comes before `1: "one"`. This is because dictionaries, different from sets, are ordered data structures. Their elements appear in all sequenced versions always in the same order in which they were originally inserted into the dictionary [156].

Dictionaries can be modified. We can add an entry simply by assigning a value to a key. `num_str[5] = "fivv"` assigns the value `"fivv"` to the key `5`. This association is now part of the dictionary. Oops, we noticed a typo: We wanted to add `"five"`, not `"fivv"`. `num_str[5] = "five"` will overwrite this faulty assignment. All keys in a dictionary are unique, so there can only be one value associated with key `5`. This also means that dictionary keys must obey the same requirement as set elements:

Best Practice 30

Dictionary keys must be immutable (and hashable).

We can also delete key-value pairs. This is done by invoking `del dict[key]`, which deletes the key `key` and its associated value from the dictionary. The method `pop` allows us to get the value of a given key and then immediately deletes the key-value pair from the dictionary. `num_str.pop(5)` will return `"five"` and remove the key-value pair `5: "five"` from the dictionary. Afterwards, requesting `num_str[5]` would lead to an which would terminate the program (unless properly caught).

As stated before, the types for keys and values in a dictionary can be chosen arbitrary, with the limitation that keys must be immutable. We now create the empty dictionary `str_num: dict[str, int] = {}`.[3] Notice that only due to the type hint `dict[str, int]`, other programmers and static type-checking tools can know at this point that we intend to use strings as keys and integers as values in this new dictionary. The `update` method allows us to append the values of an existing dictionary to a given dictionary. `str_num.update({"one": 1, "three": 3, "two": 2, "four": 4})` therefore stores four key-value pairs into our new dictionary. This time, the keys are indeed strings and the values are integers. The slightly convoluted f-string `f"{num_str[1]}+ {num_str[2]}= {str_num['three']}"` evaluates to `"one + two = 3"`. The first two values are looked up in the `num_str` dictionary under keys `1` and `2`, respectively. The last value is obtained from `str_num` under key `"three"`.

We will discuss a bit more about the mechanism that allows us to store and retrieve objects from dictionaries in Section 13.2.

5.6 Summary

If you have managed to fight your way through the book until this point, then you can already do quite a few things. You can use the computer like a fancy calculator by evaluating numerical expressions, which you learned in Chapter 3. By utilizing variables as discussed in Chapter 4, you can realize some simple algorithms like approximating π. In this section, you learned about compound data structures that can store multiple values. You learned about mutable and immutable sequences of values, namely

[3] Using the `dict()` without arguments has the same effect, but several linters discourage this and encourage using `{}` instead.

Table 5.1: Some properties of the four basic collection datatypes of Python, namely `list`, `tuple`, `set`, and `dict`.

type hint	`list[A]`	`tuple[B, C, ...]`	`set[D]`	`dict[E, F]`
literal	`[a1, a2, ...]`	`(b, c, ...)`	`{d1, d2, ...}`	`{e1: f1, e2: f2, ...}`
empty Literal	`[]`	`()`	✗ / `set()`	`{}`
copy other collection `x`	`list(x)`	`tuple(x)`	`set(x)`	`dict(x)` (`x` is `dict`)
ordered	✓	✓	✗	✓ (insertion sequence)
collection is mutable	✓	✗	✓	✓
mutable elements OK	✓	✓/ ✗	✗	`E`: ✗; `F`: ✓
all elements of same datatype	✓	✗	✓	✓
indexing via `[i]`	✓ (`i` is `int`)	✓ (`i` is `int`)	✗	✓ (`i` is `E`)
element repetition	✓	✓	✗	`E`: ✗; `F`: ✓
add element in	$\mathcal{O}(1)$	✗	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`in`/`not in`	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
delete element in	$\mathcal{O}(n)$	✗	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Overhead	low	low	some	some

lists and tuples. You also learned that Python offers you the mathematical notion of sets as well as dictionaries, which are key-value mappings. You now have the basic knowledge of the most commonly used datatypes in Python. You know operations to manipulate and query them.

Table 5.1 summarizes some of the features of these collection datatypes. All four basic collection datatypes can be type hinted. This is important to ensure that other programmers can clearly understand the meaning of our code and that tools like Mypy can better recognize dodgy statements and potential errors. Here, the collection type comes first followed by the element data type(s) in square brackets.

We can create the collections using literals, which allow us to directly specify their contents. Here, lists are defined using square brackets, tuples in parentheses, and dictionaries as well as sets with curly braces. If we leave the contents between the delimiters empty, this will allocate an empty container – with the notable exception of sets. Empty sets must be created via `set()` (otherwise, an empty set would be indistinguishable from an empty dictionary).

We can copy collections into lists, sets, tuples, or dictionaries by simply using the datatype name as function name to which we pass the collection. This way, we can create sets from lists or lists from tuples. For dictionaries, the collection passed to `dict` should also be a dictionary.

The elements in lists, tuples, and dictionaries can be accessed in an order, defined by collection (literal) definition and/or insertion and deletion sequence. With the exception of tuples, all collections are mutable. The keys of dictionaries and the elements of sets must be immutable. Their location in the storage backing these collection depends on their value. If we would store mutable objects and later change their value, the collection would not know that and then would be unable to find these objects, because now their value would no longer fit to their storage position. If we use tuples as constants, then obviously their elements should also be immutable. If we use tuples to package multiple return values of functions together, which we will learn much later, then these do not need to be immutable.

Sets can contain each element at most once and each dictionary key can occur only once in a dictionary. The other collections can contain each element arbitrarily often (and dictionary values can occur arbitrarily often as well). Adding elements can be done quickly by all immutable collections, but finding and deleting elements is fast only for sets and dictionaries. They pay for this with some additional overhead.

Now, when learning about these structures, you may have felt some eerie awkwardness. Something did not feel right. If I can only construct a list step by step, then what is the advantage compared to just using many variables? The missing ingredient are control flow statements. Statements that allow us to iterate over a list. Statements that allow us to perform an action A if an element is contained in a set, for example, and an action B otherwise. The next big part of this book will close this gap and provide you with all the tools necessary to write "real" programs. Later, in Section 7.5 and Chapter 10, we will circle back to the topic of collections and look into how sequences of elements can be processed iteratively.

Part II

Control Flow Statements

With the things we learned so far, we can write simple linear programs. These programs perform instruction after instruction in exactly the same order in which we write them down. If we want something done twice, we have to write it down twice. Our programs have no way to adapt their behavior based on their input data. They cannot do one thing if a variable has a certain value and do something else otherwise. They do not branch or loop. Now we will learn how to write programs that *do* branch or loop. We will learn statements that change the *control flow*.

Definition 5.1: Control Flow

The *control flow* is the order in which the statements in a program are executed.

Chapter 6

Conditional Statements

Conditional statements allow us to execute a piece of code *only* if a certain Boolean expression evaluates to `True`. This allows us to let the program make choices, to do one thing in one situation and another thing in a different situation. Conditionals are therefore the most fundamental control flow statement.

6.1 The `if` Statement

A conditional statement is basically a piece of program code which will execute a set of instructions if a condition is met. The condition is provided as the kind of Boolean expression that we already discussed back in Section 3.5. The syntax for this in Python is very simple[404]:

```
"""The syntax of an "if"-statement in Python."""

if booleanExpression:
    conditional statement 1  # Only executed if booleanExpression
    conditional statement 2  # evaluates to True.
    # ...

normal statement 1  # Always executed, regardless of the result of
normal statement 2  # booleanExpression.
# ...
```

The first line begins with `if`, which is followed by a Boolean expression, followed by a colon (`:`). If – and only if – the Boolean expression evaluates to `True`, then the *indented* block of statements below the `if` are executed. Notice that each of these conditionally executed statements is indented by four spaces compared to the `if`. After the body block of the `if` statement, the normal program code resumes without additional indentation. The statements that are not indented anymore will be executed regardless of the outcome of the Boolean expression.

Best Practice 31

Blocks are indented by four spaces [594].

Let's use this construct to check whether a year is a leap year. According to the Gregorian calendar, a year y is a leap year if y is divisible by 4 but not by 100; or if it is divisible by 400 [502]. Program `if_example.py` in Listing 6.1 implements this rule. We begin by storing the year that we want to investigate in the integer variable `year`. But how can we check the condition?

From Section 3.2 you may remember the modulo division operator `%`: Applying it to two values `a` and `b`, it will return the remainder of the division `a // b`. If the remainder is 0, then `a` can be divided by `b` evenly. For example, `10 % 5 == 0` whereas `10 % 6 == 4` and `10 % 3 == 1`. So to check whether `year` can be divided by 4, we just need to check whether `(year %4) == 0`. Furthermore, `year` should not be divisible by 100, so we also check `(year %100) != 0`, i.e., whether the remainder of the division of `year` by 100 is *not* 0. We now have to combine both conditions with an `and`. We write `((year %4) == 0) and ((year %100) != 0)`. This way, the result is only `True` if both conditions

Listing 6.1: An example of using the `if` statement. (stored in file `if_example.py`; output in Listing 6.2)

```
"""
An example of using the `if` statement.

A year is a leap year if it is divisible by 4 but not divisible by 100
or if it is divisible by 400.
We can use the modulo operator `%` to check this.
`a % 100` will be `0` if and only `a` is a multiple of `100`.
The Boolean statements can be combined with `and` and `or` and grouped
using parentheses.
"""

year: int = 2024  # The year 2024 should be a leap year.
if (((year % 4) == 0) and ((year % 100) != 0)) or ((year % 400) == 0):
    print(f"{year} is a leap year.")  # This line will be executed.

year = 1900       # The year 1900 is not a leap year.
if (((year % 4) == 0) and ((year % 100) != 0)) or ((year % 400) == 0):
    print(f"{year} is a leap year.")  # This line is never reached.

print("End of year checking.")  # This text is always printed.
```

↓ `python3 if_example.py` ↓

Listing 6.2: The `stdout` of the program `if_example.py` given in Listing 6.1.

```
2024 is a leap year.
End of year checking.
```

are `True`, i.e., if the year is divisible by 4 but not by 100. We use parentheses so that it is easy to see how the Boolean expression is structured.

Well, if that combined condition turns out to be `False`, the year could still be a leap year if it was divisible by 400. So the final condition to be checked is `(year %400) == 0`. We add this condition with the `or` operator, because it is sufficient that either our original expression *or* the divisible-by-400-check are `True` for the overall condition to be met. We obtain the expression `(((year %4) == 0) and ((year %100) != 0)) or ((year %400) == 0)`. Again, we use parentheses to neatly group the sub-expressions for readability.

All that is left to do is put an `if` in front of the condition and a `:` after it, and we have the conditional check.[1] If it evaluates to `True`, we will print `f"{year}is a leap year."`. For this purpose, we need to indent the `print` command with four spaces.

We copy this code twice and check the years `2024` (the year when I began writing this book) and `1900` (some other year). Notice that the second `if` starts at the same level of indentation of the first `if`. This means that it is executed regardless of the outcome of the first conditional. After all of this code, we print the text `End of year checking.` to the console. This last line of code is again not indented and therefore will be executed regardless of the two conditional statements before it. The output of our program given in Listing 6.2 confirms that 2024 was indeed a leap year, while 1900 was not.

6.2 The `if...else` Statement

The ability to perform some action if a given expression evaluates to `True` is already nice. However, often we want to perform some action if the expression evaluates to `True` and another action otherwise. Well, we could already do this. We could write the same `if` statment twice, but in the second one we would just wrap `not (...)` around the condition. This seems to be quite verbose and verbosity

[1] Unlike in programming languages like Java or C, parentheses are not needed around the conditions of `if` statements in Python.

leads to confusion. So we are offered a better alternative: the `if ... else` statement, which looks like this[404]:

```
"""The syntax of an "if-else"-statement in Python."""

if booleanExpression:
    conditional statement 1  # Only executed if booleanExpression
    conditional statement 2  # evaluates to True.
    # ...
else:
    alternative statement 1  # Only executed if booleanExpression
    alternative statement 2  # evaluates to False.
    # ...

normal statement 1  # Always executed, regardless of the result of
normal statement 2  # booleanExpression.
# ...
```

The statement begins like a normal `if` statement: `if`, followed by a Boolean expression, followed by a colon (`:`) marks the condition. The following block – where each statement is indented by four spaces – is executed only if the Boolean expression evaluates to `True`. Then, the `else:` line follows. Notice that it is not indented. It is at the same indentation level as the original `if`. All the statements in the following block are again indented by four spaces. They are only executed if the Boolean expression used in the `if` line evaluated to `False`. After this code, un-indented instructions can follow, which would then be executed regardless of which branch of the alternative was chosen.

This now allows us to, for example, print one text if a year is a leap year and print another text otherwise. In Listing 6.3 we show exactly this functionality with program `if_else_example.py`. We modify our leap year check program to also print text if the year is not a leap year. For this purpose,

Listing 6.3: An example of using the `if ... else` statement. (stored in file `if_else_example.py`; output in Listing 6.4)

```
"""An example of using the `if-else` statement."""

year: int = 2024  # the year 2024 should be a leap year
if (((year % 4) == 0) and ((year % 100) != 0)) or ((year % 400) == 0):
    print(f"{year} is a leap year.")  # This line will be executed.
    print("yes, it really is.")       # This line as well.
else:  # If we get here, it's not a leap year.
    print(f"{year} is not a leap year.")  # This line is never reached.
    print("Believe you me, it indeed is not.")    # also never reached.

year = 1900  # the year 1900 is not a leap year
if (((year % 4) == 0) and ((year % 100) != 0)) or ((year % 400) == 0):
    print(f"{year} is a leap year.")      # This line is never reached.
    print("yes, it really is.")           # This line is never reached.
else:  # If we get here, it's not a leap year.
    print(f"{year} is not a leap year.")  # This line will be executed.
    print("Believe you me, it indeed is not.")  # This line too.

print("End of year checking.")  # This text is always printed.
```

↓ `python3 if_else_example.py` ↓

Listing 6.4: The stdout of the program `if_else_example.py` given in Listing 6.3.

```
2024 is a leap year.
yes, it really is.
1900 is not a leap year.
Believe you me, it indeed is not.
End of year checking.
```

Listing 6.5: An example of using nested `if ... else` statements. (stored in file `if_else_nested.py`; output in Listing 6.6)

```
"""An example of using the nested `if-else` statements."""

a: int = 13   # the first number
b: int = 7    # the second number
c: int = 9    # the third number

if a > b:       # This means that a > b
    if a > c:   # This means that a > b and a > c.
        print(f"{a} is the greatest number.")
    else:       # This means that a > b and c >= a.
        print(f"{c} is the greatest number.")
else:           # This means that b >= a.
    if b > c:   # This means that b >= a and b > c.
        print(f"{b} is the greatest number.")
    else:       # This means that b >= a and c >= b.
        print(f"{c} is the greatest number.")

# Some side information: The max and min function are very useful.
print(f"The maximum is {max(a, b, c)}.")   # max: get the largest element
print(f"The minimum is {min(a, b, c)}.")   # min: get te smallest element
```

↓ `python3 if_else_nested.py` ↓

Listing 6.6: The stdout of the program `if_else_nested.py` given in Listing 6.5.

```
13 is the greatest number.
The maximum is 13.
The minimum is 7.
```

we use the new `else`-branch. Additionally, we also show that there can be multiple statements both in the `if` and the `else` block.

We can place multiple statements in the branches of both the `if` and `if...else` statements. Actually, the `if` and `if...else` are ... also *statements*. Does this mean that we can nest them? In other words, can we place an `if` statement inside the body of another `if` statement?

Yes, we can, and we explore this in our program `if_else_nested.py` given in Listing 6.5. Here, we have three numbers stored in variables `a`, `b`, and `c`. We want to know the maximum, i.e., the largest number stored in either `a`, `b`, or `c`. For this purpose, we first check if `a > b`. If this is `True`, then `b` cannot contain the largest number and it must be stored in either `a` or `c`.

Therefore, inside the body of that first `if` statement, we only have to check if `a > c`. Notice how this second `if` and its whole associated `else` and body are indented by another four spaces compared to the outer `if`. If `a > c` turns out to be `True`, we print the f-string `f"{a} is the greatest number."`. If this is `False`, then it could either be that `a < c` or that `a == c`. Either way, printing the f-string `f"{c} is the greatest number."` will yield the correct result.

Now we need to return to the outer `if` and tackle its alternative branch, the `else` part. What happens if `a > b` did not evaluate to `True`? Well, in this case, either `b > a` or `b == a`. It thus is sufficient to compare `b` with `c` to get the result. Therefore, we again indent a new `if`, this time with the condition `b > c`. If that one is `True`, then the largest value is equal to the one stored in `b`. We can print `f"{b} is the greatest number."`.

If that was `False`, we come to the `else` branch. Here, we know that `b >= a` and that either `c > b` or `c >= b`. Therefore, we again print the f-string `f"{c} is the greatest number."`.

Just for fun, I included two more lines in Listing 6.5 that show two useful functions we did not yet learn before: the built-in functions `max` and `min`.[2] Both receive a sequence of values and return the largest and smallest value, respectively. Therefore, the same work as with our nested `if` can be achieved by computing `max(a, b, c)`. The minimum is obtained using `min(a, b, c)`.

[2] Like `print`, these functions are always available and do not need to be imported.

6.3 The `if...elif...else` Statement

In some cases, we need to query a sequence of alternatives in such a way that the first `else` block would contain the next `if` and only that `if`. Then its `else` block would contain another `if` and only that `if`. Then its `else` block would contain another `if` and only that `if`, and so on. Every time we nest a conditional statement into another one we have to add four spaces of indentation. In the above situation, this would quickly fill the horizontal of our screens and look rather ugly. Therefore, the `elif` statement has been developed, which can replace an `else` containing just another `if`. The syntax of a combined `if ... elif` is quite easy[404]: Instead of the `else` containing the indented `if`, we simply write `elif` followed by the conditional expression that we would have used with the `if`.

```
"""The syntax of an "if-elif-else"-statement in Python."""

if booleanExpression 1:
    conditional 1 statement 1  # Only executed if booleanExpression 1
    conditional 1 statement 2  # evaluates to True.
    # ...
elif booleanExpression 2:  # such block can be placed arbitrarily often
    conditional 2 statement 1  # Only executed if booleanExpression 1
    conditional 2 statement 2  # evaluates to False and
    # ...                      # booleanExpression 2 evaluates to True.
else:  # The else-part is optional, it can be omitted.
    alternative statement 1  # Only executed if booleanExpression 1 and
    alternative statement 2  # 2 both evaluate to False.
    # ...

normal statement 1  # Always executed, regardless of the result of
normal statement 2  # booleanExpression 1 and 2.
# ...
```

Notice that we can use arbitrarily many `elif`s and, optionally, one `else`. Only if the condition of the `if` evaluates to `False`, the condition of the first `elif` is checked. Only if both the conditions of the `if` and the first `elif` returned `False`, the condition of the second `elif` is checked. Only if both the conditions of the `if` and the first and second `elif` returned `False`, the condition of the third `elif` is checked, and so on. Only if all the conditions of the `if` and all `elif`s were `False`, the body of the `else` is executed (if any).

We use this syntax to classify a person's current phase of life by their age in Listing 6.7. First, we define the `age` as an integer variable and pick a value, say 42. We want to fill a string in the variable `phase` that describes the current phase of life of that person.

Initially, nothing is filled in, so we set `phase = None`. We used the type hint `str | None` to specify that the variable can either be a string or `None` [453].

Anyway, we start with the first conditional and write `if age <= 3`. If this evaluates to `True`, then we set `phase = "infancy"`. This means that the person is a child in the earliest phase of life. In this case, the complete nested `if...elif...else` block ends and no other conditions are checked.

If this is not the case, i.e., if `age <= 3` evaluates to `False`, i.e., if `age > 3`, the next condition is checked. Instead of writing `else` followed by another (indented) `if`, we can just write `elif age <= 6` (without additional indentation). The `elif` basically functions as the `else` followed by an (indented) `if`. In our case, this next one checks whether the person's age is less than or equal to 6 (after it is already clear that `age <= 3` is `False`). If `age <= 6` holds, we call the life phase `"early childhood"`. If this is `True`, the, again, the complete nested `if...elif...else` block ends and no other conditions are checked.

If this is not the case, i.e., if `age <= 6` evaluates to `False`, we move on to the next `elif`. Only then, `elif age <= 8` is executed and if its condition evaluates to `True`, we refer to the life phase as `"middle childhood"` and are done with the `if...elif...else`. If `age <= 8` does not hold, we move to the next `elif`, and so on.

If even `age < 80`, the condition of the last `elif`, is `False`, the `else` block is executed. It will set `phase = "late adulthood"`. Finally, our program `prints` the life phase using an f-string.

Listing 6.7: An example of using the `if ... elif` statement. (stored in file `if_elif_example.py`; output in Listing 6.8)

```
"""An example of `elif` using human age groups."""

age: int = 42              # the age of the person
phase: str | None = None   # the life phase: to be computed

if age <= 3:     # If the age is no more than 3 years...
    phase = "infancy"  # then the person is in their infancy.
elif age <= 6:  # If (NOT age <= 3) and (age <= 6) ...
    phase = "early childhood"  # then they are in their early childhood.
elif age <= 8:  # If (NOT age <= 3) and (NOT age <= 6) and (age <= 8)
    phase = "middle childhood"
elif age <= 11:  # If ... (NOT age <= 8) and (age <= 11)
    phase = "late childhood"
elif age <= 20:  # If ... (NOT age <= 11) and (age <= 20)
    phase = "adolescence"
elif age <= 35:  # If ... (NOT age <= 20) and (age <= 35)
    phase = "early adulthood"
elif age <= 50:  # If ... (NOT age <= 35) and (age <= 50)
    phase = "midlife"
elif age < 80:   # If ... (NOT age <= 50) and (age < 80)
    phase = "mature adulthood"
else:  # otherwise, i.e., if age >= 8
    phase = "late adulthood"

print(f"A person of {age} years is in their {phase}.")
```

↓ `python3 if_elif_example.py` ↓

Listing 6.8: The stdout of the program `if_elif_example.py` given in Listing 6.7.

```
A person of 42 years is in their midlife.
```

Now, if you followed the book so far, you have noticed that I am promoting using static code analysis tools. For example, we introduced `Ruff` in Useful Tool 5 back in Section 5.2. Of course, we should apply such tools to all of our code. We now apply `Ruff` to our program `if_else_nested.py` given Listing 6.5. It produces the output shown in Listing 6.9. This is an example of how a linter can help us to improve the code quality and make our programs more compact.

If we implement its recommendation, we obtain Listing 6.10. We can compact the `else ... if` branch into an `elif` statement and the nested `else` can be moved one indentation level up. This program achieves in 15 lines for what Listing 6.5 needs 16, and it is easier to read, too. (In this program, I left the additional example of the `min` and `max` function away.)

Best Practice 32

Prefer `elif` over nested `else ... if` constructs [378].

Listing 6.9: The results of linting with `Ruff` of the program given in Listing 6.5.

```
$ ruff check --target-version py312 --select=A,AIR,ANN,ASYNC,B,BLE,C,C4,COM
    ↪ ,D,DJ,DTZ,E,ERA,EXE,F,FA,FIX,FLY,FURB,G,I,ICN,INP,ISC,INT,LOG,N,NPY,
    ↪ PERF,PIE,PLC,PLE,PLR,PLW,PT,PYI,Q,RET,RSE,RUF,S,SIM,T,T10,TD,TID,TRY,
    ↪ UP,W,YTT --ignore=A005,ANN001,ANN002,ANN003,ANN204,ANN401,B008,B009,
    ↪ B010,C901,D203,D208,D212,D401,D407,D413,INP001,N801,PLC2801,PLR0904,
    ↪ PLR0911,PLR0912,PLR0913,PLR0914,PLR0915,PLR0916,PLR0917,PLR1702,
    ↪ PLR2004,PLR6301,PT011,PT012,PT013,PYI041,RUF100,S,T201,TRY003,UP035,W
    ↪  --line-length 79 if_else_nested.py
PLR5501 Use `elif` instead of `else` then `if`, to reduce indentation
  --> if_else_nested.py:12:1
10 |       else:      # This means that a > b and c >= a.
11 |           print(f"{c} is the greatest number.")
12 | / else:          # This means that b >= a.
13 | |     if b > c:  # This means that b >= a and b > c.
   | |____^
14 |           print(f"{b} is the greatest number.")
15 |       else:      # This means that b >= a and c >= b.
help: Convert to `elif`
11 |         print(f"{c} is the greatest number.")
   - else:          # This means that b >= a.
   -     if b > c:  # This means that b >= a and b > c.
   -         print(f"{b} is the greatest number.")
   -     else:      # This means that b >= a and c >= b.
   -         print(f"{c} is the greatest number.")
12 + elif b > c:  # This means that b >= a and b > c.
13 +     print(f"{b} is the greatest number.")
14 + else:      # This means that b >= a and c >= b.
15 +     print(f"{c} is the greatest number.")
16 |
note: This is an unsafe fix and may change runtime behavior
Found 1 error.
1 unsafe fix available with `--fix --unsafe-fixes`.
# ruff 0.17.3 failed with exit code 1.
```

Listing 6.10: An example of using the nested `if ... elif` statement based on the recommendations of the `Ruff` linter applied to Listing 6.5. (stored in file `if_elif_nested.py`; output in Listing 6.11)

```
"""An example of using the nested `if-else` statements and `elif`."""

a: int = 13  # the first number
b: int = 7   # the second number
c: int = 9   # the third number

if a > b:       # This means that a > b
    if a > c:   # This means that a > b and a > c.
        print(f"{a} is the greatest number.")
    else:       # This means that a > b and c >= a.
        print(f"{c} is the greatest number.")
elif b > c:     # This means that b >= a and b > c.
    print(f"{b} is the greatest number.")
else:           # This means that b >= a and c >= b.
    print(f"{c} is the greatest number.")
```

↓ `python3 if_elif_nested.py` ↓

Listing 6.11: The stdout of the program `if_elif_nested.py` given in Listing 6.10.

```
13 is the greatest number.
```

6.4 The Inline Ternary `if...else` Statement

A very common use case of `if...else` statements is to conditionally assign values to variables. In Listing 6.12 we display such a situation. We want to write some code that tells us whether a number is large or small and positive or negative. For this purpose, we assume that the number is stored in an integer variable `number`. As an example, we choose the value `100`. For the sake of this example, let's assume that a `number` whose absolute value |`number`| is less than ten should be small.

The absolute value of a number in Python can be computed using the function `abs`. Hence, we can build the condition `if abs(number)< 10:`. We store the string `"small"` in the variable `size` if the condition evaluates to `True`. Otherwise, we store `"large"` in `size`. Similarly, we can define the string variable `sign`. We store `"negative"` in `sign` if `number < 0`, `"positive"` if `number > 0`, and `"unsigned"` otherwise, i.e., if `number == 0`. Finally, we can `print` the result of these conditions using an f-string.

Now you notice that in the `if...else` statement, all we did was to assign values to the same variable `size`. Likewise, in the `if...elif...else` statement, each branch *only* assigned one value to the same variable `sign`. There are many cases where we use an alternative *only* to decide which value should be stored in a certain variable. In languages like C and Java, you have a ternary operator looking like `condition ? valueIfTrue : valueIfFalse` as shorthand in such situations. Python, too, offers us a compact syntax for this, and Ruff can again give us some idea about that in Listing 6.14: The inline ternary `if...else` statement [591], which has the following syntax:

Listing 6.12: An example of using the nested `if ... else` statements that could be inlined. See Listing 6.15 for the more compact inlined variant. (stored in file `if_else_could_be_inline.py`; output in Listing 6.13)

```
"""An example of if-elif-else expressions that could be inlined."""

number: int = 100  # the number

# Let's say: Numbers with an absolute value less than ten are small.
# If their absolute value is >= ten, they are large.
size: str
if abs(number) < 10:  # Just a random threshold for this example...
    size = "small"    # If |number| < 10, we say the number is "small".
else:
    size = "large"    # If |number| >= 10, we say the number is "large".

# Numbers can be positive, negative, or unsigned (0 is unsigned).
sign: str
if number < 0:    # If the number is < 0, the sign is "negative".
    sign = "negative"
elif number > 0:  # Otherwise, if it is > 0, the sign is "positive".
    sign = "positive"
else:             # "unsigned" means neither "positive" nor "negative".
    sign = "unsigned"

print(f"The number {number} is {size} and {sign}.")  # Print the result.
```

↓ `python3 if_else_could_be_inline.py` ↓

Listing 6.13: The stdout of the program `if_else_could_be_inline.py` given in Listing 6.12.

```
The number 100 is large and positive.
```

Listing 6.14: The results of linting with `Ruff` of the program given in Listing 6.12.

```
$ ruff check --target-version py312 --select=A,AIR,ANN,ASYNC,B,BLE,C,C4,COM
    ↪ ,D,DJ,DTZ,E,ERA,EXE,F,FA,FIX,FLY,FURB,G,I,ICN,INP,ISC,INT,LOG,N,NPY,
    ↪ PERF,PIE,PLC,PLE,PLR,PLW,PT,PYI,Q,RET,RSE,RUF,S,SIM,T,T10,TD,TID,TRY,
    ↪ UP,W,YTT --ignore=A005,ANN001,ANN002,ANN003,ANN204,ANN401,B008,B009,
    ↪ B010,C901,D203,D208,D212,D401,D407,D413,INP001,N801,PLC2801,PLR0904,
    ↪ PLR0911,PLR0912,PLR0913,PLR0914,PLR0915,PLR0916,PLR0917,PLR1702,
    ↪ PLR2004,PLR6301,PT011,PT012,PT013,PYI041,RUF100,S,T201,TRY003,UP035,W
    ↪  --line-length 79 if_else_could_be_inline.py
SIM108 Use ternary operator `size = "small" if abs(number) < 10 else "large
    ↪ "` instead of `if`-`else`-block
  --> if_else_could_be_inline.py:8:1
 6 |    # If their absolute value is >= ten, they are large.
 7 |    size: str
 8 | / if abs(number) < 10:  # Just a random threshold for this example...
 9 | |      size = "small"    # If |number| < 10, we say the number is "
    ↪ small".
10 | | else:
11 | |      size = "large"    # If |number| >= 10, we say the number is "
    ↪ large".
   | |__________________^
12 |
13 |    # Numbers can be positive, negative, or unsigned (0 is unsigned).
help: Replace `if`-`else`-block with `size = "small" if abs(number) < 10
    ↪ else "large"`
Found 1 error.
# ruff 0.17.3 failed with exit code 1.
```

```
"""The syntax of a inline "if-else"-statements in Python."""

# If conditionForUsingValueA evaluates to True, then valueA will be
# assigned to variable. Otherwise, valueB will be assigned to variable.
variable = valueA if conditionForUsingValueA else valueB

# If conditionForUsingValueA evaluates to True, then valueA will be
# assigned to variable. Otherwise, conditionForUsingValueB is evaluated.
# If conditionForUsingValueB did evaluate to True, then valueB is
# assigned to variable (but of course, only if conditionForUsingValueA
# was False). If conditionForUsingValueB did evaluate to False (and
# conditionForUsingValueA was also False), then valueC will be assigned
# to variable.
variable = valueA if conditionForUsingValueA else (valueB if
    ↪ conditionForUsingValueB else valueC)
```

In the first variant, the value `valueA` will be assigned to `variable` if `conditionForUsingValueA` evaluates to `True`. Otherwise, `valueB` is assigned. This statement can again arbitrarily nested, as we show in the second variant: Here, again, value `valueA` will be assigned to `variable` if `conditionForUsingValueA` evaluates to `True`. Otherwise, `valueB` is used if `conditionForUsingValueB` evaluates to `True` and, again otherwise, `valueC` is used.

Best Practice 33

If your `if...else` statement is only used to decide which value to assign to a variable, use the inline ternary variant discussed in Section 6.4, as it is more compact [378].

We apply this syntax to refine Listing 6.12 and obtain the much more compact Listing 6.15. Both programs are equivalent, but the second one only has 13 instead of 22 lines of code. Notice again how a linter can help us to refine and compactify our code.

Listing 6.15: An example of using the inline `if ... else` expression to shorten Listing 6.12, which incorporates the suggestion by `Ruff` in Listing 6.14. (stored in file `inline_if_else.py`; output in Listing 6.16)

```
"""An example of using the inline if-else expression."""

number: int = 100  # the number

# Let's say: Numbers with an absolute value less than ten are small.
# If their absolute value is >= ten, they are large.
size: str = "small" if abs(number) < 10 else "large"

# Numbers can be positive, negative, or unsigned (0 is unsigned).
sign: str = "negative" if number < 0 else (
    "positive" if number > 0 else "unsigned")

print(f"The number {number} is {size} and {sign}.")  # Print the result.
```

↓ `python3 inline_if_else.py` ↓

Listing 6.16: The `stdout` of the program `inline_if_else.py` given in Listing 6.15.

```
The number 100 is large and positive.
```

6.5 Summary

With the statements we discussed in this section, you are now able to create a program that makes decisions based on data. Before this, we could only perform straightforward computations and calculate the results of simple functions. Now our variables can receive the result of a function $\mathcal{A}$ if the input meets a condition B and otherwise the result of a function $\mathcal{C}$. This is already quite nice. For example, we can now implement and hard-code decision trees [487, 506] and Listing 6.7 is basically an example of that. With alternatives, we can basically "jump over" instructions. Still, with the exceptions that we can now omit steps, the instructions in our programs are still executed in the sequence in which we wrote them down. While our control flow can now branch, it cannot perform anything more fancy and advanced ... like looping back upon itself...

Chapter 7

Loops

When we are working with sequences of data, we do not just want to perform an action on one specific data element. We instead often want to apply the actions repetitively to many or even all data elements. *Loops* allow us to do just that, to perform the same actions multiple times.

In principle, alternatives with `if...else` allow us skip certain parts of the code. In other words, they allow the control flow to jump over them and to continue with what follows after. Loops are a complementary concept as they allow us to repeat certain parts of the code. This is basically equivalent to let the control flow jump back to the beginning the the code to be repeated after finishing with its present execution. Before we delve into this new concept, let us pause a minute and enjoy the significance of what we are going to learn.

Definition 7.1: Structured Programming Theorem

The *Structured Program Theorem* states that any computable function can be computed using only three different control flow elements, namely *(1)* the sequential execution of instructions, *(2)* the selective execution of instructions (i.e., alternatives, conditionals), and *(3)* the iterative (repetitive) execution of instructions (until some condition is met) [65, 66, 250, 496].

We already learned the first two elements of the control flow. And now we are going to learn the third one, namely *loops*. And after we learn it, we can, basically, perform *any* computation that is possible on today's computers. Everything that follows after that are devices to make the life of programmers easier.

7.1 The `for` Loop Statement

The most basic such sequence in Python may be the `for` loop, which has the following pattern[404]:

```
"""The syntax of a for-loop in Python."""

for loopVariable in sequence:
    loop body statement 1  # The loop body is executed for every item
    loop body statement 2  # in the sequence.
    # ...

normal statement 1  # After the sequence is exhausted, the code after
normal statement 2  # the for loop will be executed.
# ...
```

The keyword `for` is followed by a loop variable. Then comes the keyword `in`, the `sequence` we want to iterate over, and finally a colon (`:`). This variable will iteratively take on the values in the `sequence`. The loop body statements is the indented block below the loop header. It will be executed for each of these values and can access the values via the loop variable. Each time the body of the loop is executed is called an *iteration* of the loop. After the loop, we leave one blank line followed by the code that will be executed after the loop completes.

Listing 7.1: An example of using the `for` loop over a `ranges` of integer numbers. (stored in file `for_loop_range.py`; output in Listing 7.2)

```
"""Apply a for loop over a range."""

# We will construct a dictionary holding square numbers.
squares: dict[int, int] = {}  # Initialize `squares` as empty dict.

for i in range(5):  # i takes on the values 0, 1, 2, 3, 4 -- one by one.
    squares[i] = i * i  # Stores 0: 0, 1: 1, 2: 4, 3: 9, 4: 16.

for i in range(6, 9):  # i takes on the values 6, 7, and 8 one by one.
    squares[i] = i * i  # Stores 6: 36, 7: 49, 8: 64.

for i in range(20, 27, 2):  # i takes on the values 20, 22, 24, and 26.
    squares[i] = i * i  # Stores 20: 400, 22: 484, 24: 576, 26: 676.

for i in range(40, 30, -3):  # i takes on the values 40, 37, 34, and 31.
    squares[i] = i * i  # Stores 40: 1600, 37: 1369, 34: 1156, 31: 961.

print(squares)  # Print the dictionary.

for _ in range(3):  # Iterate the loop three times. Ignore counter `_`.
    print("Hello World!")  # We don't need the counter.
```

↓ `python3 for_loop_range.py` ↓

Listing 7.2: The stdout of the program `for_loop_range.py` given in Listing 7.1.

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 6: 36, 7: 49, 8: 64, 20: 400, 22: 484, 24:
    ↪ 576, 26: 676, 40: 1600, 37: 1369, 34: 1156, 31: 961}
Hello World!
Hello World!
Hello World!
```

We already learned about several collection datatypes which could be used as `sequence` to iterate over, namely `lists`, `tuples`, `sets`, and `dicts`. However, in its most simple form, the `for` loop is applied to a `range` of integer numbers. Ranges are sequences which work basically like slices (see Sections 3.6.1 and 5.1).

`range(5)` will give us a sequence of integers starting with 0, incrementing in steps of 1, and reaching up to right *before* 5, i.e., the integer range 0..4. `range(6, 9)` gives the sequence of integers starting with 6, incrementing in steps of 1, and stopping right *before* 9, i.e., the integer range 6..8. Finally, `range(20, 27, 2)` results in a sequence of integers that begins at 20, increments by 2 in each step, and ends right before 27. This is the sequence $(20, 22, 24, 26)$. `ranges`, like slices, can also have negative increments: The `range(40, 30, -3)` starts with 40 and stops before reaching 30 and decrements by 3 in each step. This is equivalent to the set $(40, 37, 34, 31)$.

In program `for_loop_range.py` given as Listing 7.1, we loop over exactly these ranges. In this listing, we try to create a dictionary (see Section 5.5) where some integer numbers are mapped to their squares. We use four `for` loops to fill this dictionary with data. In each of these first four `for` loops, we use `i` as the loop variable.

When iterating over the `range(5)` in the first loop, `i` will hold the value `0` in the first iteration (= execution of the loop body). The loop body `squares[i] = i * i` will thus effectively be `squares[0] = 0` and thus store the value `0` under key `0` into the dictionary `squares`. In the second iteration, `i` will hold the value `1`. Then, the body `squares[i] = i * i` will effectively be `squares[1] = 1`. In the third iteration, `i` will hold the value `2` and the body will perform `squares[2] = 4`. Next, `i = 3` and `squares[3] = 9` will be executed and in the laste iteration of the first loop, we store `squares[4] = 16`. With this, the first loop is completed.

In the second loop, which uses `range(6, 9)`, `i` will take on the values `6`, `7`, and `8`, one by one. The dictionary `squares` will thus be extended with the values `squares[6] = 36`, `squares[7] = 49`,

Listing 7.3: A variant of Listing 4.3 which uses a `for` loop instead of five copies of the same instructions. (stored in file `for_loop_pi_liu_hui.py`; output in Listing 7.4)

```
"""We execute Liu Hui's method to approximate pi in a loop."""
from math import pi, sqrt

print(f"We use Liu Hui's Method to Approximate \u03c0\u2248{pi}.")
e: int = 6  # the number of edges: We start with a hexagon, i.e., e=6.
s: float = 1.0  # the side length: Initially 1, i.e., radius is also 1.

for _ in range(6):
    print(f"{e} edges, side length={s} give \u03c0\u2248{e * s / 2}.")
    e *= 2  # We double the number of edges...
    s = sqrt(2 - sqrt(4 - (s ** 2)))  # ...and recompute the side length
```

↓ `python3 for_loop_pi_liu_hui.py` ↓

Listing 7.4: The stdout of the program `for_loop_pi_liu_hui.py` given in Listing 7.3.

```
We use Liu Hui's Method to Approximate π≈3.141592653589793.
6 edges, side length=1.0 give π≈3.0.
12 edges, side length=0.5176380902050416 give π≈3.1058285412302498.
24 edges, side length=0.2610523844401031 give π≈3.132628613281237.
48 edges, side length=0.13080625846028635 give π≈3.139350203046872.
96 edges, side length=0.0654381656435527 give π≈3.14103195089053.
192 edges, side length=0.03272346325297234 give π≈3.1414524722853443.
```

and `squares[8] = 64`. In the third loop, iterating over `range(20, 27, 2)`, the following updates will be performed one by one `squares[20] = 400`, `squares[22] = 484`, `squares[24] = 576`, and `squares[26] = 676`. In the fourth loop, `i` takes on the values of the sequence `range(40, 30, -3)`, which has the negative step length `-3`. `i` therefore first becomes `40`, then `37` in the second iteration, then `34`, and, finally, `31`. We then `print` the dictionary and get the expected output in Listing 7.2.

Best Practice 34

If we do not care about the value of a variable (or parameter), we should name it `_` [324]. This information is useful for other programmers as well as static code analysis tools.

At the end of Listing 7.1 we show this special case: We want to `print` `"Hello World!"` three times. Instead of copying the line `print("Hello World!")` three times, we put it in a loop iterating over `range(3)`. However, nowhere in the loop body we care about the value of the loop variable. We thus simply call it `_`.

Because of that, everybody reading this code immediately understands that actual value of the counter variable is uninteresting to us. They will see that we just want to do something three times, that it does not matter which index which iteration has. If we would not call it that, then another programmer seeing our code (or a static code analysis tool for that matter) could be confused as to why we do not use the loop variable. Always remember that "real" code could be much more complicated, and any semantic hint we can include to convey our intentions will be helpful.

With these new tools at hand, we can revisit our program Listing 4.3 for approximating π from back in Section 4.1.2. In the original program , we executed the same code five times. We always doubled the number `e` of edges by doing `e *= 2`. Then we updated the sidelength `s` of the regular `e`-gon. Finally, we printed the approximation of π as `e * s / 2`. Instead of copying the code several times, we can just as well put it inside a loop. This reduces the lines of code from over 25 to about 10 in program `for_loop_pi_liu_hui.py` given in Listing 7.3. The outputs in Listings 4.4 and 7.4 are exactly the same.

Listing 7.5: An example of the `continue` and `break` statements in a `for` loop. (stored in file `for_loop_continue_break.py`; output in Listing 7.6)

```
"""Here we explore the `break` and `continue` statements."""

for i in range(15):  # i takes on the values from 0 to 14 one by one.
    s: str = f"i is now {i}."  # Create a string with the value of i.
    if i > 10:  # If i is greater than 10, then...
        break  # ...abort the loop altogether, do not execute next line.
    if 5 <= i <= 8:  # If i is in the range of 5..8, then...
        continue  # ...skip the rest of the loop body, do next iteration
    print(s)  # We get here if neither continue nor break were called.

print("All done.")  # We always get here.
```

↓ `python3 for_loop_continue_break.py` ↓

Listing 7.6: The stdout of the program `for_loop_continue_break.py` given in Listing 7.5.

```
i is now 0.
i is now 1.
i is now 2.
i is now 3.
i is now 4.
i is now 9.
i is now 10.
All done.
```

7.2 The `continue` and `break` Statements

Loops often have complex bodies, maybe containing conditional statements or other loops. It is not an uncommon situation that, after performing some computations in the body of the loop, we already know that we can continue directly with the next iteration instead of executing the remainder of the loop body. Sometimes we also find that we can entirely stop with the loop and continue with whatever instructions come after it, even if we did not yet exhaust the sequence over which we are iterating. The former can be achieved using the keyword `continue` and the latter with the `break` statement.

An example of both statements is given in Listing 7.5. Here, we iterate the variable `i` over the 15 values from `0` to `14`, i.e., over `range(15)`. In the loop body, we first create a string `s` with the current value of `i` via the f-string `f"i is now {i}."`. The very last instruction of the loop body, `print(s)`, prints this string.

While `i` would go from `0` to `14`, we actually want to abort the loop as soon as `i` becomes greater than `10`. We could modify the `range` over which we loop, which would be the reasonable thing to do. Just for fun, we here instead want to use the `break` statement. We therefore wrap it into the conditional `if i > 10:`. If and only if `i > 10`, the `break` statement is executed. The `break` will then immediately abort the loop, without even finishing the current iteration. No further instruction in the loop body is executed and no further iteration is performed. Instead, the process will continue after the loop, with the line `print("All done.")`.

If `i > 10` did not hold, the rest of the loop body is executed. For the case that $i \in 5..8$, we now want to directly jump to the next loop iteration. We do not want to print text and, thus, do not want to invoke the `print` in the loop body. We can achieve this by invoking the `continue` statement.

If `5 <= i <= 8` holds, we will invoke `continue`. This means that if `i == 5`, the `continue` statement lets the control directly return to the head of the loop. The loop will set `i = 6`. Then, the same thing will happen again, and again. This will continue until `i == 9`. The condition `5 <= i <= 8` is *not* met for all $i \in 0..4 \cup 9..15$. The next line, namely the `print(s)`, can only be reached in these cases. Of course, we already know that the loop will terminate anyway as soon as `i == 11`.

As a result the program will print the string `s` only for $i \in 0..4 \cup \{9, 10\}$ before finally outputting `All done.`. This can be observed in the program output collected in Listing 7.6. With `break` and `continue`, we now have two tools that can help us to either abort any loop prematurely or to abort the current iteration of the loop prematurely (and continue with the next one, if any), respectively.

Listing 7.7: Computing a list of all primes from 2..200 using nested `for` loops. (stored in file `for_loop_nested_primes.py`; output in Listing 7.8)

```
"""Compute all primes less than 200 using two nested for loops."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = [2]  # the list for the primes; We know 2 is prime.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.

    for check in range(3, isqrt(candidate) + 1, 2):  # ...odd numbers
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_nested_primes.py` ↓

Listing 7.8: The stdout of the program `for_loop_nested_primes.py` given in Listing 7.7.

```
After 252 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37,
    ↪ 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107,
    ↪ 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181,
    ↪ 191, 193, 197, 199].
```

7.3 Nesting Loops

Like conditional statements, loops can be arbitrarily nested. The bodies of loops can contain other loops or alternatives. The branches of alternatives can contain loops as well. Let us explore this by computing the list of all prime numbers less than 200 in Listing 7.7.

Definition 7.2: Prime Number

A prime number $p \in \mathbb{N}_1$ is a positive integer $p > 1$, i.e., greater than one, that has no positive integer divisors other than 1 and p itself [135, 481, 626].

In our example program, we want to store all of these prime numbers $p < 200$ in the list `primes`. We know that `2` is a prime number and the only even one, so we directly initialize `primes = [2]`, i.e., we set `primes` to initially be a list containing only the integer `2`. This will allow us to later on only consider the odd numbers in the range 3..199. Just out of interest, we will count the total number of divisions that we need to perform until we have the full list of primes in the variable `n_divisions`.

To find all primes in the integer set 2..199, we let the loop variable `candidate` iterate over `range(3, 200, 2)`. This is the sequence of integer numbers starting `3` increasing with step length `2` and stopping right before `200`. We know that even numbers except 2 are not prime, so this should be OK. Therefore, `candidate` will iteratively become `3`, `5`, `7`, ..., and eventually `195`, `197`, and `199`.

For each value of `candidate`, we begin with the assumption that it is prime and then try to prove the opposite. We therefore first set a variable `is_prime = True` and then try to find an integer divisor of `candidate`. If we succeed in this, then we set `is_prime = False`. If we cannot find a divisor of `candidate`, then `is_prime` will remain `True`. In this case, we can add `candidate` to the list `primes`. At least, this is the plan.

We will implement this idea with a nested inner loop. Since the loop variable `candidate` will always be odd, only odd numbers can be potential divisors. Obviously, only integers greater than or equal to `3` are potential divisors. We also only need to explore whether a number `check` is a divisor of `candidate` if it is not bigger than $\sqrt{\texttt{candidate}}$. If we had three integer numbers a, b, and c such that $a = b * c$, then it must be that $c = \frac{a}{b}$. If $b > \sqrt{a}$, this means that $c < \frac{a}{\sqrt{a}}$, which means that $c < \sqrt{a}$. Viewing this the other way around, since $a = \sqrt{a} * \sqrt{a}$ holds by definition, it would be impossible that $a = b * c$ if both $b > \sqrt{a}$ and $c \geq \sqrt{a}$. Thus, if $b > \sqrt{a}$ was a divisor of a, then there also must be a divisor $c \leq \sqrt{a}$. And we would discover this divisor with the variable `check` before `check` reaches $\sqrt{\texttt{candidate}}$.

Now, most integer numbers do not have integer square roots. Since integer divisors cannot have fractions anyway, it is sufficient for us to use $\lfloor\sqrt{\texttt{candidate}}\rfloor$. In Python, such a "truncated" integer square root $\lfloor\sqrt{a}\rfloor$ of an integer number a can be computed using the `isqrt` function from the `math` module. We therefore `import` this function at the top of our program.

We can therefore iterate a second, inner loop variable `check` over `range(3, isqrt(candidate)+ 1, 2)`. If `candidate <= 8`, then this loop will never be executed because no number `check` with $3 \leq$ `check` $<$ `isqrt(candidate)` exists. Then, `is_prime` will remain `True` and we will append `candidate` to `primes` further down the outer loop body. If `candidate > 8`, then the loop is actually entered and `check` will go from `3` to $\lfloor\sqrt{\texttt{candidate}}\rfloor$.

In the body of our inner loop, we try to find out whether `check` is an integer divisor of `candidate`. We can do this by computing the remainder of the division `canddiate`/`check`. Back in Section 3.2.1 we learned the modulo division operator `%`, which does exactly that. If `candidate %check` is `0`, then we can divide `candidate` by `check` without remainder. Then, `candidate` is divisible by `check` and cannot be a prime number. We thus can set `is_prime = False`. We can also immediately exit the inner loop using the `break` statement. Once we know that `candidate` is not a prime number, we do not need to check further potential divisors. (Notice that we also count all the modulo division operations we perform by doing `n_divisions += 1` at the beginning of the inner loop.)

After the inner loop, we check whether `is_prime` is still `True`. If so, we append `candidate` to `primes` by invoking the `append` method of the list. Once we have completed the outer loop as well, we print the number `n_divisions` of divisions we have performed, the number `len(primes)` of prime numbers we have discovered and, finally, the list `primes` of prime numbers itself. The output in Listing 7.8 tells us that, after performing 252 divisions, we could identify 46 prime numbers inside 2..199. (OK, we ignored or, better, do not know, whether `isqrt` performed any divisions ... but let's not be too nit-picky.)

7.4 Loops over Sequences

When introducing the `for` loop, we stated that it iterates over sequences of data. We learned that `ranges` are the most basic such sequences. However, in Chapter 5 we learned about collection datatypes, namely `lists`, `tuples`, `sets`, and `dicts`. The elements of all of these datatypes can be accessed sequentially, too. This means that a `for` loop can iterate over them as well!

In Listing 7.9 we illustrate how this is done: It works exactly as if we were using `ranges`. In the program, we will build a list `txt` with strings that we will later write to the output.

First, we want to iterate over a list `lst` containing the four integers `[1, 2, 3, 50]`. This is done by simply writing `for i in lst`. Here, `i` is the loop variable and it will step-by-step take on all the values in `lst`, one by one. In the body of this loop, we invoke `txt.append(f"{i = }")`. The f-string will evaluate to `"i = 1"` in the first iteration, to `"i = 2"` in the second, to `"i = 3"` in the third, and to `"i = 50"` in the fourth and last iteration of this loop. These strings are appended to the list `txt` via the `append` method.

Then we move on and want to iterate over a tuple `tp`, which contains three floating point numbers, namely `(7.6, 9.4, 8.1)`. This works exactly in the same way: `for f in tp` will let the loop variable `f` take on the values `7.6`, `9.4`, and finally `8.1`. We again append their textual representation to the list `txt`, this time using the f-string `f"{f = }"`.

As third example, we create the set `st` containing the three strings `{"u", "v", "w"}`. We can iterate over the elements this set by, again, simply writing `for s in st`. This lets the loop variable `s` take on the values `"w"`, `"u"`, and `"v"`, *in an arbitrary order*. Remember that sets in Python are unordered (see Best Practice 28). This means that if we run the program twice, we may get different results. Either way, we can iterate over all the values in the set `st`. We again want to store these

Listing 7.9: An example iterating over the elements in different collection data structures using `for` loops. (stored in file `for_loop_sequence.py`; output in Listing 7.10)

```
"""Iterate over several different containers with `for` loops."""

txt: list[str] = []  # We will collect the output text in this list.

lst: list[int] = [1, 2, 3, 50]  # Create a list with 4 integers.
for i in lst:  # i takes on the values 1, 2, 3, and 50.
    txt.append(f"{i = }")  # We store "i = 1", "i = 2", "i = 3"...

tp: tuple[float, ...] = (7.6, 9.4, 8.1)  # Create a tuple with 3 floats.
for f in tp:  # i takes on the values 7.6, 9.4, and 8.1.
    txt.append(f"{f = }")  # We store "f = 7.6", "f = 9.4", ...

st: set[str] = {"u", "v", "w"}  # Create a set with 3 strings.
for s in st:  # s takes on the values "u", "v", and "w" (unordered!).
    txt.append(f"s = {s!r}")  # We store "s = 'u'", "s = 'v'", ...

dc: dict[float, bool] = {1.1: True, 2.5: False}  # Create a dictionary.
for k in dc:  # Iterate over the keys in the dictionary == 1.1 and 2.5.
    txt.append(f"{k = }")  # We store "k=1.1" and "k=2.5".
for v in dc.values():       # Iterate over the values in the dictionary.
    txt.append(f"{v = }")  # We store "v=True" and "v=False".
for k, v in dc.items():     # Iterate over the key-value combinations.
    txt.append(f"{k}: {v}")  # Store "1.1: True" and "2.5: False"

# Merge text into single string with separator ", " and print it.
print(", ".join(txt))
```

↓ `python3 for_loop_sequence.py` ↓

Listing 7.10: The stdout of the program `for_loop_sequence.py` given in Listing 7.9.

```
i = 1, i = 2, i = 3, i = 50, f = 7.6, f = 9.4, f = 8.1, s = 'w', s = 'u', s
    ↪ = 'v', k = 1.1, k = 2.5, v = True, v = False, 1.1: True, 2.5: False
```

values as nicely formatted strings in `txt`. This time we use the f-string `f"s = {s!r}"`. Notice the `!r` format specifier: It will print the *representation* of the string `s`, which basically means to put quotation marks (and to potentially use escape sequences for unprintable characters, see Section 3.6.4). Thus, we will add `"s = 'v'"`, `"s = 'u'"`, and `"s = 'w'"` to `txt` (in an arbitrary order).

As a fourth and final example, we create a dictionary `dc` that maps floating point numbers to Boolean values. It contains only the two entries, namely `{1.1: True, 2.5: False}`. Now a dictionary is a bit special: It maps keys to values. When working with the whole dictionary data structure `dc` as a collection, then we can access it in three ways:

1. Iterating over `dc` directly lets us view all the keys in the dictionary `dc`. This is equivalent to iterating over `dc.keys()`.
2. Iterating over `dc.values()` lets us access all the values in the dictionary `dc`.
3. Iterating over `dc.items()` lets us access all key-value pairs in the dictionary `dc` as tuples.

In our example program, we apply all of these methods. We iterate over the keys in `dc` via `for k in dc`, which lets `k` take on the values `1.1` and `2.5`, one after the other. We iterate over the values in `dc` via `for v in dc.values()`, which lets `v` take on the values `True` and `False`, one after the other.

Finally, we iterate over all key-value pairs. Look closely at this one: We *could* write `for t in dc.items()`, which would let a loop variable `t` take on the tuple values `(1.1, True)` and then `2.5: False`. However, back in Section 5.3, we learned about tuple unpacking. So instead, we write `for k, v in dc.items()`. This is basically a shortcut for writing `for t in dc.items()` followed

Listing 7.11: Computing a list of all primes from 2..200 using nested `for` loops. Compared to Listing 7.7, this program is more efficient because it only tests primes as divisors. (stored in file `for_loop_sequence_primes.py`; output in Listing 7.12)

```
"""Compute all primes less than 200, with a for loop over a sequence."""

from math import isqrt  # the integer square root == int(sqrt(...))

primes: list[int] = []  # The list for the primes is initially empty.
n_divisions: int = 0  # We want to know how many divisions we needed.

for candidate in range(3, 200, 2):  # ...all odd numbers less than 200.
    is_prime: bool = True  # Let us assume that `candidate` is prime.
    limit: int = isqrt(candidate)  # Get the maximum possible divisor.

    for check in primes:  # We only test with the odd primes we got.
        if check > limit:  # If the potential divisor is too big, then
            break  # we can stop the inner loop here.
        n_divisions += 1  # Every test requires one modulo division.
        if candidate % check == 0:  # modulo == 0: division without rest
            is_prime = False  # check divides candidate evenly, so
            break  # candidate is not prime. We can stop the inner loop.

    if is_prime:  # If True: no smaller number divides candidate evenly.
        primes.append(candidate)  # Store candidate in primes list.

primes.insert(0, 2)  # Now we insert 2 at the beginning of the list.

# Finally, print the list of prime numbers.
print(f"After {n_divisions} divisions: {len(primes)} primes {primes}.")
```

↓ `python3 for_loop_sequence_primes.py` ↓

Listing 7.12: The stdout of the program `for_loop_sequence_primes.py` given in Listing 7.11.

```
After 224 divisions: 46 primes [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37,
    ↪ 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107,
    ↪ 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181,
    ↪ 191, 193, 197, 199].
```

by `k, v = t`. It directly unpacks the tuples in the sequence `dc.items`. We thus get pairs of `k=1.1`, `v=True` and `k=2.5`, `v=False`. All of them are again appended to our text list `txt`.

After all of these loops, we have ended up with a list `txt` containing 16 strings. We want to combine all of them into a single string, using `", "` as a separator. We could do this in another loop. However, Python offers us a much more efficient shorthand for this: The method `join` of the class `str`.

For any string `z`, `z.join(seq)` accepts a sequence `seq` of strings. It then concatenates all the strings in `seq`, placing `z` as separator between any two strings. Thus, invoking `", ".join(txt)` produces `i = 1, i = 2, i = 3, i = 50, f = 7.6, ...`. This text is finally is written to the output via the `print` function, giving us the results shown in Listing 7.10.

Let us now use this new ability to iterate over collection data structures to improve upon our prime number enumeration program Listing 7.7. Back when writing that program, we immediately stored `2` as a prime into our list `primes`. We then only tried the odd numbers less than 200 as potential prime number `candidates`. For each potential prime number `candidate`, we tested all (odd) numbers `check` from `range(3, isqrt(candidate)+ 1, 2)` as possible divisors.

When thinking about this, we realize that actually, only the prime numbers in this range are interesting. For example, we do never need to check whether `candidate` is divisible by 9, because we will have already checked whether it can be divided by 3. We also never need to check whether we can divide it by 55, because we will have already checked 5.

But how do we get a sequence that only contains prime numbers? Well... actually, we are

constructing it already! `primes` contains all the prime numbers less than `candidate`.

Therefore, in our new and more efficient program Listing 7.11, we replace the head of the inner loop with `for check in primes:`. As a minor tweak, we omit 2 from this list at the beginning and instead insert it at the end of the program. This way, we avoid the useless check whether an odd number `candidate` can be divided by 2.

Anyway, the outer loop will step-by-step add prime numbers to `primes`. For each value of the loop variable `candidate`, `primes` will contain all prime numbers which are smaller than `candidate` (except 2, that is).

We only need to check those values of `check` which are less than or equal to `isqrt(candidate)`. This is because if `check` was greater than $\sqrt{\texttt{candidate}}$, the division would yield a quotient q less than `check`. If this quotient q would be a prime number, then this means that we already tested it before. If it was not a prime number, then it means that q would be divisible by another number r. Since r would necessarily be less than q, it would also be less than `check`, and hence, we would have tested it before if it was prime. If it was not prime, then the argument applies recursively until we eventually hit a prime number.

Therefore, we store this value in a new variable `limit` to avoid re-computing it in the inner loop. The `break` statement lets us exit the inner loop as soon as we hit this `limit`. After the main loop completes, we insert 2 at index 0 into `primes`.

The lists of primes that our programs generate in Listings 7.8 and 7.12 are exactly the same. However, our new program needs to perform only 224 divisions instead of 252.

Later, in Chapter 10, we will learn that loops in Python can be applied to the more general concept of `Iterators`. We will also see how the `for` keyword can be used to construct sequences via a process called *comprehensions*.

7.5 `enumerate` and Interlude: Pylint

Assume that we have a list `data` of integer values `v`. If we want to iterate over all the values `v` in `data`, then `for v in data` would do the trick. However, what if we also want to know the current index `i` for each data element during the iteration? Then this construct no longer works.

We basically have two choices: As first option, we can iterate over `data` by accessing the elements by their index `i` directly. In this case, we would use a `range` that goes from `0` to `len(data)- 1`. The

Listing 7.13: Enumerating over the index-value pairs of a list (inefficiently) by using the index as loop variable and directly indexing the container. (stored in file `for_loop_no_enumerate_1.py`; output in Listing 7.14)

```
"""
Enumerate the index-value pairs in a list: Idea 1.

Lists and tuples can be indexed directly. Therefore, we can let the
index `i` go from `0` to `len(data)-1` and access the data elements by
using the index `i`. This would not work for stets or dicts.
"""

data: list[int] = [1, 2, 3, 5]  # The data to iterate over.

for i in range(len(data)):          # We iterate over indices and
    print(f"data[{i}]={data[i]}")   # use them to get the data elements.
```

↓ `python3 for_loop_no_enumerate_1.py` ↓

Listing 7.14: The stdout of the program `for_loop_no_enumerate_1.py` given in Listing 7.13.

```
data[0]=1
data[1]=2
data[2]=3
data[3]=5
```

Listing 7.15: Enumerating over the index-value pairs of a list (inefficiently) by iterating over the container and manually maintaining the current index in a variable. (stored in file `for_loop_no_enumerate_2.py`; output in Listing 7.16)

```
"""
Enumerate the index-value pairs in a list: Idea 2.

We can iterate over any container class directly with a for loop.
In this case, we simply increment the index by ourselves.
"""

data: list[int] = [1, 2, 3, 5]  # The data to iterate over.

i = 0           # The initial index is 0.
for v in data:  # We iterate over the data values directly.
    print(f"data[{i}]={v}")
    i += 1      # And we increment the index by ourselves.
```

↓ `python3 for_loop_no_enumerate_2.py` ↓

Listing 7.16: The stdout of the program `for_loop_no_enumerate_2.py` given in Listing 7.15.

```
data[0]=1
data[1]=2
data[2]=3
data[3]=5
```

head of the loop now looks like this: `for i in range(len(data))`. The data values `v` are then accessed via `data[i]`. This is illustrated in Listing 7.13.

The drawback of this method is that it will *only* work for `lists` and `tuples`. Among the collection data structures discussed so far, only `lists` and `tuples` are directly indexable. However, as we have seen, we can iterate over all of them. Therefore, the second option to achieve our goal would be to just "normally" iterate over the container. Of course, then we do not know the index of the current element. Well, nothing can stop us from using an additional integer variable `i` and maintain its proper value by ourselves. We would initialize it as `i: int = 0` before the loop. We would then increment it as `i += 1` at the bottom of the loop body. Of course, `i += 1` is equivalent to `i = i + 1`. This would work with all collection classes. This is illustrated in Listing 7.15.

Both options are a bit iffy, as they introduce more and less readable additional code. There is a third option, though. And because I advocate using static code analysis tools, I let another such tool discover this option for us.

Useful Tool 6

`Pylint` is a Python linter that analyzes code for style, potential errors, and possible improvements [461]. It can be installed via `pip install pylint` as shown in Figure 7.1 on page 153. You can then apply `Pylint` using the command `pylint fileToScan.py`. We provide a script for using `Pylint` with a reasonable default configuration in Listing 16.3 on page 441.

We first install `Pylint` citePC2024PL. We therefore open a terminal by either pressing Ctrl+Alt+T under `Ubuntu Linux` or Ctrl+Alt+T or under Microsoft `Windows` via press ⊞+R, type in `cmd`, and hit ↵. As sketched in Figure 7.1, we then type in `pip install pylint` and hit ↵. (Normally, we would do that in a virtual environment, which we discuss later in Section 14.1.) Anyway, `Pylint` gets installed.

We can apply it a file `fileToScan.py` by typing `pylint fileToScan.py` in a terminal (`fileToScan.py` can also be a directory). Like with `Ruff`, in the context of this book, we will disable some linter rules for `Pylint`, too. This can be done by adding the parameter `--disable=...` with the rules to be deactivated. The list of rules that `Pylint` applies when analyzing a file can be found at [461].

Let us now apply both our new linter `Pylint` as well as `Ruff`, which we ave already used,

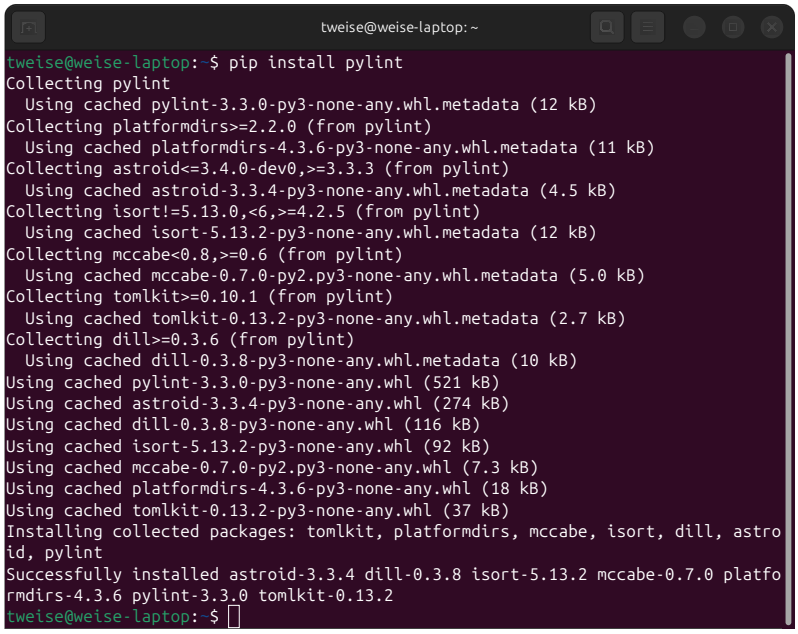


Figure 7.1: Installing `Pylint` in a `Ubuntu` terminal via `pip` (see Section 14.1 for a discussion of how packages can be installed).

Listing 7.17: The results of static code analysis using the `Pylint` linter for the program `for_loop_no_enumerate_1.py` given in Listing 7.13.

```
$ pylint for_loop_no_enumerate_1.py --disable=C0103,C0302,C0325,R0801,R0901
    ↪ ,R0902,R0903,R0911,R0912,R0913,R0914,R0915,R1702,R1728,W0212,W0238,
    ↪ W0703
************* Module for_loop_no_enumerate_1
for_loop_no_enumerate_1.py:11:0: C0200: Consider using enumerate instead of
    ↪  iterating with range and len (consider-using-enumerate)
-----------------------------------
Your code has been rated at 6.67/10
# pylint 4.1.3 failed with exit code 16.
```

to the two ideas for enumerating over the data and indices of collections. We first analyze `for_loop_no_enumerate_1.py`, the program where we let the index variable iterate and use it to access the data elements. In Listing 7.17, `Pylint` suggests that we should iterate over the list `data` using the `enumerate` function, whereas `Ruff`, at the time of this writing, has no issue with this approach (see Listing 7.18). Now we analyze `for_loop_no_enumerate_2.py`, where we iterate over the elements of the container and manually maintain an index variable `i`. Interestingly, this time `Ruff` suggests us to "Use '`enumerate()`' for index variable '`i`' in '`for`' loop", as shown in Listing 7.20. `Pylint`, on the other hand, does not discover any problem with `for_loop_no_enumerate_2.py`.

Our two linters have discovered the same issue, just with different programs. They independently propose using a function called `enumerate` to solve the problem of iterating over collections and knowing the index of the current element. But what is `enumerate`?

`enumerate` accepts a collection as parameter and creates a sequence of index-value tuples [261]. We can unpack such tuples while iterating over this sequence (see Listing 7.9). Therefore, when using `enumerate`, we can change the head of our loop to the form `for i, v in enumerate(data)`. Notice that the index `i` comes first and the value `v` comes second in this unpacked-tuple-based enumeration. This is implemented in program `for_loop_enumerate.py` given as Listing 7.21. And it would work exactly the same, regardless whether `data` was a `list`, `set`, `tuple`, or `dict`ionary.

We also confirmed Best Practice 24: It makes sense to use several different tools for static code analysis. Each tool has its own strengths. Some, like `Mypy`, can discover problems with typing. Others, like `Ruff` and `Pylint`, can make suggestions on how to improve our code. But even if two tools have the same "area of expertise," they may still discover different issues. Had we implemented

Listing 7.18: The results of static code analysis using the Ruff linter for the program for_loop_no_enumerate_1.py given in Listing 7.13.

```
$ ruff check --target-version py312 --select=A,AIR,ANN,ASYNC,B,BLE,C,C4,COM
    ↪ ,D,DJ,DTZ,E,ERA,EXE,F,FA,FIX,FLY,FURB,G,I,ICN,INP,ISC,INT,LOG,N,NPY,
    ↪ PERF,PIE,PLC,PLE,PLR,PLW,PT,PYI,Q,RET,RSE,RUF,S,SIM,T,T10,TD,TID,TRY,
    ↪ UP,W,YTT --ignore=A005,ANN001,ANN002,ANN003,ANN204,ANN401,B008,B009,
    ↪ B010,C901,D203,D208,D212,D401,D407,D413,INP001,N801,PLC2801,PLR0904,
    ↪ PLR0911,PLR0912,PLR0913,PLR0914,PLR0915,PLR0916,PLR0917,PLR1702,
    ↪ PLR2004,PLR6301,PT011,PT012,PT013,PYI041,RUF100,S,T201,TRY003,UP035,W
    ↪  --line-length 79 for_loop_no_enumerate_1.py
All checks passed!
# ruff 0.17.3 succeeded with exit code 0.
```

Listing 7.19: The results of static code analysis using the Pylint linter for the program for_loop_no_enumerate_2.py given in Listing 7.15.

```
$ pylint for_loop_no_enumerate_2.py --disable=C0103,C0302,C0325,R0801,R0901
    ↪ ,R0902,R0903,R0911,R0912,R0913,R0914,R0915,R1702,R1728,W0212,W0238,
    ↪ W0703
-----------------------------------
Your code has been rated at 10.00/10
# pylint 4.1.3 succeeded with exit code 0.
```

Listing 7.20: The results of static code analysis using the Ruff linter for the program for_loop_no_enumerate_2.py given in Listing 7.15.

```
$ ruff check --target-version py312 --select=A,AIR,ANN,ASYNC,B,BLE,C,C4,COM
    ↪ ,D,DJ,DTZ,E,ERA,EXE,F,FA,FIX,FLY,FURB,G,I,ICN,INP,ISC,INT,LOG,N,NPY,
    ↪ PERF,PIE,PLC,PLE,PLR,PLW,PT,PYI,Q,RET,RSE,RUF,S,SIM,T,T10,TD,TID,TRY,
    ↪ UP,W,YTT --ignore=A005,ANN001,ANN002,ANN003,ANN204,ANN401,B008,B009,
    ↪ B010,C901,D203,D208,D212,D401,D407,D413,INP001,N801,PLC2801,PLR0904,
    ↪ PLR0911,PLR0912,PLR0913,PLR0914,PLR0915,PLR0916,PLR0917,PLR1702,
    ↪ PLR2004,PLR6301,PT011,PT012,PT013,PYI041,RUF100,S,T201,TRY003,UP035,W
    ↪  --line-length 79 for_loop_no_enumerate_2.py
SIM113 Use `enumerate()` for index variable `i` in `for` loop
  --> for_loop_no_enumerate_2.py:13:5
11 | for v in data:  # We iterate over the data values directly.
12 |     print(f"data[{i}]={v}")
13 |     i += 1      # And we increment the index by ourselves.
   |     ^^^^^^
Found 1 error.
# ruff 0.17.3 failed with exit code 1.
```

for_loop_no_enumerate_1.py and only used Ruff, at least at the time of this writing, we would not have discovered that we could use `enumerate` to solve the problem more efficiently. Vice versa, if our solution idea had been for_loop_no_enumerate_2.py and only used Pylint, then at the time of this writing, we, too, would not have arrived at the more efficient version . Thus, indeed, it is important to understand and use the tools that surround our programming language. And not just one. Keep learning and adding new tools to your tool belt.

Listing 7.21: Enumerating over the index-value pairs of a list using the `enumerate` function. (stored in file `for_loop_enumerate.py`; output in Listing 7.22)

```
"""
Enumerate the index-value pairs in a list using enumerate.

We now use the function `enumerate` to generate a sequence of
index-value pairs and directly unpack them in the loop header into the
variables `i` (for the index) and `v` (for the value).
"""

data: list[int] = [1, 2, 3, 5]   # The data to iterate over.

for i, v in enumerate(data):     # Generate index-value pair sequence...
    print(f"data[{i}]={v}")
```

↓ `python3 for_loop_enumerate.py` ↓

Listing 7.22: The stdout of the program `for_loop_enumerate.py` given in Listing 7.21.

```
data[0]=1
data[1]=2
data[2]=3
data[3]=5
```

7.6 The `while` Loop

Old clay tablets show that the Babylonians were able to approximate $\sqrt{2}$ maybe as far back as 4000 years ago [206, 497]. Our friend LIU Hui (刘徽), too, developed an algorithm for computing the square root of a number [507], which he probably used for approximating π (see Section 4.1.2). However, the mathematician Hero(n) of Alexandria lived in the first century CE, proposed a different method earlier on. He specified the algorithm for computing the square root of numbers which, today, is known as Heron's Method [326, 497]. This famous researcher, illustrated in Figure 7.2, was also a famous engineer who invented a steam engine [117] and a formula for the computation of the area of a triangle [334].

Let's say that we want to find the square root $\sqrt{a}$ of a given a. Then, this algorithm starts with a guess x_0, let's say $x_0 = 1$. In each iteration, it will compute a new guess x_{i+1} based on the current approximation x_i as follows [326, 497]:

$$x_{i+1} = \frac{1}{2}\left(x_i + \frac{a}{x_i}\right) \tag{7.1}$$

We can roughly imagine that the algorithm works as follows: If x_i was too big, i.e., $x_i > \sqrt{a}$, then $\frac{a}{x_i} < \sqrt{a} < x_i$. If x_i was too small, i.e., $x_i < \sqrt{a}$, then $\frac{a}{x_i} > \sqrt{a} > x_i$. By computing the average



Figure 7.2: Heron of Alexandria. Codex of Saint Gregory Nazianzenos. Greek manuscript of the ninth century CE. Public Domain. Source: [117].

of x_i and $\frac{a}{x_i}$ as the next guess, we hope to approach $\sqrt{a}$. If $x_i = \sqrt{a}$, then $\frac{a}{x_i} = \sqrt{a}$ by definition and $x_{i+1} = x_i$. Showing that this actually works and that the error gets smaller over time is more complicated [497]. But luckily, we do not need to do that here. Let's simply trust the two thousand year old genius Heron.

If we want to implement this algorithm, we will naturally need a loop of some sort. Clearly, we perform the same computation multiple times. We will evaluate Equation 7.1 again and again. However, a `for` loop will not do: We do not know in advance the number of steps that we will need until $x_i = x_{i+1}$. Of course, we could try to pick a very very huge number and then `break` the loop when the guesses converge ... but that is just ugly. The `while` construct comes to the rescue[404]:

```
"""The syntax of a while-loop in Python."""

while booleanExpression:
    loop body statement 1  # The loop body is executed as long as the
    loop body statement 2  # booleanExpression evaluates to True.
    # ...

normal statement 1  # After booleanExpression became False, the while
normal statement 2  # loop ends and the code after it is executed.
# ...
```

Like the `for` loop, the `while` loop has a head and a body. The head consists of a `while`, followed by a loop condition `booleanExpression` and a colon `:`. The loop body is again just a code block indented with four spaces. Every time *before* the loop body is executed, the loop condition `booleanExpression`

Listing 7.23: We compute the square root of a number using Heron's Method [326, 497] implemented as py `while` loop. (stored in file `while_loop_sqrt.py`; output in Listing 7.24)

```
"""Using a `while` loop to implement Heron's Method."""

from math import isclose  # Checks if two float numbers are similar.
from math import sqrt     # Compute the root as exactly as possible.


for number in [0.5, 2.0, 3.0]:  # The three numbers we want to test.
    # Apply Heron's method to get square root of `number`.
    guess: float = 1.0          # This will hold the current guess.
    old_guess: float = 0.0      # 0.0 is just a dummy value != guess.

    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess      # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # Compute the new guess.

    actual: float = sqrt(number)  # Use the `sqrt` function from `math`.
    print(f"\u221A{number}\u2248{guess}, sqrt({number})={actual}")


# We use `while not isclose(old_guess, guess)` instead of
# `while old_guess != guess` to avoid a strict comparison of floats:
# Looping until two floating point numbers become equal is very
# dangerous. It  may, in some cases, lead to endless loops. (Not in case
# of this algorithm, though, but let's be on the safe side and always
# follow best practices.)
```

↓ `python3 while_loop_sqrt.py` ↓

Listing 7.24: The stdout of the program `while_loop_sqrt.py` given in Listing 7.23.

```
√0.5≈0.7071067811865475, sqrt(0.5)=0.7071067811865476
√2.0≈1.414213562373095, sqrt(2.0)=1.4142135623730951
√3.0≈1.7320508075688772, sqrt(3.0)=1.7320508075688772
```

is evaluated. If and only if it yields `True`, the loop body is executed. Only in this case, the next iteration begins by again checking the loop condition, and so on. If the loop condition does not yield `True`, the loop is immediately terminated. In other words, the body of the loop is executed as long as a Boolean expression in the head of the loop evaluates to `True`.

We now use this new construct to implement Heron's Method as program `while_loop_sqrt.py` in Listing 7.23 and use it to compute the square roots of 0.5, 2, and 3.

We begin the program with an outer `for` loop that iterates a variable `number` over the `float` values `0.5`, `2.0`, and `3.0`. We want to apply the algorithm to each of these values. We use two variables `guess` and `old_guess`. `guess` will be the current guess of what $\sqrt{\texttt{number}}$ could be. `old_guess` will be the previous approximation, which we need to remember in order to decide when to stop.

We initialize `guess` with `1.0` and `old_guess` with a different value, say `0.0`. Our `while` loop should keep iterating as long as `guess != old_guess` and update the approximation of the square root in each step.

The loop condition here deserves some discussion, because it is both quite interesting and quite important. Let us think about it for a bit. First, it is clear that if we could actually represent the real numbers $\mathbb{R}$ at infinite precision, we would never reach `guess == old_guess` for any `number` with an irrational square root. Therefore, for irrational roots, our algorithm would never terminate. However, the `float` datatype has limited precision (see Section 3.3.1).

We work with limited precision and thus eventually reach a point where we won't be able to further improve the approximation precision of the square root. So looping while `guess != old_guess` should work and we should eventually reach a situation where `guess == old_guess`. But this very imperfection can come and bite us when comparing floating point numbers from an unexpected direction.

Best Practice 35

Due to the limited precision of floating point numbers, comparing the result of a floating point computation with another value using the strict `==` or `!=` operators is discouraged [26, 334]. It may lead to unanticipated results. For example `(0.1 + 0.2)== 0.3` gives `False`. Using functions like `isclose` from the `math` module that test whether two values are approximately the same based on their relative and absolute difference can be a (somewhat [227]) safer choice [26].

Now, if the direct comparison of floating point numbers for equality can cause us trouble in an `if`, it will probably not surprise you that it may cause even worse problems in a `while`:

Best Practice 36

As a corollary of Best Practice 35: Do not use strict equality or inequality comparisons of `floats` as loop termination criteria [334], as they quite likely lead to endless loops that never terminate. There can always be inputs that cause endless oscillations between values or the appearance of `nan` values (see Section 3.3.5). The former issue can again be made somewhat less likely by using methods like the function `isclose` from the `math` module that checks whether two numbers approximately equal based on their relative and absolute difference [26].

So we do not use `guess != old_guess` as loop criterion. We import the function `isclose` from the module `math`. Then we write `not isclose(guess, old_guess)` instead of `guess != old_guess`. The function `isclose` will consider `guess` and `old_guess` to be equal if their relative difference is less than one part in a billion [26]. Our loop will thus terminate once the current and the last guess are quite close to each other.

The inside of the loop is quite easy. First, the current guess is stored as the old guess via `old_guess = guess`. Then we update the guess as specified in Equation 7.1. We set `guess = 0.5 * (guess + number / guess)`. That's it. We have implemented Heron's method.

Finally, we print the result of the computation. For the sake of comparison, we also print the output of the `sqrt` function of the `math` module. As you can see, our algorithm delivers almost the same result. It works quite well, at very high precision. Also, notice how we used Unicode escape sequences from Section 3.6.6 to represent the characters $\sqrt{\cdot}$ and $\approx$ as `\u221A` and `\u2248` to get them neatly printed on the console.

7.7 The `else` Statement at the Bottom of Loops

Now, you have learned before that we can leave a loop body immediately by calling the `break` statement. Let's say that we want to perform a certain action $\mathcal{A}$ *after* the loop *if and only if* the loop has completed normally. In other words, we want to perform an action $\mathcal{A}$ if `break` was not invoked and the loop condition has led to a normal termination.[1]

We can do this by declaring a Boolean variable `ok` denoting whether the loop has completed normally before the loop and initializing it with `ok = True`. If we invoke `break`, then we would first set this variable `ok` to `False`. After the loop, we could place an `if` to check if the variable `ok` is still `True` and then execute the action $\mathcal{A}$.

This is totally fine, but `Python` offers us a much less verbose method: Using the `else` statement at the bottom of the loop. The body of the `else` will only be executed if the loop has terminated normally, i.e., based on its loop condition. This works for both `for` and `while` loops[404].

```
"""The syntax of a for-loop with else staement in Python."""

for loopVariable in sequence:
    loop body statement 1  # The loop body is executed for every item
    loop body statement 2  # in the sequence.
    # ...
else:
    else statement 1  # The body of the 'else' block is only executed if
    else statement 2  # 'break' was never invoked in the for loop body.
    # ...

normal statement 1  # After the sequence is exhausted, the code after
normal statement 2  # the for loop will be executed.
# ...
```

```
"""The syntax of a while-loop with else statement in Python."""

while booleanExpression:
    loop body statement 1  # The loop body is executed as long as the
    loop body statement 2  # booleanExpression evaluates to True.
    # ...
else:
    else statement 1  # The body of the 'else' block is only executed if
    else statement 2  # 'break' was never invoked in the while loop.
    # ...

normal statement 1  # After booleanExpression became False, the while
normal statement 2  # loop ends and the code after it is executed.
# ...
```

We now use this construct to implement a binary search [51, 247, 322]. Binary search is an algorithm that finds the index of an element in a *sorted* sequence `data` of values.

The core concept of binary search is that we consider a segment S of the list in which the element E we search may be contained. In each step of the algorithm, we want to reduce the size of this segment by ruling out the *half* in which E cannot be. We do this looking at the element M right in the middle of S. Now, the whole sequence `data` and, hence, also the segment S, are sorted. If M is bigger than E, then E can only be in the first, lower half, i.e., in the sub-segment from the start of S to right before M. If M is smaller than E, then E can only be in the second, upper half, i.e., in the sub-segment starting right after M and reaching until the end of S. Otherwise, i.e., if $E = M$, we have found the element. Of course, if we did not yet find E bit the selected half of S is empty because S only had one or two elements left in it ... then E is not in S and, hence, not in `data`.

[1] Of course, except `break`, there are also other things that can prevent the action from being performed, including the invocation of `return` or raising an `Exception` in the loop... ... but that's not the point here.

Listing 7.25: We implement binary search [51, 247, 322] using a `while` loop with a `break` statement. (stored in file `while_loop_search.py`; output in Listing 7.26)

```
"""Using a `while` loop to implement Binary Search."""

data: str = "abdfjlmoqsuvwyz"  # A string of sorted characters.

for search in ["a", "c", "o", "p", "w", "z"]:  # Search six characters.
    # Perform binary search to find `search` in `data`.
    upper: int = len(data)      # *Exclusive* upper index.
    lower: int = 0              # Lowest possible index = 0 (inclusive).
    while lower < upper:        # Repeat until lower >= upper.
        mid: int = (lower + upper) // 2  # Works ONLY in Python 3 :-).
        mid_str: str = data[mid]  # Get the character at index mid.
        if mid_str < search:    # If mid_str < search, then clearly...
            lower = mid + 1     # ...the index of search must be < mid.
        elif mid_str > search:  # If mid_str > search, then clearly...
            upper = mid         # ...the index of search must be > mid.
        else:  # If neither (mid_str < search) nor (mid_str > search)...
            print(f"Found {search!r} at index {mid} in {data!r}.")
            break  # Exit while loop and skip over while loop's else.
    else:  # executed if the while condition is False; not after break
        print(f"Did not find {search!r} in {data!r}.")
```

↓ `python3 while_loop_search.py` ↓

Listing 7.26: The stdout of the program `while_loop_search.py` given in Listing 7.25.

```
Found 'a' at index 0 in 'abdfjlmoqsuvwyz'.
Did not find 'c' in 'abdfjlmoqsuvwyz'.
Found 'o' at index 7 in 'abdfjlmoqsuvwyz'.
Did not find 'p' in 'abdfjlmoqsuvwyz'.
Found 'w' at index 12 in 'abdfjlmoqsuvwyz'.
Found 'z' at index 14 in 'abdfjlmoqsuvwyz'.
```

This means in one step we have effectively halved the size of S. If $n =$ `len(data)`, then we can do this halving at most $\log_2 n$ times and the time complexity of binary search is $\mathcal{O}(\log n)$ [51, 247, 322].

In Listing 7.25, we want to find the indices of some characters in the alphabetically sorted string `data = "abdfjlmoqsuvwyz"`. Of course, there is the string method `find` that can do this. This method would linearly search through the string. We, however, want to take advantage of the fact that the characters in `data` are sorted and use binary search.

We search for the six characters `"a"`, `"c"`, `"o"`, `"p"`, `"w"`, and `"z"`. Four of them are in `data`, but `"c"` and `"p"` are not. We search them anyway. We let a variable `search` iterate over the list `["a", "c", "o", "p", "w", "z"]` in an outer loop.

Now in the inner loop, we implement the binary search. This search will maintain and update two indices `lower` and `upper`. `lower` is the inclusive lower end of the segment S in which `search` could be contained. It is therefore initialized with `0`. `upper` is the exclusive upper end of the segment S in which `search` could be contained. We initialize it with `len(data)`; as it is *exclusive*, it will be 1 bigger than the largest valid index `len(data)- 1`. Our segment S is not empty, i.e., contains at least one element, as long as `lower < upper`. This is therefore the loop condition of the inner loop.

Inside the binary search loop, we first compute the mid index as `mid = (lower + upper)// 2`[2]. We obtain the value `mid_str` as the single character at that index via `mid_str = data[mid]`.

We know that if `mid_str < search`, then our character `search` cannot be located at any index in the range 0..`mid`. So in this case, we can update the *inclusive* index `lower` to become `mid + 1`. Otherwise, if `mid_str > search`, then we know that `search` could not possibly located anywhere in

[2]Interestingly, this works only because Python 3 has integers of infinite range (see Section 3.2). In programming languages like C or Java where integer types have limited ranges, we need to do `mid = lower + (upper - lower)// 2` [247].

the range `mid..len(data)- 1`. We thus would set the *exclusive* index `upper` to `mid`, which excludes all items starting at index `mid` from further consideration.

Now if neither `mid_str < search` nor `mid_str > search` were `True`, it must be that `mid_str == search`. This means that we found the location of `search` – it is at the index `mid`. Therefore, we print this result to the output. (Notice that the `!r` format specifiers in the f-string we use add the nice single quotes around `search` and `data`.) After printing the information, we exit the `while` loop using the `break` statement.

Now, there is the possibility that we cannot find `search` in `data` because it is simple in there. In this case, we will never print the output and also not leave the loop with `break`. In each iteration that does not end with `break`, we will either increase `lower` or decrease `upper`. Thus, eventually `lower < upper` will become `False`. This means that the loop terminates normally, because of its loop condition becoming `False`. Then and only then the body of `else` statement at the bottom of the loop is executed. Then and only then we print that we did not find the string `search`.

As you can see from the output in Listing 7.26, our binary search indeed works. It finds the strings that it was supposed to find. When an element was contained in the searched sequence, it correctly pointed that out. Wow. We now can implement some real fancy algorithms.

7.8 Summary

With this, we depart from the subject of loops. We have learned two ways to execute code iteratively: The `for` loop iterates over sequences of objects, which can either be `ranges` of numbers or arbitrary collections. The `while` loop permits us to specify an arbitrary Boolean expression as loop condition. In the bodies of both loops, we can jump to the next iteration at any time using the `continue` statement or we can exit the loops entirely using the `break` statement. Finally, placing an `else` statement at the bottom of the loop allows us to execute some code when the loop completes regularly, i.e., not via `break`.

We now have some nice tools in our hands. We can create code that branches and conditionally performs actions via `if`-`else`. And we can repeatedly perform actions via `for` and `while`. Our examples also have become more elaborate and interesting. We can now approximate π with arbitrarily many steps of the approach of LIU Hui (刘徽). We can implement Heron's Method to compute the square root of a number and we perform binary search over arbitarily large (sorted) data.

We can do quite a lot! These are *real* algorithms. This is no longer child's play. Slowly, we are moving towards some serious coding.

What we cannot yet do is to have a block of code that we want to re-use in *different* places.

Chapter 8

Functions

Functions are blocks of code that can be invoked from anywhere else in a program. You already learned many functions, from the basic `print` routine that just prints the value of its parameter to the `sqrt` function from the `math` module which computes the square root. Now you will learn how to make your own functions [404].

8.1 Defining and Calling Functions

We distinguish the definition and the invocation of a function. The function definition is where we specify the name, parameters, return value, and the body of code of a function. The function invocation is where we actually use the function. We can *invoke*, also known as *call*, from anywhere in our code by using its name. We then have to supply the values for its parameters and can receive the return value, e.g., in a variable. The syntax for defining our own functions in Python is as follows:

```
"""The syntax of function definitions and function calls."""

def my_function(param_1: type_hint, param_2: type_hint) -> result_type:
    """
    Short sentence describing the function.

    The title of the so-called docstring is a short sentence stating
    what the function does. It can be followed by several paragraphs of
    text describing it in more detail. Then follows the list of
    parameters, return values, and raised exceptions (if any).

    :param param_1: the description of the first parameter (if any)
    :param param_2: the description of the second parameter (if any)
    :returns: the description of the return value (unless `-> None`).
    """
    body of function 1
    body of function 2
    return result  # if result_type is not None we return something


normal statement 1  # some random code outside of the function
normal statement 2
my_function(argument_1, argument_2)  # We call the function like this.
```

A function in Python is created by using the `def` keyword, followed by the name of the function.

Best Practice 37

Function names should be lower case, with underscores separating multiple words if need be [594]. This style is called *snake case* [656].

After the function name follows an opening and a closing parenthesis, i.e., `(...)`. A function can have parameters through which we can pass values to it. In case of the `print` function, this was the string to be printed. In case of the `sqrt` function, this was the number to compute the square root of. Inside the function, these parameters act like variables. The values of these variables can be passed in when we call (invoke, execute) the function. The values thus stem from the code that calls the function.

Definition 8.1: Parameter

A function *parameter* is a variable defined inside the function that receives its value from the calling code when the function is called.

The parameters of a function are written between the opening and closing parentheses in the function header. Each parameter has a name, which is the name under which it can be accessed inside the function. The parameters are separated by commas. We can specify arbitrarily many parameters or no parameter at all. Notice that, just like variables, all such parameters should be annotated with type hints (see Section 4.4) [638]. This is especially important: You do want that users of your function can clearly understand whether they can pass integers, strings, or floating point numbers as parameter values to your function or not. If we define a function `add(value_1, value_2)`, it would not be clear at all what datatypes we can pass in for `value_1` and `value_2`. With type hints like `add(value_1: int, value_2: int)`, it will be very clear.

Functions can return results (like the `sqrt` function of the `math` module does) or return nothing (like `print`). If they return a result, the type of this result should be specified. This is done with the type hint `-> result_type` following the closing parenthesis in the function header.

Definition 8.2: Signature

The sequence of parameter types and the return type of a function together are called the signature of the function.

The function header ends with a colon (`:`).

Best Practice 38

All parameters and the return value of a function should be annotated with type hints [638]. From my perspective: *A function without type hints is wrong.*

After the function header, indented by four spaces, follows the function body.

Best Practice 39

The body of a function is indented with four spaces.

The function body can be an arbitrary block of code, which may contain all the things we already learned. From `if...else` over `for` and `while` loops to variable assignments and calls to other functions, all can be used.

When a function is called, this code block is executed. When the end of the block is reached, the function automatically exits without returning any value. At that point, the control flow resumes in the code that called the function, right after the function call. The function can also be exited at any point by using the `return` statement. If the function is supposed to return a value `result`, then this is done via `return result`. Notice that, like the `break` statement in loops, we can place `return` at any location we want. We can also have multiple `return` values at different places in the function.

The function `my_function` then can be called from anywhere in the code by writing `my_function(value_1, value_2)`. Here, `value_1` is passed in as value of `param_1` and `value_2` is passed in as value of `param_2`. This follows the same pattern of function calls that we already used in many of our examples.

Definition 8.3: Argument

An *argument* is the actual value given for a function parameter when the function is called.

Between the header of a function and its body, we always place a so-called docstring, which is a multi-line string (see Section 3.6.5).[1] This string consists of a title line shortly describing what the function does. Then follows an empty line. After that, we can (but not necessarily need to) place paragraphs of text providing a more detailed discussion. Then follows the list of parameters, each in the syntax `:param parameter_name: description`. Then follows the return value description (if the function returns something) in the form `:returns: description`.

Best Practice 40

Each function should be documented with a docstring. If you work in a team or intend to place your code in public repositories like on GitHub, then this very very much increases the chance that your code will be used correctly. From my perspective: *A function without docstring is wrong.*

Best Practice 41

After the function and its body are defined, leave *two* blank lines before writing the next code [594].

Now we know the general scheme based on which functions can be defined. After all of this long introduction, let us finally come to an example. Let's implement the factorial function as, well, function. The factorial is defined as follows [106, 175]

$$a! = \begin{cases} 1 & \text{if } a = 0 \\ \prod_{i=1}^{a} i & \text{otherwise, i.e., if } a > 0 \end{cases} \tag{8.1}$$

where $\prod_{i=1}^{a} i$ stands for the product $1*2*3*\cdots*(a-1)*a$. We will implement this function in Python call it `factorial` in Listing 8.1. So we begin the header of our function a `def factorial(`. It should receive a single parameter `a`, so we write it into the parentheses. `a` will be type-hinted as integer via `a: int`. The result of our function will be an integer as well, so we add the type hint `-> int`. The header of our function ends with the colon `:` and is thus `def factorial(a: int)-> int:`.

The body of this function is straightforward. We begin by initializing a variable `product` with the value `1`. Then, we need a loop that iterates a variable i over all positive integers less than or equal to `a`. In each step, we want to multiply the value `i` with `product` to get the next value of `product`. Well, we can skip over `i` $= 1$, because that would be useless. So we will use a `for` loop iterating `i` over the `range(2, a + 1)`. This effectively starts `i` at `2`. Since the upper limit `a + 1` of the `range` is always *exclusive*, the last value for `i` will be `a`. Notice that we really use `a` like a normal variable that was assigned a value.

Anyway, inside the loop body, we compute `product *= i`, which is equivalent to `product = product * i`. After the loop, `product` holds `a`!. So we can return it as the result of the function, by writing `return product`.

We can now compute the factorial of any positive integer number `x` by calling `factorial(x)`. After the function body, we leave two empty lines. And then we compute the factorials of the numbers from 1 to 9 in a `for` loop and print them by using f-strings. Inside this loop and in the f-string, we can use the function `factorial` exactly like any other function we used before, like `sqrt` or `sin`. It may be an interesting side information at the end of this example that the factorial can actually be computed *faster* than using this product form, see, e.g. [370].

Functions can have more than one parameter or no parameter at all. They can return one value or return nothing at all[2]. Functions can also be called from other functions.

[1] Technically, you are not strictly required to use docstring. However, let's stick with Best Practice 40. Let's do things *right*.

[2] We already learned that if they return nothing, they actually return `None` back in Section 3.7.

Listing 8.1: Implementing a function computing the factorial of a positive integer number. (stored in file def_factorial.py; output in Listing 8.2)

```
"""Implementing the factorial as a function."""


def factorial(a: int) -> int:  # 1 `int` parameter and `int` result
    """
    Compute the factorial of a positive integer `a`.

    :param a: the number to compute the factorial of
    :return: the factorial of `a`, i.e., `a!`.
    """
    product: int = 1  # Initialize `product` as `1`.
    for i in range(2, a + 1):  # `i` goes from `2` to `a`.
        product *= i  # Multiply `i` to the product.
    return product  # Return the product, which now is the factorial.


for j in range(10):  # Test the `factorial` function for `i` in 0..9`.
    print(f"The factorial of {j} is {factorial(j)}.")
```

↓ `python3 def_factorial.py` ↓

Listing 8.2: The stdout of the program def_factorial.py given in Listing 8.1.

```
The factorial of 0 is 1.
The factorial of 1 is 1.
The factorial of 2 is 2.
The factorial of 3 is 6.
The factorial of 4 is 24.
The factorial of 5 is 120.
The factorial of 6 is 720.
The factorial of 7 is 5040.
The factorial of 8 is 40320.
The factorial of 9 is 362880.
```

Let us investigate these options by investigating another interesting mathematical operation: The computation of the greatest common divisor, also known as `gcd`. This can be done using the Euclidean algorithm [76, 186, 211], going back to Euclid of Alexandria ($Εὐκλείδης$) who flourished about 300 before Common Era (BCE) and is sketched in Figure 8.1.

The greatest common divisor of two numbers positive $a \in \mathbb{N}_1$ and $b \in \mathbb{N}_1$ is the greatest number $g \in \mathbb{N}_1 = \texttt{gcd}(a, b)$ such that $a \bmod g = 0$ and $b \bmod g = 0$, where $\bmod$ is the modulo division operator, i.e., the remainder of a division, which, in turn, is equivalent to Python's `%`. This means that g divides both a and b without remainder. If $a = b$, then obviously $\texttt{gcd}(a, b) = a = b$ as well. Otherwise, we know that $a = ig$ must be true for some $i \in \mathbb{N}_1$ and $b = jg$ will hold for some $j \in \mathbb{N}_1$.

Let us assume, without loss of generality, that $a > b$. Then, $c = a - b = (i - j)g$. It is clear that $c \bmod g = (a - b) \bmod g = (i - j)g \bmod g = 0$ must be true, too. It also holds, obviously, that $a - b < a$. Similarly, we could use the division remainder d instead of the difference c: $d = a \bmod b = ig \bmod (jg) = ig - \lfloor i/j \rfloor * jg = g(i - j\lfloor i/j \rfloor)$ is still divisible by g without remainder as $d \bmod g = 0$. And it is also true that $a \bmod b < a$.

Since both d and c are less than a *and* divisible by g, we could replace a with either of them. The cool thing about d is that d will be less than both a and b. This means that we could replace the old a with b and store d in the variable b.

If we keep repeating these computational steps, then our values will become smaller and smaller. But they will always be both divisible by g. We would repeat this until reaching $b = 0$, at which point a will be g. Matter of fact, by choosing the module-based update, we do not even need to assume that $a > b$. Because if $b > a$, then $a \bmod b = a$ and we would just switch a and b in the first step. If $a = b$, then $a \bmod b = 0$ and we would immediately terminate after the first step and return a as

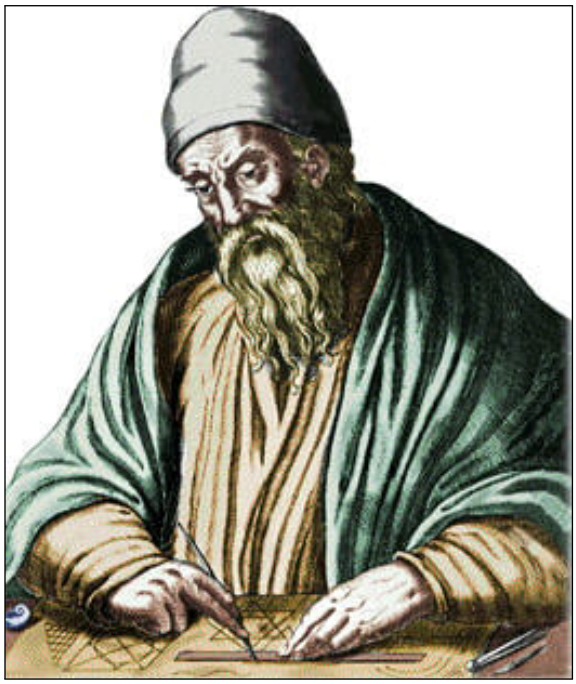


Figure 8.1: An illustration of Euclid of Alexandria, attributed to the painter Charles Paul Landon (1760–1826). Source: Vikidia, where it is noted as *domaine public,* i.e., as in the Public Domain.

the greatest common divisor.

This algorithm is implemented in file `def_gcd.py` given in Listing 8.3 as function `gcd`. Our new function `gcd` has two integer parameters, `a` and `b`. It returns another `int`. Notice the proper type hints in the function header. After the function header, we place a docstring describing the function, its parameters, and its return value.

The function body is surprising short: We use a `while` loop that iterates as long as `b > 0`. After the loop, we `return a` as the result. If `b == 0` holds at the beginning, the loop will never be executed and `a` is returned as-is, which is correct: `gcd`$(a, 0) = a$ for all $a \in \mathbb{N}_1$.

However, if `b > 0`, we enter the loop's body, which is a single line of code: the multi-assignment (or tuple-unpacking) command `a, b = b, a %b`. It works approximately like this: This line first completely evaluates the right-hand side. This creates a tuple where the first value is `b`. The second value is `a % b`. The tuple is then unpacked and stored in the variables `a` and `b`. `a` will thus receive the value that `b` had during the evaluation of the right-hand side. `b` will receive the previously computed value of `a % b`. In other words, `b` is stored in `a` and the remainder of the division of the old `a` by the old `b` is stored in `b`. Clearly, `b` will become smaller in each iteration. Since it can never become negative, it will eventually reach `0`. Then the loop will terminate. Similarly, the *gcd* is never "lost" during the loop. It will be the value in `a` at the end. And this value is returned.

So with `gcd`, we implemented a function with two parameters and one return value. Let us now implement a second function, this time with no return value. Our function `print_gcd` accepts again two parameters `a` and `b`. However, it returns nothing. Instead, it will print the `gcd` nicely using `print` and an f-string. Of course, we properly annotate it with type hints and also give it a proper docstring.

The `math` module also provides a function names `gcd`. It, too, computes the greatest common divisor. Naturally, we want to compare the result of our function with this one.

Of course, we cannot have two functions named `gcd` in the same context. So we import the function from the `math` module *under a different name*: `from math import gcd as math_gcd` makes the `gcd` function from the module `math` available under the name `math_gcd`. And we use it in the f-string in `print_gcd` under that name.

Finally, we confirm that `gcd` and `math_gcd` compute the same result for four test cases at the bottom of our program. Now that all is said and done, it should be mentioned that the Euclidean Algorithm has a particularly efficient binary variant which is faster than our implementation in Listing 8.3. This binary variant may have been developed in China in the first century CE [76] and published in the

Listing 8.3: Implementing the Euclidean Algorithm as a function and calling it from another function. (stored in file def_gcd.py; output in Listing 8.4)

```
"""Euclidian Algorithm for the Greatest Common Divisor as a function."""

from math import gcd as math_gcd  # Use math's gcd under name `math_gcd`


def gcd(a: int, b: int) -> int:  # 2 `int` parameters and `int` result
    """
    Compute the greatest common divisor of two numbers `a` and `b`.

    :param a: the first number
    :param b: the second number
    :return: the greatest common divisor of `a` and `b`
    """
    while b != 0:  # Repeat in a loop until `b == 0`.
        a, b = b, a % b  # the same as `t = b`; `b = a % b`; `b = t`.
    return a  # If `b` becomes `0`, then the gcd is in `a`.


def print_gcd(a: int, b: int) -> None:  # `-> None` == returns nothing
    """
    Print the result of the gcd of `a` and `b`.

    :param a: the first number
    :param b: the second number
    """
    print(f"gcd({a}, {b})={gcd(a, b)}, math_gcd={math_gcd(a, b)}.")
    # Notice: no `return` statement. Because we return nothing.


print_gcd(1, 0)
print_gcd(0, 1)
print_gcd(765, 273)
print_gcd(24359573700, 35943207300)
```

↓ `python3 def_gcd.py` ↓

Listing 8.4: The stdout of the program def_gcd.py given in Listing 8.3.

```
gcd(1, 0)=1, math_gcd=1.
gcd(0, 1)=1, math_gcd=1.
gcd(765, 273)=3, math_gcd=3.
gcd(24359573700, 35943207300)=2148300, math_gcd=2148300.
```

famous *Jiu Zhang Suanshu* (九章算术) [140, 144, 311, 420, 540].

We are now able to implement our own functions. We can properly annotate them with type hints and docstrings. We can thus define re-usable blocks of code that can be called from multiple, arbitrary places in other code. And we can make them available as units that can be used by others who understand their documentation. Nice.

8.2 Functions in Modules

You may not have noticed it, but we just made a very big step in our programming skills. We moved from simple programs which only consist of one big block of code to modular programs. We can now reuse code. When we began our journey, we typed all commands into the Python interpreter and executed them one by one. Then we became able to write our code into files, which allowed us to execute the same program several times. Now we can even structure the code into functions, which we can explain via docstrings and annotate with type hints. We can already write useful and reasonably

Listing 8.5: The module `my_math`, which provides two mathematics functions, namely `sqrt`, implementing the algorithm of Heron to compute the square root from Listing 7.23, and `factorial`, copied from Listing 8.1. (src)

```
"""A module with mathematics routines."""

from math import isclose  # Checks if two float numbers are similar.


def factorial(a: int) -> int:  # 1 `int` parameter and `int` result
    """
    Compute the factorial of a positive integer `a`.

    :param a: the number to compute the factorial of
    :return: the factorial of `a`, i.e., `a!`.
    """
    product: int = 1  # Initialize `product` as `1`.
    for i in range(2, a + 1):  # `i` goes from `2` to `a`.
        product *= i  # Multiply `i` to the product.
    return product  # Return the product, which now is the factorial.


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v` is approximately `number`.
    """
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess    # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

large programs However, so far, all of our programs in their entirety are stored in single files.

This puts a certain limit on the complexity of the applications that we can realize. After a few thousand lines of code in a single file and maybe a few dozens of functions, it will become very hard to keep track of what is what and where. This limit can easily be broken if we can group different types of functionality into different files.

We will therefore now learn how to distribute code over multiple files. This answers two main questions: How can we avoid writing our applications as a single, huge, and unstructured file which would be impossible to maintain in the long run? How can we divide our application into smaller units that we can test, improve, and maintain separately and maybe use and reuse in different contexts? A big part of the answer to this question are *modules* and *packages* [555].

For all intents and purposes within this book, a *module* is a Python file and a *package* is a directory wherein such files are located. As described in [555], modules do not necessarily need to be files and packages can probably be created otherwise as well, but let us keep it simple here.

Indeed, we have already worked with modules, most prominently the `math` module. This module is basically a collection of mathematical functions. Since we have implemented several mathematical functions by ourselves, let us put some of them in a module as well.

In Listing 8.5 we do just that. We create the Python file `my_math.py` and place two functions into it: The function `factorial` from Listing 8.1 and a new function called `sqrt`. The `sqrt` function basically encapsulates our code from back in Listing 7.23, where we implemented the Heron's Method to compute the square root, as a function. Now, `number`, the input of this algorithm, comes in as a parameter.

Listing 8.6: A program using the functions `sqrt` and `factorial` from the module `my_math` given in Listing 8.5. (stored in file use_my_math.py; output in Listing 8.7)

```
"""Using the mathematics module."""

from my_math import factorial, sqrt  # Import our two functions.

print(f"6!={factorial(6)}")  # Use the `factorial` function.
print(f"\u221A3={sqrt(3.0)}")  # Use the `sqrt` function.
print(f"\u221A(6!*1.5)={sqrt(factorial(6) * 1.5)}")  # Use both.

print("We now use Liu Hui's Method to Approximate \u03c0.")
e: int = 6  # the number of edges: We start with a hexagon, i.e., e=6.
s: float = 1.0  # the side length: Initially 1, i.e., radius is also 1.
for _ in range(6):
    print(f"{e} edges, side length={s} give us \u03c0\u2248{e * s /2}.")
    e *= 2  # We double the number of edges...
    s = sqrt(2 - sqrt(4 - (s ** 2)))  # ...and recompute the side length
```

↓ `python3 use_my_math.py` ↓

Listing 8.7: The stdout of the program use_my_math.py given in Listing 8.6.

```
6!=720
√3=1.7320508075688772
√(6!*1.5)=32.863353450309965
We now use Liu Hui's Method to Approximate π.
6 edges, side length=1.0 give us π≈3.0.
12 edges, side length=0.5176380902050417 give us π≈3.10582854123025.
24 edges, side length=0.2610523844401035 give us π≈3.1326286132812418.
48 edges, side length=0.13080625846028637 give us π≈3.139350203046873.
96 edges, side length=0.0654381656435527 give us π≈3.14103195089053.
192 edges, side length=0.03272346325297234 give us π≈3.1414524722853443.
```

Best Practice 42

Package and module names should be short and lowercase. Underscores can be used to improve readability. [594]

Our new module has the name `my_math`, because it is in file `my_math.py`. It does not look very special or different from what we did so far. The one difference that we notice, however, is that it "does nothing". In the file, we define two functions, but we do not actively call them, we do not use them for anything. This is the purpose of this module: It just provides the functions. We will use them elsewhere.

And Listing 8.6 is where we use them: We write a program, i.e., another Python file, named use_my_math.py. In this file, we want to use our two functions `factorial` and `sqrt` from the module `my_math`. For this purpose, we have to tell the Python interpreter where it can find these two functions. We do this by writing `from my_math import factorial, sqrt`. The meaning of the line is quite obvious: There is a module `my_math` `from` which we want to `import`, i.e., make available, two functions, namely `factorial` and `sqrt`.

Now, the Python interpreter knows a lot of modules. Several modules ship with any Python installation, like `math`. Others are installed via a package manager like pip [286] (we will eventually discuss this in-depth later). The `my_math` module is found because it is in the same directory as the program use_my_math.py.

We could have placed the my_math.py file into a sub-directory named `math_pack` instead. Then we would import our functions from `math_pack.my_math`, where `math_pack` would be called a *package*. Of course, we could also create another level of directories, say we could have directory `utils`, containing directory `math_pack`, containing our file my_math.py. In this case, we would import our functions like `from utils.math_pack.my_math import`.... The names of package and module are separated by

a `.` when importing from them. This allows us to nicely and hierarchically structure our projects into modules and packages for different purposes.

In Listing 8.6 we can use both `sqrt` and `factorial` exactly as if we had defined them in this program. We first print a few values for `sqrt` and `factorial` and also show that we can compute the result of the square root of a factorial. We also just copy the code from Listing 7.3, where we use the method of LIU Hui (刘徽) method to approximate π – but this time, we use our own implementation of the square root function instead of the one from the `math` module. Interestingly, the sixth and last approximation step in Listing 8.7 shows exactly the same result as in Listing 7.4.

Ever since we became able to use `while` loops, we had the basic means to implement *any* function that our computers can compute. However, we were limited on a technical level. Theoretically, we could have implemented an arbitrarily complex program. However, without functions, we would probably have very repetitive (and therefore, long) code. Without modules, we would have ended up with one giant, unreadable, unmaintainable file.

We now broke through these limitations. We can create reusable code by writing functions. We can group functions with similar tasks into modules. We even can place modules with similar domains into the same package and such with another domain into another one. We are now able to implement well-structured and maintainable software.

8.3 Interlude: Unit Testing

Structuring our code into functions and modules has several advantages. We can reuse code and we can divide big application into smaller pieces, both of which make it easier to understand what our program is doing. Or is *supposed* to be doing. Because errors happen in programming. They happen often and they happen naturally. And bigger programs are more likely to contain errors, and more likely to contain more errors than smaller programs.

We are now at a stage where we can design almost arbitrarily large programs, which are well-structured and maintainable. We are not limited anymore at the architectural level. However, our goal is not to just design large maintainable software ... we also want it to be correct and reliable.

Imagine that you have a big application consisting of unstructured code for reading and writing files, code for making mathematical computations, and so on. Then, it would not be easy to figure out how to check whether the big application behaves exactly as it is supposed to. There would be too many combinations of inputs and environments and conditions that we would need to test. And even if we would find that the application behaves incorrectly in a few cases ... then what? It would be very hard to figure out which of its many interacting lines of code causes the error.

Here, dividing code into functions and placing it into modules has yet another advantage: We now have smaller units of code – the functions – that we can test separately. It is much easier to check whether our `factorial` and `sqrt` functions behave as we expect them than to do this for a whole complex program.

Definition 8.4: Unit Testing

Unit Testing is a software testing technique where separate components or functions of an application are tested in isolation [44, 424, 426, 430, 486, 568].

This idea works hierarchically. We implement and test the smallest functions $\mathcal{A}$. We "code a little, test a little" [45], allowing us to both understand and verify our code. Then we move on. We implement and test the functions $\mathcal{B}$ that use these smaller functions $\mathcal{A}$. We implement and test the functions $\mathcal{C}$ that use the functions $\mathcal{B}$ (and maybe also $\mathcal{A}$). We work our way up, from implementing and testing the smallest, most primitive pieces of code at the lowest level of our application, to the most abstract code at the top-level of our application. We only go to the next level after the tests at the lower levels all pass. If we then find that a test at a higher level fails, then we would first search for an error in the higher-level code. Of course, it could also be that the tests at the lower level are either not sufficient and did not spot an error there that they themselves are wrong. However, we do know a most likely source of an error. This can tremendously speed up the process of locating and fixing bugs.

Testing also can very much increase our confidence in what we do. We have not just written down this code. We have tested it. And we tested the code that uses this code. And all tests pass. Surely,

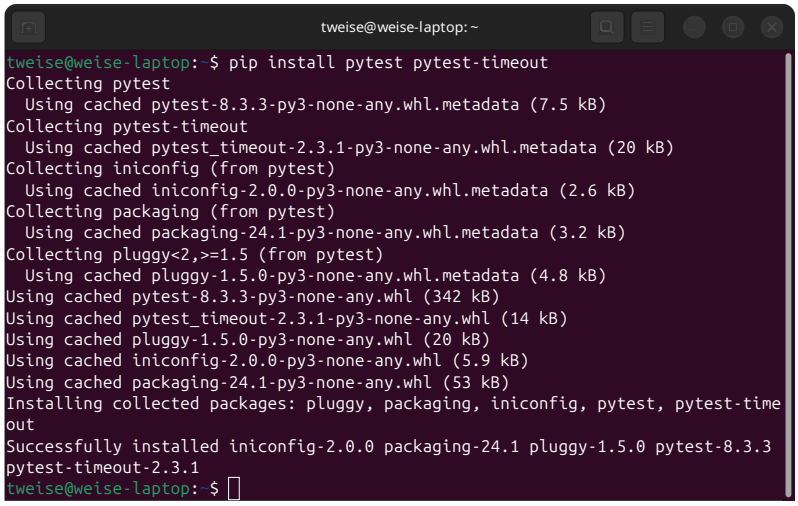


Figure 8.2: Installing `pytest` in a `Ubuntu` terminal via `pip` (see Section 14.1 for a discussion of how packages can be installed).

there still might be undetected errors. But we did cover the likely use cases with proper tests and they did work. Our work is not just some fiddly botch, it is a piece of sound engineering.

But how does this work on a technical level? It's easy. A unit test is basically a separate program. This program loads the module and function it is supposed to test. It invokes the function and compares its behavior to the expected behavior in a given scenario. If it matches, the test passes. If it differs, then the test fails. Here, a "scenario" can be specific parameter values (arguments) and the "behavior" can be the return value of a function, for example. We know which return value we expect for the given input and we compare the actual return value with that. We can design arbitrarily many such scenarios (test cases) for testing one function. The tests are part of the software developement and the documentation.

The tests are normally not part of the final product (the program) that ships to the customer or end users. They have no real function beyond testing the actual and documenting the expected behavior of components.

Actually, we even *almost* did this already. Remember how we compared the result of our self-implemented `sqrt` function with the `sqrt` function provided by the module `math`? We printed the values of both functions side-by-side. We could also have compared programmatically. We could have reported "Test Success!" if `isclose` for them yields `True` and a "Test Error!" otherwise.

The goal of using unit tests is to ensure that each unit of the software performs as expected. Since tests can be developed alongside (or even before!) the single functions are implemented, potential errors can be found early in the development process. As the unit tests focus on smaller, well, units, they can be less complex and easier to understand. The consequent usage of unit tests supports a modular and cleaner programming style, as it forces the developer to divide bigger programs into smaller pieces that can be invoked and tested in separation.

Finally, unit tests are especially useful if an application is developed, improved, and maintained over a long time: It is important that test cases are preserved and re-tested every time the software changes. This way, we can discover if a change that we applied to an older piece of code breaks a unit test. In other words, we can detect if a change on some module has unanticipated consequences on other code, which may lead to unexpected and unwanted changes of the behavior our program. Especially with its increased automation, e.g., due to CI, unit tests in software development have steadily gained importance during the past decades [486, 568, 633] and are an important cornerstone of `Python` software development [154, 425, 430].Let us now test our code. We will use `pytest` for this, a `Python` framework for unit testing [329]. We install it by opening the terminal by either pressing Ctrl + Alt + T under `Ubuntu Linux` or Ctrl + Alt + T or under Microsoft `Windows` via press ⊞ + R, type in `cmd`, and hit ↵. As sketched in Figure 8.2, we then type in `pip install pytest pytest-timeout` and hit ↵. (Normally, we would do that in a virtual environment, which we discuss later in Section 14.1.) It is important that we install *two* `Python` packages here: The package `pytest` offers the basic testing functionality. The package `pytest-timeout` allows us to limit the runtime of tests, which is a very

Listing 8.8: A small unit test suite for Listing 8.5. (src)

```
"""Testing our mathematical functions."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math import factorial, sqrt  # Import our two functions.


def test_factorial() -> None:
    """Test the function `factorial` from module `my_math`."""
    assert factorial(0) == 1  # 0! == 1
    assert factorial(1) == 1  # 1! == 1
    assert factorial(2) == 2  # 2! == 2
    assert factorial(3) == 6  # 3! == 6
    assert factorial(12) == 479_001_600  # 12! == 479'001'600
    assert factorial(30) == 265_252_859_812_191_058_636_308_480_000_000


def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

important feature (as you will see).

Useful Tool 7

pytest is a Python framework for writing and executing software tests [329]. It can be installed via `pip install pytest pytest-timeout` as shown in Figure 8.2 on page 170. You can then apply pytest using the command `pytest --timeout=toInS file(s)`, where `toInS` should be replaced with a reasonable timeout in seconds and `file(s)` is one or multiple files with test cases. We provide a script for using pytest with a reasonable default configuration in Listing 16.4 on page 442. See also Useful Tool 9 later on.

You will ask yourself how testing and especially the reuse of test cases works. It is, actually, fairly simple: We have our actual program code in one or multiple Python files. We have our test code in some other Python files. For the sake of clarity, if the actual code is in a file my_math.py, then we could put the code for the tests into a file called test_my_math.py.

In Listing 8.8 we do just that. We create a file test_my_math.py and into this file, we want to put the code for testing our module `my_math`. This testing code is defined in form of functions. The names of these functions must start with `test_`.

Now our module `my_math` provides two functions, `factorial` and `sqrt`. At the top of our new tests module `test_my_math`, we `import` both of these functions. Naturally, we would create corresponding test functions and call them `test_factorial` and `test_sqrt`. These functions have no parameters and no return values.

Tests are often defined in the form of several assertions assertions:

```
"""The␣syntax␣of␣an␣assert␣statement␣in␣Python."""

assert␣booleanExpr␣␣#␣raises␣AssertionError␣if␣not␣booleanExpr
```

Listing 8.9: The output of the unit tests in Listing 8.8: While the test of `factorial` succeeds, our `sqrt` function fails for input `0.0`.

```
$ pytest --timeout=10 --no-header --tb=short test_my_math.py
========================== test session starts ==========================
collected 2 items
test_my_math.py .F                                                  [100%]
============================== FAILURES ==============================
_____________________________ test_sqrt _____________________________
test_my_math.py:20: in test_sqrt
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
           ^^^^^^^^
my_math.py:30: in sqrt
    guess = 0.5 * (guess + number / guess)  # The new guess.
                           ^^^^^^^^^^^^^^
E   ZeroDivisionError: division by zero
======================== short test summary info ========================
FAILED test_my_math.py::test_sqrt - ZeroDivisionError: division by zero
====================== 1 failed, 1 passed in 0.02s ======================
# pytest 9.2.0 with pytest-timeout 2.4.0 failed with exit code 1.
```

An assertion is defined by the keyword `assert` followed by an arbitrary Boolean expression. If the Boolean expression evaluates to `True`, then nothing happens. If it evaluates to `False`, then an `AssertionError` is raised. This error will cause the test function to fail and terminate immediately. We will learn in Chapter 9 what `Exceptions` are and how to handle them in detail. If the complete test function runs through without raising any `Exception`, then the test has succeeded. If it raises an `Exception` at any point during its execution, the test has failed.

Now we create the function `test_factorial` to test the `factorial` function. For example, we know that $0! = 1$ and therefore it must be that `factorial(0)== 1`. So it makes sense to define `assert factorial(0)== 1`. The Boolean expression here is `factorial(0)== 1`, which is obviously only `True` if `factorial(0)` evaluates to `1`. We now manually compute some other factorials. We thus can define similar cases for 1!, 2!, and 3!.

Obviously, we cannot write down a complete list of all possible inputs. But these are the four smallest ones for which the factorial is defined. Therefore, we test them. By testing also 12!, we have checked `factorial` for a mid-sized argument. We then also create a test case for a fairly large number, say 30!. It is *very* ***very*** important that we did *not* use our `factorial` function to compute the expected return values ... otherwise the whole test would make no sense at all. Instead, we used some other tool. Here, we have computed this value using an online calculator[3] that offers arbitrary precision. The result of 30! is more than $265 * 10^{30}$, which is fairly beyond the range of 64 bit integers (and thus showcases that integers in Python 3 have an unlimited range, as stated back in Section 3.2). This test case validates whether our `factorial` function works well for large numbers, too.

If our `factorial` function passes all of these tests, we can be fairly certain that it is implemented correctly. There could still be errors, though. For example, we did not test 4!. But at least it looks unlikely that 3! and 12! "work," but not 4!.

For the `sqrt` function, we create similar test function and call it `test_sqrt`. Reasonable test cases are `assert sqrt(0.0)== 0.0`, `assert sqrt(1.0)== 1.0`, and `assert sqrt(4.0)== 2.0`. We would also expect that `sqrt(x)* sqrt(x)== x` for different `x`. However, we have to account for the limited precision of the datatype `float` discussed in Section 3.3.1. Even if we would get as close to $\sqrt{x}$ for some `x` as possible, we can only represent 15 to 16 digits. Therefore, we have to give a little bit of wiggle room when we compute `s3 = sqrt(3.0)` and hope that `s3 * s3 == 3.0`. We do this by writing `assert abs(s3 * s3 - 3.0)<= 5e-16`, i.e., by assuming that the difference between `3.0` and `s3 * s3` is not bigger than the very small number $5 * 10^{-16}$. On the other hand, the square root of `1e10 * 1e10` should be representable exactly, namely as `1e10`.

The datatype `float` also offers us two special values, `inf` and `nan`, which both can be imported from the `math` module and which were discussed in Section 3.3.5: `inf` stands for "too large to represent as `float`" and is often interpreted as positive infinity $+\infty$. `nan` stands basically for "undefined", a value

[3] https://www.numberempire.com

that you may get if you try to compute something like `inf - inf`.

Our `sqrt` function has to understand and correctly handle these values as well. `sqrt(inf)` should again return `inf`. `sqrt(nan)` should return `nan`. However, we cannot do `assert sqrt(nan)== nan`, since `==` will yield `False` if at least one `nan` is involved (see again Section 3.3.5). For testing whether `sqrt(nan)` yield `nan`, we use the function `isnan` from the `math` module. This function returns `True` for `isnan(nan)` and `False` otherwise.

With this, we have covered most reasonable inputs that either `factorial` or `sqrt` could receive. We have defined what we would expect as output for these inputs. These expectations are implemented as test cases. Regardless of how `factorial` or `sqrt` are implemented, they should pass these test cases. Otherwise, they are wrong.

We now execute our tests. We therefore open a terminal and type in the command `pytest --timeout=10 --no-header --tb=short test_my_math.py`. The first part, `pytest`, invokes `pytest`. `--timeout=10` defines a time out of ten seconds. If test runs longer than ten seconds, it will be aborted. In this case, it is considered as failed. `--no-header` and `--tb=short` are just there to tell pytest to produce shorter, more succinct output. I need them because otherwise, the listings with the output do not fit on a page. You will normally not use these parameters. Finally, `test_my_math.py` is the name of the file with the tests that we want to run.

Running our test cases with pytest yields the output in Listing 8.9. This output tells us that `test_factorial` ran through without any issue. `test_sqrt` however failed with a `ZeroDivisionError` that was raised inside our `sqrt` function. This happened when we tried to compute `sqrt(0.0)`.

We have to go back to our `my_math` module in Listing 8.6 to find what is going wrong. We see that our initial guess for the square root is `1`. We set it via `guess: float = 1.0`. In each step, we then compute `guess = 0.5 * (guess + number / guess)`. If `number` is `0.0`, then this effectively means `guess = 0.5 * guess`. And because `number` remains unchanged at `0.0`, we repeat this step again and again. `guess` is halved again and again.

We have learned that the precision of `floats` is finite and that positive values smaller than `5e-324` will just become `0.0`. This will happen here too. `guess` will thus eventually become `0.0`. In the next iteration after that, this leads to us trying to divide `0.0 / 0.0`, causing the `ZeroDivisionError`.

In order to fix this problem, we introduce the check `if number <= 0.0` into our `sqrt` function in the new version `my_math_2.py` of our module given in Listing 8.10. If this new conditional evaluates

Listing 8.10: An improved variant of Listing 8.5 dealing with the failing test case `0.0` discovered in Listing 8.9. (src)

```
"""A second version of our module with mathematics routines."""

from math import isclose  # Checks if two float numbers are similar.

# factorial is omitted here for brevity


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v` is approximately `number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.

    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess    # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Listing 8.11: We use the same small unit test suite given in Listing 8.8 for `sqrt` in Listing 8.10. (src)

```
"""Testing our second version of the `my_math` module."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math_2 import sqrt  # Get our 2nd square root implementation.

# test_factorial() is omitted for brevity

def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math_2`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

Listing 8.12: The output of the unit tests in Listing 8.11: This time, we hit the timeout because of an endless loop!

```
$ pytest --timeout=10 --no-header --tb=short test_my_math_2.py
========================== test session starts ==========================
collected 1 item
test_my_math_2.py F                                                [100%]
================================ FAILURES ================================
_____________________________ test_sqrt _____________________________
test_my_math_2.py:17: in test_sqrt
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
           ^^^^^^^^^
my_math_2.py:22: in sqrt
    guess = 0.5 * (guess + number / guess)  # The new guess.
    ^^^^^
E   Failed: Timeout (>10.0s) from pytest-timeout.
======================== short test summary info ========================
FAILED test_my_math_2.py::test_sqrt - Failed: Timeout (>10.0s) from pytest-
    ↪ timeout.
========================== 1 failed in 10.02s ==========================
# pytest 9.2.0 with pytest-timeout 2.4.0 failed with exit code 1.
```

to `True`, we return `0.0` directly. Now we do not consider the case that a negative number is passed into `sqrt` at this point. In that case, we would ideally `raise` an error by ourselves, but we will learn only later how to do that.[4]

We apply the same tests to the new version of the `sqrt` function as file `test_my_math_2.py` in Listing 8.11 to test our new module `my_math_2`. The output of the test case, provided in Listing 8.12, now indicates another error: We get a timeout!

We invoked `pytest` with the option `--timeout=10`, which only works if the package `pytest-timeout` is installed. This limited the maximum runtime of our test suite to ten seconds. Ten seconds is a reasonable time for *this book* as we actually run all the scripts automatically during the book building process. In practical situations, you will usually choose a larger time limit.

[4] We learn it in Chapter 9 and there we will revisit our `sqrt` function in Listing 9.1, too.

Listing 8.13: An improved variant of Listing 8.10 dealing with the failing test case `inf` discovered in Listing 8.12. (src)

```
"""A third version of our module with mathematics routines."""

from math import isclose    # Checks if two float numbers are similar.
from math import isfinite   # Checks whether a number is NOT nan or inf.

# factorial is omitted here for brevity


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v` is approximately `number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.
    if not isfinite(number):  # Fix for case `+inf` and `nan`:
        return number  # We return `inf` for `inf` and `nan` for `nan`.

    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Best Practice 43

Always attach a timeout to your unit tests. This timeout can be generous, maybe one hour, but it will serve as sentinel against either endless loops, deadlocks, or other congestion situations which all would be practical test failures. Timeouts protect automated builds or CI systems from clogging.

We can see from the output of pytest, that the timeout occurred when the argument `inf` was passed in as value for the parameter `number` of our `sqrt` function. What could have happened here? Again, we initially set `guess = 1.0`. In the first iteration of the loop in `sqrt`, we compute `guess = 0.5 * (guess + number / guess)`. This is the same as `guess = 0.5 * (1 + inf / 1)`, which is the same as `guess = 0.5 * inf`. This makes `guess` become `inf`, too. Then, in the second iteration, we again have `guess = 0.5 * (guess + number / guess)`. This now is the same as `guess = 0.5 * (inf + inf / inf)`. And `inf / inf` yields `nan` [227].

The `nan` infects the rest of the computation, and the result turns into `guess = nan`. From now on, all calculations yield `nan`. Now it holds that `nan != nan` and `isclose(nan, nan)` never becomes `True`. Therefore, our loop never ends. The time limit protected us here. Our `sqrt` function goes into an endless loop if `number = inf`. And our unit test would have run forever.

We solve this problem in the third version of our module, `my_math_3.py`, given in Listing 8.13. Here, we add a new condition: `if not isfinite(number)`, we return `number` as-is. `isfinite` is another function from the `math` module. It takes one parameter and it returns `True` if it is a finite number. `isfinite` returns `False` if its argument is `inf`, `-inf`, or `nan`. This means that condition can only become `True` in our function for `inf` or `nan`, as we already return `0.0` if `number <= 0.0`. Thus, we return `inf` if the input of our function is `inf`. And this is right. We also return `nan` if the input is `nan`. And this is right too. These are also the results that we would expect.

In Listing 8.14, we apply our unit tests to this new version of our `sqrt` function. As you can see in the output provided in Listing 8.15, the tests now complete successfully. This was a good example of how tests can help us to spot errors in our code. When looking at Listing 8.6, we certainly assumed

Listing 8.14: We use the same small unit test suite given in Listing 8.8 for `sqrt` in Listing 8.13. (src)

```
"""Testing our third version of the `my_math` module."""

from math import inf, isnan, nan  # some float value-checking functions

from my_math_3 import sqrt  # Get our 3rd square root implementation.

# test_factorial() is omitted for brevity

def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math_3`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

Listing 8.15: The output of the successful unit tests in Listing 8.14.

```
$ pytest --timeout=10 --no-header --tb=short test_my_math_3.py
========================== test session starts ==========================
collected 1 item
test_my_math_3.py .                                                [100%]
=========================== 1 passed in 0.01s ===========================
# pytest 9.2.0 with pytest-timeout 2.4.0 succeeded with exit code 0.
```

that all of our functions were implemented correctly. However, pytest has helped us to spot two errors. We then fixed these errors.

Best Practice 44

A function which is not unit tested is *wrong*.

Best Practice 45

Good unit tests for a given function should cover both expected as well as extreme cases. For a parameter, we should test both the smallest and largest possible argument values, as well as values from its normally expected range.

Best Practice 46

Good unit tests for a function should cover all branches of the control flow inside the function. If a function does one thing in one situation and another thing in another situation, then both of these scenarios should have associated unit tests.

Many junior programmers are not aware how important unit tests are. Being able to understand, design, and use unit tests is one of the most important abilities in software development.

> No single factor is likely responsible for SQLite's popularity. Instead, in addition to its fundamentally embeddable design, several characteristics combine to make SQLite useful in a broad range of scenarios. In particular, SQLite strives to be:
> [...]

> **Reliable**. There are over 600 lines of test code for every line of code in `SQLite` [265]. Tests cover 100% of branches in the library. The test suite is extremely diverse, including fuzz tests, boundary value tests, regression tests, and tests that simulate operating system crashes, power losses, Input/Output (I/O) errors, and out-of-memory errors. Due to its reliability, `SQLite` is often used in mission-critical applications such as flight software [264]
>
> — *Kevin P. Gaffney, Martin Prammer, Laurence C. Brasfield, D. Richard Hipp, Dan R. Kennedy, and Jignesh M. Patel [212], 2022*

`SQLite` is the most used SQL DB in the world. It is installed in nearly every smartphone, computer, web browser, television, and automobile [107, 212, 637]. And its core developers mark reliability, shown by thorough tests, as one of the four reasons for that.

Tests are incredibly important. And we have seen this here. I think that to most of us, our original `sqrt` implementation looked fine. Of course, we probably noticed that it won't handle negative numbers in a reasonable way, but this one we can set aside for when we can raise `Exceptions` by ourselves. I do not think that most of us saw the issue with `0.0`. And I also doubt that many predicted the endless loop for `inf`. Or even had the values `inf` and `nan` on their radar as possible inputs. Indeed, we may have only realized this issue once we began thinking about testing the possible inputs. Well, even if you did realize all these problems long before me, the issue stands: We can benefit tremendously from tests.

8.4 Function Arguments: Default Values, Passing them by Name, and Constructing them

After the discussion of unit tests, let us now come to a lighter topic: passing arguments to functions. We have already seen many examples of this. Our `gcd` function from back in Listing 8.3 has two parameters `a` and `b`. We can invoke this function by writing the values of these parameters in parentheses after the function name. `gcd(12, 4)` will invoke `gcd` and assign `12` to `a` and `4` to `b`. What else can we do with parameters?

We can also let parameters have so-called *default values*. If a parameter has a default value, then specifying a value for the parameter when calling the function becomes optional. We can specify the value of the parameter *or* we can simply omit it, i.e., not assign a value to it. In the latter case, the parameter will then have the default value. From inside the function, this looks the same as if we passed in the default value.

As a simple example, let us implement the PDF of the normal distribution [2, 190, 607]. You may

Listing 8.16: Implementing the Probability Density Function (PDF) of the normal distribution as function with default argument values. (src)

```
"""The Probability Density Function (PDF) of the Normal Distribution."""

from math import exp, pi, sqrt


def pdf(x: float, mu: float = 0.0, sigma: float = 1.0) -> float:
    """
    Compute the probability density function of the normal distribution.

    :param x: the coordinate at which to evaluate the normal PDF
    :param mu: the expected value or arithmetic mean, defaults to `0.0`.
    :param sigma: the standard deviation, defaults to `1.0`
    :return: the value of the normal PDF at `x`.
    """
    s2: float = 2 * (sigma ** 2)  # stored for reuse
    return exp((-((x - mu) ** 2)) / s2) / sqrt(pi * s2)  # compute pdf
```

Listing 8.17: Using the PDF of the normal distribution implemented in Listing 8.16. (stored in file use_normal_pdf.py; output in Listing 8.18)

```
"""Use the Probability Density Function of the Normal Distribution."""

from normal_pdf import pdf  # import our function

print(f"f(0,0,1) = {pdf(0.0)}")  # x is given, default used otherwise.
print(f"f(2,3,1) = {pdf(2.0, 3.0)}")  # x and mu are given, sigma=1.0.
print(f"f(-2,7,3) = {pdf(-2.0, 7.0, 3.0)}")  # all three are given.
print(f"f(-2,0,3) = {pdf(-2.0, sigma=3.0)}")  # x and sigma given.
print(f"f(0,8,1.5) = {pdf(mu=8.0, x=0.0, sigma=1.5)}")  # unordered...

# We call the function using a dictionary of parameter values.
args_dict: dict[str, float] = {"x": -2.0, "sigma": 3.0}
print(f"f(-2,0,3) = {pdf(**args_dict)}")  # notice the double "*" ("**")

# We call the function using a tuple of parameter values.
args_tup: tuple[float, float, float] = (-2.0, 7.0, 3.0)
print(f"f(-2,7,3) = {pdf(*args_tup)}")  # notice the single "*"

# We call the function using a list of values, but leave one default.
args_lst: list[float] = [2.0, 3.0]
print(f"f(2,3,1) = {pdf(*args_lst)}")  # notice the single "*"
```

↓ `python3 use_normal_pdf.py` ↓

Listing 8.18: The stdout of the program use_normal_pdf.py given in Listing 8.17.

```
f(0,0,1) = 0.3989422804014327
f(2,3,1) = 0.24197072451914337
f(-2,7,3) = 0.0014772828039793357
f(-2,0,3) = 0.10648266850745075
f(0,8,1.5) = 1.7708679390146084e-07
f(-2,0,3) = 0.10648266850745075
f(-2,7,3) = 0.0014772828039793357
f(2,3,1) = 0.24197072451914337
```

remember from high school math that this function, let's call it f, defined as

$$f(x, \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-(x-\mu)^2}{2\sigma^2}} \tag{8.2}$$

Here, μ is the expected value of the distribution and σ is its standard deviation (making σ^2 its variance). x is, so to say, the input value of this function. It represents a value that a normally-distributed random variable could take on. This function f describes typical bell-shaped curve of the normal distribution as sketched in Figure 8.3. Implementing this function as a, well, function in Python is straightforward. The Python file normal_pdf.py in Listing 8.16 offers the function `pdf` with three parameters: `x`, `mu`, and `sigma`, which represent x, μ, and σ, respectively.

Now, with the two parameters μ and σ of f (respectively `mu` and `sigma` of `pdf`), we can represent the general normal distribution. The *standard* normal distribution has $\mu = 0$ and $\sigma = 1$, i.e., is centered around the mean 0 and has a standard deviation (and variance) of 1. Very often, we want to compute the `pdf` for the standard normal distribution. We therefore define the *default* values for `mu` to be `0.0` and for `sigma` to be `1.0`. This is done directly in the header of the function. Instead of writing `x: float, mu: float, sigma: float`, we just have to write `x: float, mu: float = 0.0, sigma: float = 1.0`.

Nothing changes regarding how the parameters are used inside the function. We just use them completely normally, like any other parameter. The function body does not know where their values come from.

In order to implement the PDF of the normal distribution, we first need to import the functions `exp` and `sqrt` from the module `math` as well as the constant `pi`. Here, `exp(x)` computes $e^{\texttt{x}}$. The term $2\sigma^2$

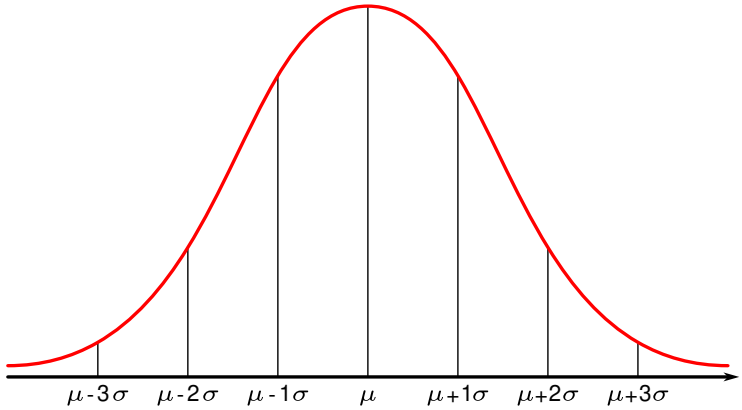


Figure 8.3: A sketch of the PDF of the normal distribution given in Equation 8.2.

appears twice in Equation 8.2, once under the square root in the first fraction and once in the fraction used in the exponent. So we compute it once and store it in a variable `s2`. This simplifies the equation to $[e^{-(x-\mu)^2/\texttt{s2}}]/[\sqrt{\pi * \texttt{s2}}]$. Notice how the power operator `a ** 2` is equivalent to `a`^{2}.

In program `use_normal_pdf.py` given in Listing 8.17, we import and use our new `pdf` function. When calling `pdf`, we can omit the values of the parameters with default values, in which case they will take on their default values. For example, invoking `pdf(0.0)` is equivalent to calling `pdf(0.0, 0.0, 1.0)`. We also can specify some of the parameters with default values while omitting others. For instance, the function call `pdf(2.0, 3.0)` is the same as `pdf(2.0, 3.0, 1.0)`. Obviously, we must always specify the first parameter (`x`), because it has no default value.

Default values do not need to be literals. They can also be the results of expressions, such as `sqrt(2.0)`. However, they are only evaluated exactly once, when the function is defined [275]. Then they are stored in the function header and reused whenever needed. This has an interesting implication: What if the default value for a function argument is mutable, say, a `list` or `set`? Well, they never should be. Because that could lead to awful problems [275].

Best Practice 47

Default parameter values must always be immutable [275].

The default value of a function argument must always be immutable. Or, at least, *should* be immutable, because, technically, you could use mutable values as well. Of course, technically, you can also set the value of the variable `pi` in the `math` module to `3` (see Section 4.1.3). If you *would* pass in, e.g., a `list`, then the function could modify the list and the next call to this function would then receive this modified list. Even worse, if the function was to return the list, it could be modified outside of the function. The behavior of such code could become arbitrarily hard to debug.

In cases where a parameter is of a mutable type and we require a default value for it, we could instead use `None`. Inside the function body, we could then check for `None` and implement appropriate behavior.

Let's circle back to passing arguments to functions. Assume that we have a function `def g(x: int, y: int)`. Normally, we would call it like `g(1, 2)`, in which case the function body sees `x == 1` and `y == 2`. However, you can also pass in arguments very much in the same way that you use to assign a variable, in the form `parameterName=value`. We could write `g(x = 1, y = 2)` or, if we feel naughty, `g(y = 2, x = 1)`. Both variants will be equivalent to our original function call. All we did is to explicitly write the names of the parameters when providing their values.

We now revisit our `pdf` function defined in file `normal_pdf.py` and used in `use_normal_pdf.py`, given in Listings 8.16 and 8.17, respectively. The function `pdf` has a parameter `x` without default value, followed by a parameter `mu` with default value, which, in turn, is followed by a parameter `sigma` with default value. What would we do if we want to specify the value of the parameter `sigma` of our function, but leave `mu` at its default value?

We can do this by passing in values by parameter name: `pdf(-2.0, sigma=3.0)` passes in `-2.0` for `x` and `3.0` for `sigma`. It does not specify any value for `mu`, leaving it at its de-

fault value. This renders the call equivalent to `pdf(-2.0, 0.0, 3.0)`. This passing in of arguments by specifying `parameterName=value` also allows us to specify the arguments in arbitrary order. `pdf(mu=8.0, x=0.0, sigma=1.5)` is an example of this. Don't do such things, though.

Listing 8.17 illustrates also another interesting way to call a function in Python. As we have established by now, the parameters of a function have names. We can write something like `pdf(mu=8.0, x=0.0, sigma=1.5)` to assign arguments. Calling `pdf(-2.0, sigma=3.0)` is equivalent to writing `pdf(x=-2.0, sigma=3.0)`. Passing arguments to a function basically means to a assign values to keys (the parameters). This is at least a little bit similar to the creation of a dictionary literals.

In fact, Python allows us to also construct the arguments for a function call in a collection. We can create a dictionary with the values `{"x": -2.0, "sigma": 3.0}`. Let's store this dictionary in variable `args_dict`. Can we now somehow pass in the key-value pairs from `args_dict` as parameter-argument pairs to `pdf`?

Indeed, we can. We just have to write `pdf(**args_dict)`. Doing this will unpack the dictionary `args_dict` and pass all the values under their assigned names in as arguments to their corresponding parameters. `pdf(**args_dict)` is thus equivalent to `pdf(x=-2.0, sigma=3.0)`.

Two things are to notice here: First, the double `*`. The `*` here is often called wildcard, star or asterisk. The double-wildcard `**` is written *before* the dictionary. The `**` is telling Python to unpack the dictionary this way. Second, default argument values still apply here. We did not specify a value for `mu`. This means that `mu` will have value `0.0` in this function call.

Maybe we do not want to pass in the arguments by parameter names but simply by position. Actually, we have always done it like this in the past. Then, we can construct a sequence, e.g., a `list` or `tuple` with the parameter values. Of course, `lists` and `tuples` do not store key-value relationships, only values at positions. We could create a tuple `args_tuple` with the value `(-2.0, 7.0, 3.0)`.

Then, we invoke `pdf` like this: `pdf(*args_tuple)`. This will basically fill in the (three) values from the `tuple` one by one into the parameter slots of the function. In other words, this is equivalent to writing `pdf(-2.0, 7.0, 3.0)`.

This time, only a single wildcard `*` is placed before `args_tuple`. We can also pass the parameters in by "unpacking" a list. In our example Listing 8.17, we create the list `args_list = [2.0, 3.0]`. Calling `pdf(*args_list)` then is the same as writing `pdf(2.0, 3.0)`, which, in turn, is identical to `pdf(2.0, 3.0, 1.0)`. Again, parameters with default values do not need to be supplied.

At first glance, the use of all of the above is not entirely clear. What do we need default parameter values for? Well, in some cases, you may want to enable a user to "customize" your functions. A typical example is the `plot` method of the `Axes` object provided by popular Matplotlib library. You will normally provide sequences of x- and y-coordinates to this function it will draw a line which goes through all the points specified this way. However, you can also optionally specify a color for the line, markers to be painted at the points, line dash style, a label, colors and sizes for the markers, a z-order to be used if multiple lines are drawn, and so on, and so on. The use of default arguments allows the function call to be relatively simple in most cases, while still allowing the user to do more complex formatting if need be.

From this example, we can also directly extrapolate a use case for building the arguments of a function in a dictionary. Imagine that you write an own function that uses one of the plotting methods of Matplotlib. Let's say that your function does a plot call where it provides ten parameter values. However, you have one special case where you need to provide one more parameter, maybe a line dash style that you otherwise do not need to provide. Then, you could have some `if ... else` in your code that branches to do the ten-parameter-call in one case and the eleven-parameter-call in the other. This means that a rather complex function call appears twice in a very similar manner. If you instead construct a dictionary with the ten parameters. In the `if` branch just add the eleventh parameter if need be. We now only need a single function call using the double-wildcard method. The code will become much simpler. The difference between the two cases will also be much more obvious. We have reduced the potential for errors significantly.

With the default parameter values and function calls via `*` and `**`, we learned two more aspects that make it easier for us to work with functions in Python. With the default values, we can now design more versatile function-based APIs that allow the user of our function to provide values for some parameters, while leaving others at reasonable default settings. By constructing arguments in collections, we can create code that is easier to read when we deal with functions with many arguments

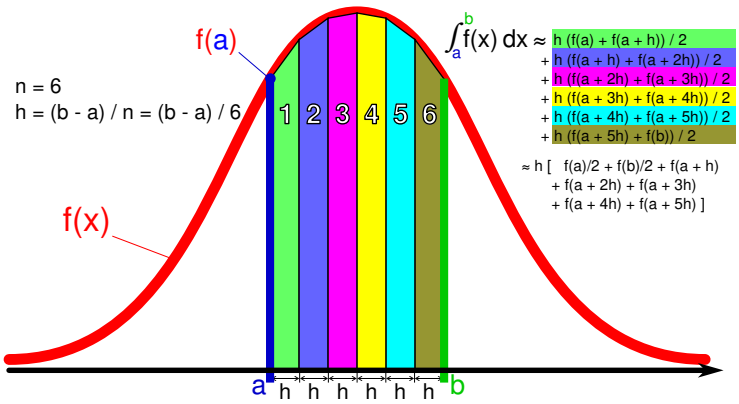


Figure 8.4: A sketch of the composite trapezoid method for approximating definite integrals.

that may be called slightly differently depending on the situation.

8.5 Functions as Parameters, `Callables`, and `lambdas`

We have learned how to define and call functions. We learned how functions can return their results. We learned how to test functions. And we just learned that we can basically construct a function call by placing the parameter values into collection objects and then invoke the function by "unpacking" the collection using either `*` (for position-based parameters) or `**` (for dictionaries). But there is one more interesting thing that we can do with functions. You see, in `Python`, all things are objects [142, 274]. References to objects can be stored by variables or passed in as function arguments. Functions are objects too [274]. This means that we can also store functions in variables or pass them as arguments to other functions!

At first glance, this sounds awfully odd. Why would someone like to pass a function as parameter to another function? At second glance, there are a wide variety of situations where we would actually want to do that. Having done so many mathematics-based examples in this book, let's pick one such situation arising in maths.

In high school, you would have learned about integration and differentiation. A *definite integral* is the formal calculation of the area under a function over a certain range along the x-axis. It uses infinitesimal [181, 312] stripes over the region to do so.

The second[5] fundamental theorem of calculus states [14, 627]: Given a function $f(x)$ which is continuous over the real interval $[a, b]$ and its antiderivative $F(x)$, the definite integral $\int_a^b f(x)\,dx = F(b) - F(a)$ (where antiderivative means that $F' = f$). In other words, if we want to get the area beneath $f(x)$ over the interval $[a, b]$, we would first obtain the antiderivative and then simply calculate $F(b) - F(a)$.

Obtaining the antiderivative involves pen-and-paper symbolic maths. We cannot really program such maths at this stage in this book as an example. But we can go back to the original definition of the definite integral, namely that it equals the area beneath the function over the interval $[a, b]$. This area can obtained by using infinitesimal (read: immeasurably small) strips of the region.

While we cannot really use "immeasurably small" strips, we could use "fairly small" ones to approximate the right result. Let's say that we have a function $f(x)$ as a parameter of our program. We would also have the interval limits a and b as parameters. Then we could divide the range $[a, b]$ into n strips. For each strip, we could approximate the area beneath $f(x)$. Then we add up the n areas, and have a rough approximation of the total area. n could be another parameter for our integration approach. The larger n, the smaller will the strips get, the more accurate should our estimate become (and the longer it will take to get it)[6].

This brings us to the question how we can do this, in particular, how to approximate the area of one such strip. The composite trapezoid method is one very simple implementation of this idea [20, 183, 305]. As illustrated in Figure 8.4, it treats the strips as right trapezoids. The baseline of the trapezoid

[5] Depending on the source, sometimes this is also called the first fundamental theorem of calculus [627].

[6] Of course, eventually, we will run into numerical precision issues and the approximation will become worse again for larger n...

Listing 8.19: Using the composite trapezoid method to numerically approximate definite integrals, as sketched in Figure 8.4. (stored in file `integral.py`; output in Listing 8.20)

```
"""Numerical integration using the trapezoid method."""

from math import pi, sin
from typing import Callable

from normal_pdf import pdf


def integrate(
        f: Callable[[float], float | int],
        a: int | float = 0.0,
        b: int | float = 1.0,
        n: int = 100) -> float:
    """
    Integrate the function `f` between `a` and `b` using trapezoids.

    :param f: the function to integrate: in float, out float or int
    :param a: the lower end of the range over which to integrate
    :param b: the upper end of the range over which to integrate
    :param n: the number of trapezoids to use for the approximation
    :return: the approximated integration result
    """
    result: float = 0.5 * (f(a) + f(b))  # Initialize with start + end.
    h: float = (b - a) / n  # The base length of the trapezoids.
    for i in range(1, n):  # The steps between start and end.
        result += f(a + h * i)  # Add f(x) between trapezoids.
    return result * h  # Multiply result with base length.


print(f"\u222b1dx|0,1 \u2248 {integrate(lambda _: 1, n=7)}")
print(f"\u222bx²-2dx|-1,1 \u2248 {integrate(lambda x: x * x - 2, -1)}")
print(f"\u222bsin(x)dx|0,\u03C0 \u2248 {integrate(sin, b=pi, n=200)}")
print(f"\u222bf(x,0,1)dx|-1,1 \u2248 {integrate(pdf, -1)}")
print(f"\u222bf(x,0,1)dx|-2,2 \u2248 {integrate(pdf, -2, 2)}")
```

↓ `python3 integral.py` ↓

Listing 8.20: The `stdout` of the program `integral.py` given in Listing 8.19.

```
∫1dx|0,1 ≈ 1.0
∫x²-2dx|-1,1 ≈ -3.3332
∫sin(x)dx|0,π ≈ 1.9999588764792162
∫f(x,0,1)dx|-1,1 ≈ 0.6826733605403601
∫f(x,0,1)dx|-2,2 ≈ 0.9544709416896361
```

is a piece of the x-axis with length $h = (b-a)/n$. Clearly, $n*h = b-a$ and thus, n trapezoids of equal base length form the range of the x-axis under $f(x)$. The baseline first trapezoid starts right at $x = a$ and extends to $x = a+h$. The second trapezoid starts at $x = a+h$ and extends to $x = a+2h$. The last trapezoid starts at $x = a+(n-1)h$ and extends to $x = a+nh = b$. Each trapezoid has two parallel sides meeting with its baseline in right angles. The length of these sides are the values of $f(x)$ at the corresponding x-coordinate. The area of the i-th trapezoid is thus $h[f(a+(i-1)h) + f(a+ih)]/2$. Summing up the n areas yields an approximation of the definite integral, as sketched in Figure 8.4.

Each value $f(a+ih)$ except for $i = 0$ and $i = n$ appears twice in the sum and each time is halved. Instead of computing these values twice, dividing them by two, and then adding them, we can simply compute them only once.

We want to implement this method as a function `integrate`. Obviously, `integrate` has the parameters `a` and `b` denoting the limits of the interval to integrate over. We allow them to be either `int` or `floats`, denoted by the type hint `float | int`. Let's give them the default values `0.0` and

`1.0`, respectively. Then there is the parameter `n`, an `int`, denoting the number of trapezoids we need to construct. A good default value for it may be `100`. But we need another parameter, namely the function `f` that we want to integrate. How can we specify this parameter?

The answer is given as program `integral.py` shown in Listing 8.19. The most interesting part of our `integrate` function is its header. More precisely, it is the first parameter: `f: Callable[[float], float |int]`. `Callable` is the type hint for anything that can be called, i.e., functions [12]. Like the type hints for lists and tuples, it can be parameterized following a scheme with square braces [391]:

```
"""The syntax of type hints for callable objects like functions."""

Callable[[parameterType1, parameterType2, ...], resultType]
```

Inside the `Callable[...]`, we first provide the list of types of the parameters of function, again in square brackets. Then follows a comma `,` and then follows the return type. The `f: Callable[[float], float |int]` in our listing thus states that our function `integrate` expects, as first parameter, *another function* `f`. `f` must accept one parameter of type `float`. The return type of `f` is `float |int`, i.e., it should return either a `float` or an `int` [453]. Notice that the type `Callable` is provided by the module `typing`[7], so we need to import it first if we want to use it.

The actual implementation of `integrate` is straightforward. Inside the function, we add up the different trapezoids as shown in the simplified equation in Figure 8.4. Only the values of `f` at the interval ends `a` and `b` need to be halved in the sum. We therefore initialize the sum `result` as `0.5 * (f (a)+ f(b))`. We compute the trapezoid base length `h` as `(b - a)/ n`. Then we iterate counter `i` from `1` to `n - 1` and add `f(a + h * i)` to `result` in each step. Finally, we return need to multiply this with the trapezoid base length `h` to get the approximation of the area under the curve. We thus return `result * h`.

Let's now test how well our trapezoid-based integration works. First, let's compute the definite integral $\int_0^1 1\,dx$. For this purpose, we need to pass the function $f(x) = 1$ as the `f` parameter of `integrate`. We could do this by writing:

```
"""The syntax of a unary function always returning a constance value."""
def const_1(x: float) -> float:
    return 1.0    # always return 1.0, regardless of the value of x

# or

# It is common to use "_" as name for parameters that we will ignore.
def const_1(_: float) -> float:
    return 1.0    # always return 1.0, parameter "_" is ignored.
```

where the `_` indicates that we are actually going to ignore this parameter (see Best Practice 34). We could then invoke `integrate(const_1)`.

However, there also is a more compact way to specify functions that we are only going to use once: The so-called `lambdas` [335], nameless functions defined inline, which have the structure

```
"""The syntax of lambdas, i.e., inline function definitions."""

# This defines a nameless inline function with two parameters that
# returns "return_value".
lambda param1, param2: return_value
```

This inline function definition begins with the keyword `lambda`. Then, the names of the parameters follow (if any), separated by commas `,`. Then there is a colon `:`, after which an expression computing the return value of the inline function follows. The special thing about `lambdas` is, that their body only consists of this single expression. They are basically a single line of code with the only purpose to

[7] [12] states that this import is now deprecated and we should use `collections.abc.Callable` instead. In the past, this created some errors for me, so for now we stick with using `typing`.

Listing 8.21: The results of static type checking with Mypy of the program given is in Listing 8.19.

```
$ mypy integral.py --no-strict-optional --check-untyped-defs
Success: no issues found in 1 source file
# mypy 2.4.1 succeeded with exit code 0.
```

compute a return value. Due to the `:` in the notation, we cannot annotate `lambdas` with type hints. As a `lambda` expression is very small and only used once, this does not pose a serious problem.

We now need to define a function returning the constant `1` as `lambda`. This function must accept one parameter, because otherwise we cannot plug it into `integrate`. Since we do not care about the value of this parameter, we can simply write:

```
"""lambdas can be used as values for Callable parameters."""

# If the Callable type hint indicates that the lambda should have a
# parameter, but we want to ignore that parameter, we should call it
# "_".
integrate(lambda _: 1.0)
```

`lambdas` are functions that we only want use in a single place. This is clearly the case for a function that ignores its parameter and always returns `1.0`. So we pass this expression into our `integrate` function as value of the parameter `f`.

Obviously, the area under the constant function $f(x) = 1$ over the range $[0, 1]$ is also 1. Our function `integrate` thus has passed a first sanity test. Using `n=7` trapezoids, we get exactly this result. In the output of our program, we write `∫1dx|0,1` where the `|0,1` denotes the limits $[0, 1]$ of the interval over which we integrate. We use the Unicode escape sequence `"\u222b"` to represent the $\int$ character in the f-strings.

Having passed this simple sanity test, let's try to compute the beneath under the function $g(x) = x^2 - 2$ over the interval $[-1, 1]$, i.e., $\int_{-1}^{1} x^2 - 2\,dx$. The antiderivative of $g(x)$ is $G(x) = \frac{1}{3}x^3 - 2x + c$ and $G(1) - G(-1) = [\frac{1}{3} - 2] - [-\frac{1}{3} + 2] = \frac{2}{3} - 4 = -3\frac{1}{3} = -3.\overline{3}$. We pass the function $g(x)$ as `lambda` expression into our `integrate` function by writing `lambda x: x*x - 2`. We also need to specify a value for the parameter `a`, namely `-1`. Using the default value of `n=100` steps, our function returns `-3.3332`, which is fairly close to $-3.\overline{3}$.

We now integrate the sine function over the interval $[0, \pi]$. We can import `sin` from the `math` module and pass it into our function for `f`. We also need to specify `b = pi`, which we, too, have imported from `math`. This time, we use `n = 200` trapezoids to approximate the definite integral. The antiderivative of $\sin x$ is $-\cos x + c$, so the expected output would be $[-\cos \pi] - [-\cos 0] = [-(-1)] - [-1] = 2$. Indeed, our function delivers about 1.99996, which is, again, fairly close.

As a final example, we also integrate the PDF of the normal distribution. We implemented this function and called it `pdf` back in Listing 8.16, so we just need to `import` it from there. Notice that this function actually had three parameters, `x`, `mu`, and `sigma`. The latter two had the default values `mu = 0.0` and `sigma = 1.0`. If these default values are used, the function computes the PDF of the *standard* normal distribution.

Interestingly, although we demand that the function `f` supplied to `integrate` should only have one parameter, we can still pass in `pdf` with its three parameters as `f`. This is possible because its second and third parameter are optional. Even Mypy accepts this function as valid parameter, as its output (or better, the lack of it) in Listing 8.21 suggests.

The standard normal distribution has standard deviation $\sigma = 1$ and mean $\mu = 0$. If we integrate it over the interval $[-1, 1]$, we basically compute the probability that a standard normally distributed random variable $\mathcal{X}$ is drawn from the interval $[\mu - \sigma, \mu + \sigma]$, i.e., from within one standard deviation from the mean. As you may still know from high school math, this probability is roughly 68.26% [2, 190, 607]. Similarly, the probability that such a random variable is sampled from within $[-2, 2]$, i.e., not more than two standard deviations away from the mean, is about 95.44% [2, 190, 607]. The output of our little integrator function in Listing 8.20 fits perfectly to this.

As a final clarification, let us mention that trapezoid-based approximation of definite integrals is by no means the best approach. Others methods, like Simpson's rule, may often provide better results [183]. Still, as an example of functions that work on other functions, it was suitable.

8.6 Summary

Functions are the central building blocks needed to create modular code. They allow us to put code into units with clearly defined interfaces. The input of a function are its parameters (if any). The output is its return value (if any). Both parameters and return value can be annotated with type hints. The description of what the function does as well as what the parameters and the return value mean, goes into the docstring.

This clear definition and separation from the rest of the code has several advantages. First, multiple programmers can work together on a project. They can work on different functions, which we can place into different modules (`Python` files). If we did not have functions available, this would be impossible.

Second, by using proper type hints for the function parameters and return value as well as proper docstrings, the behavior of functions is easier to understand by fellow programmers. Type checkers like `Mypy` can also verify whether functions are called with correct parameters more easily. The code does not just become more modular, but easier to maintain. It can be checked by static code analysis tools as well.

Third, functions allow us to reuse pieces of code in different locations. If we want to compute the definite integral of two different functions, we do not have to implement two routines for integration. We can make one and use it in two different locations.

Fourth, if we have a function with clearly defined input parameters and output results, then we can *test* this function in isolation. It may be extremely difficult to test a complete program. However, it may be rather straightforward to test a small function.

Unit tests, for example using the `pytest` framework, allow us to do just that. It is also not hard to imagine that unit tests can compound our trust into our code: In the previous sections, we developed a function `pdf` for computing the probability density function of the normal distribution and a function `integrate` for approximating definite integrals. Assume that we properly applied unit tests to these two functions and have shown for both of them that they produce the expected results for a wide range of different inputs. It would be easy to see that this then would also give us a certain confidence that using `integrate` to compute a definite integral over a certain input range of `pdf` should be correct, too. This could be verified with some additional unit tests. Working our way up from testing small components to larger and larger building blocks of an application gives us a good chance to have only few bugs in the final product.

These are a lot of advantages of using functions. But functions in `Python` also come with lots of nice features as well.

For example, we can specify default values for function parameters. This allows us to create functions that can be invoked using only few arguments in the normal case and that can be customized with more parameters in special cases. Since functions themselves are objects, they can be stored in variables and parameters as well. The proper type hint for such variables or parameters is defined using the `Callable` type. Functions that accept `Callables` as parameter can also accept `lambdas`. `lambdas` are an abbreviated way to define functions, usually in a single line and only for a single use.

Chapter 9

Exceptions

So far, we have mainly focused on writing correct code. We try to create code that is free of errors.

When we execute our programs, then there are at least two things that can go wrong. On the one hand, we can never really be sure that our program code is free of programming mistakes. The larger a project gets, the more likely it is that there are some bugs hidden somewhere in the code. Thorough unit tests can reduce the likelihood of bugs, but it cannot entirely prevent them.

On the other hand, our programs do not exist all by themselves in a vacuum. They receive input, maybe from files, maybe from the user, maybe from sensors. This input may be wrong.

Coarsely, we can group both kinds of problems together as situations that were not anticipated, that are *exceptions* from the intended program flow. We already have encountered such situations. For example, trying to access a character of a string at an index greater than or equal to the length of the string will lead to an `IndexError`, as we saw in Section 3.6.1. The attempt to modify an element of a `tuple` is punished with a `TypeError` in Section 5.3. Back in Section 3.3.5, we saw that trying to compute something like `(10 ** 400)* 1.0` will yield an `OverflowError`, as the integer 10^{400} is too large to be converted to a `float` during the multiplication with `1.0`.

Clearly, some of these errors may result from programming mistakes. But they could just as well result from invalid data being entered in the input of the program.

9.1 Introduction

When we create a new function or program, we have to face the question: *What should we do if we receive incorrect input?* We can imagine three different approaches:

1. We simply ignore the issue. If the input of our program or function is faulty, then the output will be wrong, too. This is often called Garbage In–Garbage Out (GIGO) [445]. Converting the integer `10 ** 400` to a `float` could just yield `inf`, for example.
2. We try to sanitize the input. For example, our `factorial` function in Listing 8.1 expects an integer as input. If someone were to pass in the floating point value `2.4` instead, we could round it to `2` and return the corresponding result. Matter of fact, our `sqrt` function implemented in Listing 8.10 returns `0.0` if we pass in a negative number.
3. We can guard the function by raising an `Exception` [216, 389, 505].

The latter is what `Python` does in the examples mentioned initially: While it *could* simply ignore if we try to overwrite an element of a `tuple`, it instead raises a `TypeError`, for example. Personally, I am also a fan of this approach. And the `Python` documentation is, too:

> Errors should never pass silently. Unless explicitly silenced.
>
> — *Tim Peters [443], 2004*

If we follow the GIGO paradigm, then faulty data will propagate. Maybe the output of our function is fed as input into another function, whose output is then piped into another function, and so on. An error could then lead to some crash down the line. Then again, if functions that are written under the assumption that GIGO is OK are paired with such that perform input sanitization, errors could

remain unnoticed. The erroneous results could then become part of some actual, real-life decisions and designs. And even if found out, it will be extremely hard to discover where things went wrong in the long chain of computations and function calls. It could even be that faulty results are stored in files and then cause other tools to crash later. And *later* could mean something like a week later. And *other tools* could mean *programs run by someone else in another department*. Good luck finding the piece of code that caused the error then.

Input sanitization could cover some error that happened earlier. But there are two more problems with this: First, it is not clear whether sanitized data is actually fixed. Assume that we have a function `compute` that has a parameter `x` that only accepts integers. Now our function receives a value that was somehow corrupted to `5.5`. Should this sanitize to `5` or `6`? Either could be right or both could be wrong. Would it make sense to simply always do `int(x)`? We can hardly make a fixed rule that will always pick the right choice.

Second, input sanitization can cause other programmers to use our functions incorrectly. In the previous example of the function `compute` with the parameter `x` that should always be an integer, maybe we chose to do `int(x)` and calculate with that. By doing so, we have sanitized all finite `floats` to `ints`. Maybe that looked like a good idea to us. However, maybe we now notice that people start calling our function like this: `compute("12")`. Of course, `int(x)` will work just as well with `"12"` as input and return `12`. Now we suddenly have people who call our function `compute` with strings.

And we will have to keep supporting that. Because now there maybe exists code in some applications that relies on the option to pass strings to our function. If we change our mind and modify the function to no longer accept such nonsense... ...then this other code, that until now worked perfectly fine, will crash. Thus, input sanitization also encourages sloppy programming.

This leaves raising an `Exception`. But what does that actually mean? Raising an `Exception` means two things:

1. Information about the error and information about the current execution state (the current line of code and the function call hierarchy) is stored in an object (the `Exception`).
2. The control flow immediately leaves the currently executed block of instructions as well as all calling blocks or functions. It jumps up in the call hierarchy until reaching code that can properly process the raised exception. If no such code exists, the current process is terminated with an exit code different from `0`.

In other words, raising an exception is a way to exit the current control flow path and to signal an error that must either explicitly be handled by code or will lead the process to terminate. In my opinion, this is the best way to handle incorrect input or other erroneous situations for several reasons:

1. It clearly and explicitly shows that an error has happened, *where* it happened, *when* it happened, and, to some degree, *why* it happened. This makes it much easier to find out whether the error is caused by invalid input or by a programming mistake.
2. It prevents GIGO from occurring. Thus, it prevents a faulty situation or corrupted data from propagating out of the current context. If an error occurs, then the raised exception terminates the current path of execution and this stops the contaminated control flow.
3. It forces programmers to explicitly deal with the error condition. An error cannot be simply ignored. Indeed, someone who calls our code might write code that ignores or discards the `Exceptions` that we raised. But they have to do so *explicitly* in their code. Thus, they have to *intentionally* deal with the possible error condition. It cannot happen that an error gets overlooked. An exception that is not handled will lead to the termination of the process.
4. One might argue: *"But what if the process crashes because an exception is not handled? Isn't this bad?"* The answer may be: What is worse? That an error causes the current process to crash unexpectedly or that all future results after the error are wrong and, even worse, are wrong *unnoticed*?

Best Practice 48

Errors should *not* be ignored and input data should *not* be artificially sanitized. Instead, the input of functions should be checked for validity wherever reasonable. Faulty input should always be signaled by error conditions breaking the program flow. `Exceptions` should be raised as early as possible and whenever an unexpected situation occurs.

9.2 Raising Exceptions

Raising exceptions is very easy. `Python` offers us a variety of different types of exceptions. Some prominent examples are `ValueError`, which is used when a parameter has an incorrect value, and `TypeError`, which we use if a parameter has an illegal type. If you want to signal an error of a specific type, all you have to do is to write `raise`, followed by the exception type. If, additionally, you also want to provide an error message, then you can provide this message as a string in parentheses directly after the exception type name.

```
"""
The syntax of raising an `Exception` in Python.

There are many different types of `Exception`s for different situations
in Python. For example, `ArithmeticError`, `OverflowError`,
`IndexError`, `TypeError`, and `ValueError`, to name a few.

They can be raised by writing `raise ExceptionType` without additional
information or by providing an error message string, like
`raise ExceptionType("This is the error message.")`.
"""

# Raise an `Exception` of type `TypeError` without an error message.
raise TypeError

# Raise an `Exception` of type `ValueError` with an error message.
raise ValueError("Cannot compute ln(-1)!")
```

Often, we want to raise an exception inside a function. This works exactly as above, but we should make sure to also mention the error type in the docstring of the function. This is done by putting a `:raises ExceptionType: explanation` note after the parameter list in the docstring.

```
"""Raising an `Exception` in a function, as explained in docstring."""

def my_func(x: int) -> None:
    """
    This is a function.

    :param x: a parameter
    :raises ValueError: if `x < 1`
    """
    if x < 1:
        raise ValueError("Invalid x!")
```

Best Practice 49

Any function that may raise an exception should explain any exception that it explicitly raises in the docstring. This is done by writing something like `:raises ExceptionType: why` where `ExceptionType` is to be replaced with the type of the exception raised and `why` with a brief explanation why it will be raised.

Listing 9.1: A new variant of the `sqrt` function from back in Listing 8.10 that raises an `ArithmeticError` if its input is non-finite or negative. (src)

```
"""A `sqrt` function raising an error if its result is not finite."""

from math import isclose   # Checks if two float numbers are similar.
from math import isfinite  # A function that checks for `inf` and `nan`.


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v == number`.
    :raises ArithmeticError: if `number` is not finite or less than 0.0
    """
    if (not isfinite(number)) or (number < 0.0):  # raise error
        raise ArithmeticError(f"sqrt({number}) is not permitted.")
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0, negative values were checked above.

    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess    # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Let us now first look at an example of how we can signal an error condition in our code. For this purpose, we re-visit the square root function `sqrt` that we implemented back in file `my_math_3.py` shown as Listing 8.13 in Section 8.3. In that implementation, we realized that certain input values such as `inf`, `-inf`, `nan`, and `0.0` deserve special treatment. We also found that nothing could stop a user from passing a negative number as input to our `sqrt` implementation. We did not yet have any means to deal with nonsense in a reasonable way, so we decided to just return `0.0` in that case.

Obviously, this is a bad idea because passing a negative number to `sqrt` can only mean two things: Either the programmer who did that does not know what a square root is. Or the negative input came as the result of another computation and that computation somehow was wrong. In the first case, we should somehow make it explicit to the programmer that the square root of a negative number is not defined[1] and that they should question their approach to mathematics. In the second case, we should rather stop the computation right there and then before the incorrect results propagate and do some damage elsewhere. In both cases, raising an exception is much better than returning `0.0`.

We want to signal this error explicitly. Actually, maybe our function should always raise an exception when the result of `sqrt` would not be a finite number. This way, we would ensure that our function either returns a finite number, that can be used in other computations later on. Or, otherwise, it fails and raises an exception. In no situation would it return a wrong number, `inf`, or `nan`. We therefore create a new implementation of `sqrt` in file `sqrt_raise.py` Listing 9.1.

We now have to decide which type of exception most appropriately represents the situation. On the one hand, we could choose a `ValueError`, because the parameter value is invalid. Or we choose an `ArithmeticError`, because the arithmetic computation fails. Both choices are completely valid. I here will choose `ArithmeticError` because I want to express the mathematical nature of the context.

Before writing the actual Python code, we express the new behavior in the docstring of our function, by adding a corresponding `:raises:` note. In our case, any input x for which $\sqrt{x}$ would be either undefined or not finite should lead to an error. Our docstring therefore contains the line `:raises ArithmeticError: if `number` is not finite or less than 0.0`. Any other programmer using our code therefore can easily see what kind of `Exceptions` our code could raise.

[1] at least not as real number in the scope of a `float`

Listing 9.2: Using the new variant of the `sqrt` function from Listing 9.1 that raises an `ArithmeticError` if its input is non-finite or negative. (stored in file `use_sqrt_raise.py`; output in Listing 9.3)

```
"""Using the sqrt function which raises errors."""

from math import inf, nan  # Import infinity and not-a-number from math.

from sqrt_raise import sqrt  # Import our sqrt function.

# Apply our protected square root function to several values.
for number in (0.0, 1.0, 2.0, 4.0, 10.0, inf, nan, -1.0):
    # We get an error when reaching `inf`.
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)

print("The program is now finished.")  # We never get here.
```

↓ `python3 use_sqrt_raise.py` ↓

Listing 9.3: The `stdout`, `stderr`, and exit code of the program `use_sqrt_raise.py` given in Listing 9.2.

```
√0.0≈0.0
√1.0≈1.0
√2.0≈1.414213562373095
√4.0≈2.0
√10.0≈3.162277660168379
Traceback (most recent call last):
  File "{...}/exceptions/use_sqrt_raise.py", line 10, in <module>
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)
                                ~~~~~~~~~~~~
  File "{...}/exceptions/sqrt_raise.py", line 16, in sqrt
    raise ArithmeticError(f"sqrt({number}) is not permitted.")
ArithmeticError: sqrt(inf) is not permitted.
# 'python3 use_sqrt_raise.py' failed with exit code 1.
```

In the actual code, we first check whether either `not isfinite(number)` or `number < 0.0` holds. The `isfinite` function is offered by the `math` module and returns `True` if its argument is a finite number. It will be `False` for either `inf`, `-inf`, or `nan`. The complete expression becomes `True` if the input of the function is not a finite number greater than or equal to 0. In this case, the output of the `sqrt` function would not be a finite number.

If this situation, we `raise ArithmeticError(f"sqrt({number})is not permitted.")`.[2] `raise` is the keyword used to, well, raise an exception, i.e., to signal an error. `ArithmeticError` then creates the object with the information about the exception. We can pass a string as a parameter to this function, and we here chose to pass in an f-string which contains the value of the number with which our `sqrt` function was called. This line of code will force the control flow to immediately exit our `sqrt` function. The `Exception` object "raises" up until it is "caught" (which we discuss later). If it is not caught, the process itself is terminated.

This process termination can be observed in program `use_sqrt_raise.py` shown as Listing 9.2. In the program, we iteratively apply our new `sqrt` function to the values inside a `tuple` using a `for` loop. We write the results of these computations to the standard output stream (`stdout`) using f-string and `print`. We here use the optional parameter `flush` of `print`, which has the default value `False`. We set it to `flush=True`, though, which specifies that the output of `print` is directly and immediately written to the `stdout` stream and not buffered. This is useful in this special example to prevent mix-ups between the `stdout` stream receiving the normal output of the program and the standard error stream (`stderr`) stream where error messages appear. Both streams are captured and presented in Listing 9.3 and the order of the text could otherwise get mixed up.

Anyway, the first five numbers are fine and `sqrt` returns the proper results. However, the latter

[2] It is frowned upon using f-strings directly in the exception constructions, because it adds another potential source of further errors. Imagine that `number` would be a very huge string. But since the context here is tight, I do this anyway.

three numbers, `inf`, `nan`, and `-1.0` all would cause an error. As can be seen, for `0.0`, `1.0`, `2.0`, `4.0`, and `10.0`, the results are printed as anticipated. However, when the `for` loop reaches `inf`, the program is terminated and the so-called stack trace is written to the output.

We briefly discussed the stack trace already back in Section 4.2.1, where we pointed out its high utility in locating the source of the error. This turns out to be true here as well. It begins with the line `Traceback (most recent call last):`. Then, the path of the source code file `use_sqrt_raise.py` and the index of the line in that file where the exception was originally raised are printed. Notice that the actual path to the files will be different depending on where the source code of the examples is located. We replaced the variable part of the path with "{...}". The remaining part of the path, `exceptions/use_sqrt_raise.py`, clearly points out the calling program Listing 9.2 as the culprit.

The following line of text identifies that instruction in Listing 9.2 that caused the error and even marks the offending function invocation by underlining it with `^^^^^^^^^^^^`. Below that, we get to see the context of our `sqrt` function: First, the path to its module is given (ending in `exceptions/sqrt_raise.py`) and it is pointed out that the exception was raised in line 15. This line of code is then also displayed. It indeed is the one starting with `raise ArithmeticError`. The stack trace therefore shows us exactly where the error happened and from where the code causing the error was called.

After the stack trace, we can see information about the error printed that we passed in: `ArithmeticError: sqrt(inf)is not permitted.` We made this message by ourselves when raising the exception. We did this so that it tells the user that `sqrt` was called with the argument `inf` that we did not permit.

The above information allows us to pretty much identify the source of the problem. It shall be stated here that new programmers often ignore the stack trace. They see that a program produces and error and then try to figure out why by looking at their code. They often do not read the stack trace or the error information below it.

Best Practice 50

The stack trace and error information printed on the Python console in case of an uncaught exception are essential information to identify the problem. They should *always* be read and understood before trying to improve the code. See also Best Practice 15.

In our original `tuple` of inputs that we iteratively passed to `sqrt`, the last three elements are `inf`, `nan`, and `-1.0`. The call to `sqrt` with `inf` as argument was performed and failed. After that, no further output has been generated. Indeed, the control flow has left the `for` loop and the process has been terminated with exit code 1, as the output in Listing 9.3 shows.

Terminating the process because of this one error may seem rash, but it is not. If a programmer used our `sqrt` function incorrectly, then this will force them to fix their error. If the input `inf` was the result of corrupted data, another erroneous computation, or an input mistake by the user, then terminating the program prevented this error from propagating. For both scenarios, the stack trace and error output gives clear information about what went wrong and where.

Earlier in this chapter, we mentioned that it would be a bad idea to be too lenient regarding the datatypes of our input parameters. If we conveniently coerce inputs into types that we like, we might invite users to sloppily call our functions with arbitrary input types – which we would then be forced to support indefinitely in future versions of our code. We should never permit the user to call a function with a type it is not suitable for. Of course, if we would try to invoke our current `sqrt` function with, say, a string as input parameter, it would crash with a `TypeError`. This error would occur at some point because we cannot apply floating point arithmetic to strings.

You may wonder how we, by ourselves, can explicitly create a `TypeError`? And how do we even manually detect whether a parameter has a certain type?

The answer to both of these questions is given in Listing 9.4, sketching file `sqrt_raise_2.py`, another slightly modified variant of our module with the `sqrt` function. In this new variant, we directly, at the beginning of the function, perform the check whether `isinstance(number, float)`. The function `isinstance` accepts an object as first parameter and a type as second parameter. The object we want to check clearly is the parameter `number`. The type we are looking for is `float`. `isinstance(number, float)` returns `True` if `number` belongs to the type `float`. Otherwise, it returns `False`.

Listing 9.4: A new variant of the `sqrt` function which additionally raises a `TypeError` if its parameter is not a `float`. (src)

```
"""A `sqrt` function also raising an error if input is no `float`."""

from math import isclose    # Checks if two float numbers are similar.
from math import isfinite   # A function that checks for `inf` and `nan`.


def sqrt(number: float) -> float:
    """
    Compute the square root of a given `number`.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v == number`.
    :raises ArithmeticError: if `number` is not finite or less than 0.0
    :raises TypeError: if `number` is not a `float`
    """
    if not isinstance(number, float):  # raise error if type wrong
        raise TypeError("number must be float!")
    if (not isfinite(number)) or (number < 0.0):  # raise error
        raise ArithmeticError(f"sqrt({number}) is not permitted.")
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0, negative values were checked above.

    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess    # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

It will not come as a surprise to you that we raise a `TypeError` if *not* `isinstance(number, float)` happens. We also pass in a descriptive string as error message. And we, of course, note this new behavior also in the docstring.

We then write the program `use_sqrt_raise_2.py` in Listing 9.5 that uses our new `sqrt` function pretty much in the same way we did in the previous example. The difference is that we now also try to pass in the string `"0.3"`. This then causes the `TypeError` to be raised, as illustrated in Listing 9.6.

You may wonder: How can we even call `sqrt` with a parameter that is a `str` even though we type-hinted its parameter as `float`? This is because Python does not enforce type restrictions during runtime. We can *write* that this function should only be called by passing in a `float`. But the users of our function can *ignore* this if they want to. ...We just don't let them get away with it by checking the input type in the function.

As a side note, Mypy *does* detect that our new program `use_sqrt_raise_2.py` violates the type hint, as shown in Listing 9.7. If the programmer had checked their program using this tool, they would never have made this error.

We made another step towards professional programming. We now have the means to guard our code and our functions against errors caused by its users (to some degree). Well, we cannot prevent that someone calls our function with a parameter of a wrong type. We also cannot stop anybody from passing corrupted or otherwise faulty data as input to our program. However, if we have some means to detect whether the data is wrong, then we can raise an exception. We can simply refuse to perform GIGO, i.e., to return a result based on obviously wrong inputs. While this is a big step towards better software, we should keep two things in mind:

1. It is not possible to detect *all* faulty input. If the user wanted to invoke `sqrt(3.1)` but by accident invoked `sqrt(1.3)`, we have no way to detect that.

2. Checking input parameters also comes with a performance trade-off. If our `sqrt` function is called millions of times in a loop processing a stream of input data ... do we really want to "manually"

Listing 9.5: Using the new variant of the `sqrt` function from Listing 9.4 that raises a `TypeError` if its input is not a `float`. (stored in file use_sqrt_raise_2.py; output in Listing 9.6)

```
"""Using the sqrt function which raises errors also for wrong types."""

from sqrt_raise_2 import sqrt  # Import our sqrt function.

# Apply our protected square root function to several values.
for number in [0.0, 1.0, 2.0, 4.0, "0.3"]:
    # We get an error when reaching `"0.3"`.
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)

print("The program is now finished.")  # We never get here.
```

↓ `python3 use_sqrt_raise_2.py` ↓

Listing 9.6: The stdout, stderr, and exit code of the program use_sqrt_raise_2.py given in Listing 9.5.

```
√0.0≈0.0
√1.0≈1.0
√2.0≈1.414213562373095
√4.0≈2.0
Traceback (most recent call last):
  File "{...}/exceptions/use_sqrt_raise_2.py", line 8, in <module>
    print(f"\u221A{number}\u2248{sqrt(number)}", flush=True)
                                ~~~~~~~~~~~~
  File "{...}/exceptions/sqrt_raise_2.py", line 17, in sqrt
    raise TypeError("number must be float!")
TypeError: number must be float!
# 'python3 use_sqrt_raise_2.py' failed with exit code 1.
```

Listing 9.7: The results of static type checking with Mypy of the program given in Listing 9.5.

```
$ mypy use_sqrt_raise_2.py --no-strict-optional --check-untyped-defs
use_sqrt_raise_2.py:8: error: Argument 1 to "sqrt" has incompatible type "
    ↪ object"; expected "float"  [arg-type]
Found 1 error in 1 file (checked 1 source file)
# mypy 2.4.1 failed with exit code 1.
```

type-check each value?

In my opinion, it makes a lot of sense to extremely carefully check each parameter in functions which are either slow anyway or which are not often called. The tighter the performance requirements, the more frequently our functions are invoked, the fewer explicit checks can we affort. Safe coding therefore always is a trade-off.

9.3 Built-in Exceptions

A wide variety of things may go wrong during the execution of a computer program. We have already explored a lot of different potential errors, ranging from using an invalid index when accessing a list to dividing a number by zero. In Python, `Exceptions` are raised in such a situation. An `Exception` disrupts the normal control flow and propagates upwards until a corresponding `except` clause is reached. Obviously, we cannot just treat every possible error condition in the same way.

Running out of memory is a completely different situation than trying to read from a non-existent file. Therefore, different types of exceptions can be raised: The former problem causes a `MemoryError` while the latter raises an `FileNotFoundError`. The hierarchy of the different problem types is illustrated in Figure 9.1 [82]. Some exceptions are special cases of other exceptions. For example `ArithmeticErrors` denote general errors during an arithmetic computation, whereas `OverflowErrors` mark such computa-

- `BaseException` — the base `class` for all exceptions
  - `Exception` — situations where reasonable error handling should be possible
    - `ArithmeticError` — an arithmetic operation failed
      - `FloatingPointError` not used by Python, but, e.g., NumPy on invalid floating point operations
      - `OverflowError` — the result of an arithmetic operations is too large, see, e.g., Section 3.3.5
      - `ZeroDivisionError` — a division by zero occurred, see, e.g., Section 8.3
    - `AssertionError` — if an `assert` failed, see, e.g., Section 8.3
    - `BufferError` — a buffer-related operation could not be performed
    - `EOFError` — the end of standard input stream (`stdin`) was reached by `input` without reading data
    - `AttributeError` — when an attribute reference or assignment failed
    - `ImportError` — when an `import` statement fails
      - `ModuleNotFoundError` — a module cannot be loaded
    - `LookupError` — a key or index used on a dictionary or sequence was invalid
      - `IndexError` — a sequence index is out of range, see, e.g., Section 3.6.1
      - `KeyError` — a dictionary key is not found, see, e.g., Section 5.5
    - `MemoryError` — when we are out of memory
    - `NameError` — e.g., when reading an unassigned variable, see, e.g., Listings 4.7 and 9.10
      - `UnboundLocalError` — reference to a local method or function to which no value is bound
    - `OSError` — an operating system function failed
      - `BlockingIOError` a blocking operation is applied to an object set for non-blocking operations
      - `ChildProcessError` — an operation on a child process failed
      - `ConnectionError` — a connection- or pipe-related error
        - `BrokenPipeError` — when trying to write into a pipe whose other end has been closed
        - `ConnectionAbortedError` — connection attempt aborted by peer
        - `ConnectionRefusedError` — connection refused by peer
        - `ConnectionResetError` — connection reset by peer
      - `FileExistsError` — trying to create a file that already exists
      - `FileNotFoundError` — trying to access a file or directory that does not exist
      - `IsADirectoryError` — trying to do a file operation with a directory
      - `NotADirectoryError` — trying to apply a directory operation to a file
      - `PermissionError` — trying to perform an operation without the necessary access rights
      - `ProcessLookupError` — trying to access a process that does not exist
      - `TimeoutError` — an operation timed out
    - `ReferenceError` — a weakly-referenced object is accessed after being garbage collected
    - `RuntimeError` — an error that does not fall into the other categories
      - `NotImplementedError` — a method is not yet implement, but will be later, see, e.g., Chapter 12
      - `PythonFinalizationError` — an operation is blocked during interpreter shutdown
      - `RecursionError` — the maximum recursion depth of functions is reached
    - `StopAsyncIteration` — signals the end of an asynchronous iteration; not an error
    - `StopIteration` — signals the end of an iteration; not an error, see Section 10.1
    - `SyntaxError` — an malformed `Python` file
      - `IndentationError` — incorrectly indented code
        - `TabError` — inconsistent use of tabs and spaces
    - `SystemError` — an internal error of the interpreter
    - `TypeError` — some parameter was of a wrong type or `None`, see, e.g., Section 5.3
    - `ValueError` — a parameter has the right type but an inappropriate value, see, e.g., Listing 9.8
      - `UnicodeError` — an error when dealing with Unicode text
        - `UnicodeDecodeError` — an error occurred when decoding Unicode text
        - `UnicodeEncodeError` — an error occurred when encoding Unicode text
        - `UnicodeTranslateError` — an error occurred when translating Unicode text
  - `GeneratorExit` — when a `Generator` or coroutine terminates; not an error
  - `KeyboardInterrupt` — when the user hits Ctrl+C
  - `SystemExit` — raised by `exit`; not an error

Figure 9.1: An overview of the hierarchy of `Exceptions` in `Python` [82].

tion errors that cause a value to become too large for representation. Therefore, each `OverflowError` is also an `ArithmeticError`.

While these exceptions can be caused by errors during the operations that Python provide, we can also raise them by ourselves. Of course, the documentation of the Python functions tells us which exceptions they could raise. And so should the docstrings of our own code as well as the library functions that we rely on.

9.4 Handling Exceptions

We have learned how we can raise exceptions in order to signal errors. So far, the exceptions that we raised caused our processes to print an error message to the `stderr` stream and then to terminate. Of course, we do not want that all possible unexpected error conditions will immediately terminate our process.

For example, maybe we created a program for painting a picture. A user of our program painted a picture and wants to store it, but accidentally enters a wrong destination. It would be annoying if the program would immediately crash. Our program's function for storing pictures would fail to store the picture at the invalid location. This would probably cause an exception. The current control flow branch, which might do more things like marking the picture as "saved" and clearing the undo-buffer would be aborted by the raising exception. However, at a higher level of abstraction in our program, the exception should be processed. The program would display a dialog with an error message to the user. After acknowledging the error, the user should be able to proceed and use the program as if nothing happened.

There are very many situations like this. Situations where we might want to abort a current computation and signal an error to a higher level of abstraction. In this section, we discuss how errors can be *handled* at such a higher level. We use examples that are specifically constructed to cause certain errors to illustrate how we can deal with them.

9.4.1 The `try...except` Block

The `try`-`except` clause exists as the primary approach to recover from specific errors. We place the code that may raise an exception into an indented block that is prefixed by `try:`. After this block, we write the handlers for specific exception types. For example, if we know that the `try` block could raise an `ArithmeticError` for a reason from which recovery is possible, then we could write `except ArithmeticError as ae:`. The code inside this `except` block would be executed if and only if an `ArithmeticError` was raised somewhere in the `try` block. In this case, the `ArithmeticError` would be available as local variable `ae` in this block. Of course, multiple types of `Exceptions` may be raised, so we can have multiple `except` blocks. This looks like this:

```
"""The syntax of a try-except statement in Python."""

try:  # Begin the try-except block.
    statement 1  # Code that may raise an exception or that
    statement 2  # calls a function that may rise one.
    # ...
except ExceptionType1 as ex1:  # One exception type that can be caught.
    statements processing ex1
    # Code the handles exceptions of type ExceptionType1.
except ExceptionType2:  # Another exception type (optional).
    statements processing exceptions of type ExceptionType2
    # Code the handles exceptions of type ExceptionType2 that are not
    # instances of ExceptionType1. Notice that we do not necessarily
    # need to store the exceptions in variables with, like "as ex2".

next statement  # Executed only if there are no uncaught Exceptions.
```

Let us now try the `try`-`except` block. Back in Section 3.6.1, we learned that text strings offer the method `r.find(s)` that searches a string `s` inside `r` and returns the first index where it is encountered. If `s` cannot be found in `r`, `-1` is returned instead. There exists an operation `r.index(s)`, which works

Listing 9.8: The `index` function of a string raises a `ValueError` if it cannot find the given substring. Here we catch this error in a `try`...`except` block. (stored in file `try_except_str_index.py`; output in Listing 9.9)

```
"""Demonstrate `try...except` by looking for text in a string."""

r: str = "Hello World!"  # This is the string we search inside.

try:  # If this block raises an error, we continue at `except`.
    for s in ["Hello", "world", "!"]:  # The strings we try to find.
        print(f"{s!r} is at index {r.index(s)}.")
except ValueError as ve:  # ValueError is raised if `s` isn't in `text`.
    print(f"Error: {ve}")  # Error, as "world" is not in "Hello World!".

print("The program is now finished.")  # We get here after except block.
```

↓ `python3 try_except_str_index.py` ↓

Listing 9.9: The stdout of the program `try_except_str_index.py` given in Listing 9.8.

```
'Hello' is at index 0.
Error: substring not found
The program is now finished.
```

exactly like `find`, but instead of returning `-1`, it will raise a `ValueError` if `s` cannot be found in `r`. This is quite useful in cases where we know that `s` must be contained in `r` and if it is not, then that is an error. It also allows us to use the result of `r.index(s)` directly as index in `r`, i.e., do something like `r[r.index(s)]`. We cannot do that with the result of `r.find(s)` directly, because `-1` is also a valid index into a string...

In program `try_except_str_index.py` given as Listing 9.8, we explore the new function `index`. Our string `r` is `"Hello World!"`. In the `try`-block, we place a `for` loop which lets a variable `s` iteratively take on three values. In its first iteration, `s = "Hello"` and `r.index(s)` will yield `0`. This is printed to the output. In the second iteration, `s = "world"`, which cannot be found since searching in strings is case-sensitive. `r.index(s)` will therefore raise an exception.

The `except` block after the `try` block is executed if anywhere inside the `try` block a `ValueError` is raised. In this case, this `ValueError` becomes available in variable `ve`. In this block, we simply print the error.

After the block, we print `"The program is now finished."`. This code is executed only if no uncaught exception has left the `try`-`except` block. The output of the program given in Listing 9.9 shows that the program executes as expected. We first get the results of the successful search for `"Hello"`, followed by the output for the failed search. The last line then is `The program is now finished.` If we had not specified the `except` block, the control flow would have left the loop, printed the stack trace, and terminated the program instead.

Notice that the third value in the loop, `"!"`, never gets assigned to the variable `s`. The `try` block is immediately terminated as soon as any exception occurs. If the exception can be handled by a corresponding `except` block, then this block is executed. Otherwise, the whole process will be terminated. Either way, even if a fitting `except` block exists, the code after the failing instruction in the `try` block will not be executed.

We can catch and handle all kinds of `Exceptionss`. However, it is important to only handle reasonable errors. For example, suppose your program should be writing text to a file. It is totally acceptable that this may fail for a variety of file system related reasons, like insufficient space on the device, an access rights violation, or an incorrect file name. Such errors may be handled with a corresponding `except` block and reasonable actions may be taken. If, on the other hand, a `ZeroDivisionError` occurs during our attempt to write the file, then this indicates that something else went really wrong. Such an error is not OK in this context. We should only try to catch errors that are meaningful and that we anticipate in a given context.

Any other error should indeed cause our program to crash. A crashed program is the clearest indicator to the user that something is wrong and that actions on their side are required, after all.

Listing 9.10: The handling of multiple errors, namely `ZeroDivisionError` and `ArithmeticError`, as well as what happens if a variable remains unassigned due to an error (a `NameError` is raised). (stored in file `try_multi_except.py`; output in Listing 9.11)

```
"""Demonstrate `try...except` with multiple exceptions and NameError."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Which error will this produce?
except ZeroDivisionError:  # Did a division by zero happen?
    print(f"We got a division-by-zero error!", flush=True)
except ArithmeticError as ae:  # Or an ArithmeticError?
    print(f"We got an arithmetic error: {ae}!", flush=True)

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not assigned
print("The program is now finished.")  # We never get here.
```

↓ `python3 try_multi_except.py` ↓

Listing 9.11: The `stdout`, `stderr`, and exit code of the program `try_multi_except.py` given in Listing 9.10.

```
We got a division-by-zero error!
Now we try to print the value of sqrt_of_1_div_0.
Traceback (most recent call last):
  File "{...}/exceptions/try_multi_except.py", line 16, in <module>
    print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not
        ↪ assigned
          ^^^^^^^^^^^^^^
NameError: name 'sqrt_of_1_div_0' is not defined
# 'python3 try_multi_except.py' failed with exit code 1.
```

The action could be to call us and to report a bug. Then we can improve the program and fix the bug.

Best Practice 51

Only `Exceptions` should be caught by `except` blocks that we can meaningfully handle [213, 594]. The `except` block is not to be used to just catch any exception, to implement GIGO, or to try to sanitize erroneous input.

In Listing 9.10, we revisit our new `sqrt` function. This function will `raise` an `ArithmeticError` if its argument is non-finite or negative. This time, we want to compute $\sqrt{\frac{1}{0}}$ and thus aim to store the result of `sqrt(1 / 0)` in a variable `sqrt_of_1_div_0`. This is of course complete nonsense, but still – let's see what happens. We first declare the variable as a `float`. Then, in a `try`-`except` block, we perform the actual computation and value assignment: `sqrt_of_1_div_0 = sqrt(1 / 0)`. Knowing that `sqrt` might raise an `ArithmeticError`, we provide a corresponding `except` block. However, we also know that `1 / 0` looks a bit dodgy, as we also try to intercept a potential `ZeroDivisionError`. As you can see, we can have two independent `except` clauses.

So, which one – if any – will be executed? Certainly, $\frac{1}{0}$ is not finite, so `sqrt` would raise an exception. Then again, $\frac{1}{0}$ cannot be computed at all, so maybe we get a `ZeroDivisionError` instead? We find that the `except` block for `ZeroDivisionError` is executed. The reason is that in order to invoke `sqrt(1 / 0)`, the Python interpreter must first compute the result of `1 / 0`. This computation raises a `ZeroDivisionError` and `sqrt` is never called.

This leads us to the question: If `sqrt` is never called, then what will be assigned to `sqrt_of_1_div_0`? Well, actually, `sqrt_of_1_div_0` does not exist. The variable `sqrt_of_1_div_0` would only come to existence if we would store a value in it. Which we do not do. In order to perform the assignment, we would need the result of `sqrt(1 / 0)`. But since this result never becomes available, the assignment is never performed.

This means that when we try to `print(sqrt_of_1_div_0)` after the `try`-`except` block, the *name* `sqrt_of_1_div_0` does not even exist. The variable `sqrt_of_1_div_0` never really received any value at all and thus does not exist. Trying to access it will `raise` a `NameError`.

We did declare the variable with a type hint. But the `Python` interpreter ignores type hints. So it does not know this variable yet, as it does not have a value. Our program will terminate, because the `NameError` is never caught anywhere. It would not make sense to catch such an error, since it could only occur due to a bug. And we indeed committed a bug here! Instead, the stack trace and error information will be printed in Listing 9.11.

Best Practice 52

Remember that, if an exception is raised, be aware that the control flow will immediately leave the current block. The statement in which the exception was raised will not be completed but aborted right away. Therefore, no variable assignments or other side effects can take place anymore and it is possible that variables remain undefined. Remember this when accessing variables that are assigned in a `try`-block.

9.4.2 Exceptions in Exception Handlers

What happens if an exception is raised *inside* an `except` block? We explore this in program `try_except_nested_1.py` given as Listing 9.12. We try to compute `sqrt_of_1_div_0 = sqrt(1 / 0)` twice. First in the `try` block and then again in the `except` block that handles the `ZeroDivisionError` that this will cause. So inside an `except` block for handling `ZeroDivisionErrors`, another `ZeroDivisionError` will be raised.

The result is shown in Listing 9.13: The `except` block is terminated immediately and text is written to the `stderr` indicating why. While we handled the original `ZeroDivisionError`, another error occurred. The output first presents the stack trace of the exception that we were handling. It then informs us that `During handling of the above exception, another exception occurred:`. Then it prints the stack trace of the new exception that occurred inside the `except` block. Since there is no code for handling this error, our process terminates with exit code 1.

It is of course possible that an error that we normally can handle may cause another error. Naturally, it is possible that we want to process and handle such errors. Therefore, like any other `Python` code blocks, we can also nest `try`-`except` blocks. Program `try_except_nested_2.py` in Listing 9.14 shows exactly this. By placing the second `sqrt` invocation located in the `except` block into yet another `try`-`except` block, we can catch the `ZeroDivisionError`.

In this example, we also show two methods to deal with the problem of variable assignments inside `try`-`except` blocks. First, we can assign an initial value to the variable *before* all computations. This value will be overwritten if the code succeeds and remains only if something goes wrong. This guarantees that some value is stored in the variable and that the variable exists, regardless of what happens later. For this purpose, we here imported and used the constant `nan` from the `math` module.

The second, more complicated, choice would be to make sure that all possible branches in the control flow that lead to the code which accesses the variable assign a value to it. This would here mean that the inner-most `except`-block should assign a value, too. We could use `nan` there as well.

Listing 9.12: An example of `Exceptions` being raised inside an `except` block. (stored in file try_except_nested_1.py; output in Listing 9.13)

```
"""Demonstrate `try...except` with an exception raised in `except`."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # This produces a ZeroDivisionError.
except ZeroDivisionError as de:  # Catch an ZeroDivisionError.
    print(f"We got a division-by-zero error: {de}.", flush=True)
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Let's try it again!

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not assigned
print("The program is now finished.")  # We never get here.
```

↓ `python3 try_except_nested_1.py` ↓

Listing 9.13: The stdout, stderr, and exit code of the program try_except_nested_1.py given in Listing 9.12.

```
We got a division-by-zero error: division by zero.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_nested_1.py", line 9, in <module>
    sqrt_of_1_div_0 = sqrt(1 / 0)  # This produces a ZeroDivisionError.
                           ~~^~~
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "{...}/exceptions/try_except_nested_1.py", line 12, in <module>
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Let's try it again!
                           ~~^~~
ZeroDivisionError: division by zero
# 'python3 try_except_nested_1.py' failed with exit code 1.
```

Listing 9.14: An example of nested `try`-`except` blocks, as an improvement over Listing 9.12. (stored in file try_except_nested_2.py; output in Listing 9.15)

```
"""Demonstrate nested `try...except` blocks."""

from math import nan
from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float = nan  # Declare this variable and assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # This produces a ZeroDivisionError.
except ZeroDivisionError as de:  # Catch an ZeroDivisionError.
    print(f"We got a division-by-zero error: {de}.", flush=True)
    try:  # Nesting try-except blocks is totally fine.
        sqrt_of_1_div_0 = sqrt(1 / 0)  # Let's try it again!
    except ZeroDivisionError:  # Another ZeroDivisionError?
        print(f"Yet another division-by-zero error!", flush=True)
        # We could also assign a value to sqrt_of_1_div_0 here, which
        # would work, but we already chose the solution with an initial
        # value before all the try-except blocks.

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Works, because we gave an initial value.
print("The program is now finished.")  # This time, we do get here.
```

↓ `python3 try_except_nested_2.py` ↓

Listing 9.15: The stdout of the program try_except_nested_2.py given in Listing 9.14.

```
We got a division-by-zero error: division by zero.
Yet another division-by-zero error!
Now we try to print the value of sqrt_of_1_div_0.
nan
The program is now finished.
```

9.4.3 The `try...except...else` Block

What can we do if we need the result of a computation inside a `try` block but only can use it if the `try` block completely succeeds? One possible solution in such a situation is the `try`-`except`-`else` block. The difference to the `try`-`except` block is only that an `else` block follows, which is executed if and only if no exception occurred.

```
"""The␣syntax␣of␣a␣try-except-else␣statement␣in␣Python."""

try:␣␣#␣Begin␣the␣try-except␣block.
␣␣␣␣statement␣1␣␣#␣Code␣that␣may␣raise␣an␣exception␣or␣that
␣␣␣␣statement␣2␣␣#␣calls␣a␣function␣that␣may␣rise␣one.
␣␣␣␣#␣...
except␣ExceptionType1␣as␣ex1:␣␣#␣One␣exception␣type␣that␣can␣be␣caught.
␣␣␣␣statements␣processing␣ex1
␣␣␣␣#␣Code␣the␣handles␣exceptions␣of␣type␣ExceptionType1.
#␣...␣maybe␣more␣`except`␣blocks
else:␣␣#␣The␣else␣block␣is␣optional.
␣␣␣␣statement␣3␣␣#␣Code␣executed␣if␣and␣only␣if␣no␣Exception␣occurred.
␣␣␣␣statement␣4
␣␣␣␣#␣...

next␣statement␣␣#␣Executed␣only␣if␣there␣are␣no␣uncaught␣Exceptions.
```

In Program try_except_else.py illustrated as Listing 9.16, we present an example of the `try`-`except`-`else` block. The example is structured a bit similar to Listing 9.10. We again use our own

Listing 9.16: An example of the `try`-`except`-`else` block which is structured a bit similar to Listing 9.10 but avoids the `NameError` by placing the access to the variables into the `else` blocks. (stored in file try_except_else.py; output in Listing 9.17)

```
"""Demonstrate `try...except..else`."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises DivisionByZero, we continue at `except`.
    sqrt_of_1_div_0 = sqrt(1 / 0)  # This is a DivisionByZero!
except ZeroDivisionError as de:  # Catch and print a ZeroDivisionError.
    print(f"We got a division-by-zero error: {de}.", flush=True)
else:  # This code is not executed because an exception occurred.
    print(f"\u221A(1/0)\u2248{sqrt_of_1_div_0}")  # Never reached.

sqrt_3: float  # Declare this variable, but do not assign it.
try:  # If this block raises ArithmeticError, we continue at `except`.
    sqrt_3 = sqrt(3.0)  # This will work just fine and raise no error.
except ZeroDivisionError:  # We catch and print the ZeroDivisionError.
    print(f"We got a another division-by-zero error!", flush=True)
else:  # This code is executed because no exception occurs.
    print(f"\u221A3\u2248{sqrt_3}")  # Always executed.

print("The program is now finished.")  # We do get here.
```

↓ `python3 try_except_else.py` ↓

Listing 9.17: The stdout of the program try_except_else.py given in Listing 9.16.

```
We got a division-by-zero error: division by zero.
√3≈1.7320508075688772
The program is now finished.
```

`sqrt` function, this time attempting to compute $\sqrt{\frac{1}{0}}$ and then $\sqrt{3}$, each within its own `try`-block. Back in Listing 9.10, we got a `NameError` because we wanted to access the value of a variable that did not exist. The variable would have been created via the value assignment in the `try` block. The `try` block failed. While we did catch and process the `ZeroDivisionError` in an `except` block, the variable was never assigned. So accessing the variable later cause the error.

This time, we place the code for accessing the variables into `else` blocks. This is the third solution to accessing variable values that are assigned inside a `try`. The *else* blocks are *only* executed if the `try` block succeeds. Hence, the variables are guaranteed to be exist and have values properly assigned to them if the `else` blocks are reached. In case of $\sqrt{\frac{1}{0}}$, the `else` block is not reached. In case of $\sqrt{3}$, it is and its corresponding `print` instruction is executed. Listing 9.16 shows that this program can properly finish.

9.4.4 The `try...finally` Block

We do know that errors may occur in a piece of code. These may be errors that we can reasonably expect to potentially happen. Such errors we will process with corresponding `except` blocks. Then, there might be unanticipated errors for which we cannot define a reasonable `except` block. In the latter case, the program should terminate and the stack trace should be printed.

However, there may be situations where we do not just want to immediately quit. Often, we want to perform some necessary actions before termination, even if an "unreasonable" error occurs. A typical example of this is if we are currently writing contents to a file. Let's say we are writing a table of data row by row into a text file. Suddenly some unexpected error occurs, maybe say something obscure like a `ReferenceError`. This error *should* lead to the termination of our process. But if we terminate immediately without closing the file, then all the contents of the complete file could be lost – including the successfully written data. If we close the file before terminating, then at least the data that was successfully written so far will be preserved. By terminating the process afterwards, we would still indicate to the user that there is some serious problem that needs attention. But at least we would not destroy the data that was correctly produced.

For this purpose, the `try`-`finally` block exists. Basically, we can define a `finally` block that contains the code that should *always* be executed. Of course, *always* only holds as long as our Python interpreter is correctly running, i.e., if we turn off the power or kill the interpreter via the task manager, then `finally` blocks cannot be guaranteed to be executed... Anyway. We still need a `try` block that contains that code that may cause an error. Then follows a `finally` block whose code is executed regardless of whether or not an error occurred in the `try` block. In between the two blocks, we can optionally add `except` blocks to handle certain `Exceptions` and an `else` block to be executed if no error occurs. Notice that the order of the blocks cannot be changed, i.e., first comes the `try` block, then at least one of the `except`, `else`, and `finally` blocks, in that order.

```
"""The syntax of a try-except-else-finally statement in Python."""

try:  # Begin the try-except block.
    statement 1  # Code that may raise an exception or that
    statement 2  # calls a function that may rise one.
    # ...
except ExceptionType1 as ex1:  # The except blocks are optional here.
    statements processing ex1
    # Code the handles exceptions of type ExceptionType1.
# ... maybe more `except` blocks
else:  # The else block is optional.
    statement 3  # Code executed if and only if no Exception occurred.
    statement 4
    # ...
finally:  # The optional finally block.
    statement 5  # This code will always be executed, even if there are
    statement 6  # uncaught exceptions.

next statement  # Executed only if there are no uncaught Exceptions.
```

Let us explore this in another artificial example, program `try_except_else_finally.py` in Listing 9.18. Here, we create the function `divide_and_print` that accepts two parameters `a` and `b` which can either be integers or floating point numbers. In a `try` block, the function attempts to divide `a` by `b` and to `print` the result using an f-string. Since we do not know exactly what values `b` can take on beforehand, we anticipate that a `ZeroDivisionError` may occur. In the corresponding `except` block, we would then print a message that explains the situation. In this case, the division result would not be printed because the `try` block would terminate when the f-string is interpolated.

We also attempt to catch a possible `TypeError`. Such an error would occur if the function is invoked by an argument that is neither an `int` nor a `float` and that does not support divison. This is a typical example of an error that we should *not* attempt to catch. This error could only appear if another programmer was using our function incorrectly. We only process this error here for the sake of the example and print an appropriate message in the `except` block.

In the example, we also have an `else` block which notifies us that no error occurred. The code in this block is only executed if no `ZeroDivisionError` and no `TypeError` and also no other exception was raised.

We finish the division and error handling part of the function with a `finally` block. This block will be executed if no exception was raised anywhere, but also if a `ZeroDivisionError`, `TypeError`, or other exception were raised. Even if an error was raised *inside* one of the `except` blocks, this code would be executed. It will print that the division code was completed.

Then, in the last line of the function after the whole `try`-`except`-`else`-`finally` blocks, we print yet another message. This code outside the blocks is reached only if either no error occurred at all or if the error was handled by one of the two `except` blocks (without yet another error).

In Listing 9.19, we show the `stdout` and `stderr` for invoking this function with several different arguments. For `divide_and_print(10, 5)`, the division result in the `try` block as well as the messages from the `else` block, the block, and from the very end of our function are printed. `divide_and_print(3, 0)` will cause a `ZeroDivisionError`. Therefore, the `print` instruction in the `try` block is not invoked as interpolating the f-string already fails. The first `except` block, which handles the `ZeroDivisionError`, is executed and prints its message. The `else` block is not reached but the `finally` block prints its message. Since the error was properly handled, the message in the `print` instruction at the end of the function is written to the output as well.

Invoking `divide_and_print("3", 0)` means that we intentionally ignore the type hints in the function definition. The `Python` interpreter allows us to do this without complaint, as type hints are only hints and not strict requirements (different from other programming languages like `Java` or `C`). However, the division `a / b` will fail since the string `"3"` does not support a division operation. It raises a `TypeError`, which is subsequently caught by our second `except` block. This means that the `else` block is again not reached. The `finally` block is executed still. Since the `TypeError` was properly handled, the `print` at the bottom of our function is executed as well. By the way, had we applied the static type checker `Mypy` to Listing 9.18, it would have informed us that we here try to call `divide_and_print` with an invalid argument, as shown in Listing 9.20.

Finally, we attempt to compute `divide_and_print(10 ** 313, 1.0)`, i.e., to calculate $\frac{10^{313}}{1}$. At first glance, this looks totally fine. However, back in Section 3.3.5, we learned about the limits of the datatype `float`. Indeed, the `int` 10^{313} is outside the range of numbers that a `float` can represent. By trying to divide it by `1.0`, we force it to be converted to `float` first. This will raise an `OverflowError`. We do not have an `except` block for handling `OverflowErrors`. This means that, of course, no message is printed by the `try` block and none of our `except` blocks are reached. The `else` block is not executed either. The `finally` block, however, *is* executed and prints its message to the output.

Since we did not catch the `OverflowError`, the code after our blocks at the bottom of our function is not executed. Instead, our function is terminated immediately after the `finally` block completes. Since there is no `try`-`except` block able to catch `OverflowErrors` wrapped around the function call, the whole `Python` interpreter terminates as well. It again prints the stack trace, which informs us which error occured and where it happened.

Listing 9.18: An example of the `try-except-else-finally` block. (stored in file try_except_else_finally.py; output in Listing 9.19)

```
"""Demonstrate the try-except-else-finally clause."""


def divide_and_print(a: int | float, b: int | float) -> None:
    """
    Divide the numbers `a` and `b` and print the result.

    :param a: the dividend
    :param b: the divisor
    """
    try:  # We try to do some computation that may cause an Exception.
        print(f"{a / b = }", flush=True)  # Divide and print.
    except ZeroDivisionError as zd:  # Is b == 0 ?
        print(f"We got a ZeroDivisionError when doing {a} / {b}: {zd}.")
    except TypeError as te:  # Has one of the values the wrong type?
        print(f"We got a TypeError when computing {a} / {b}: {te}.")
    else:  # Only called when no Exception occurred.
        print(f"No error occurred when computing {a} / {b}.")
    finally:  # This code is always called.
        print(f"We are finally done with division-by-{b}.", flush=True)
    print(f"This code comes after all the {a} by {b} division code.")


divide_and_print(10, 5)           # This works just fine.
divide_and_print(3, 0)            # This yields a ZeroDivisionError.
divide_and_print("3", 0)           # This yields a TypeError.
divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError!
```

↓ `python3 try_except_else_finally.py` ↓

Listing 9.19: The stdout, stderr, and exit code of the program try_except_else_finally.py given in Listing 9.18.

```
a / b = 2.0
No error occurred when computing 10 / 5.
We are finally done with division-by-5.
This code comes after all the 10 by 5 division code.
We got a ZeroDivisionError when doing 3 / 0: division by zero.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We got a TypeError when computing 3 / 0: unsupported operand type(s) for /:
    ↪ 'str' and 'int'.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We are finally done with division-by-1.0.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_else_finally.py", line 27, in <module>
    divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError!
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  File "{...}/exceptions/try_except_else_finally.py", line 12, in
      ↪ divide_and_print
    print(f"{a / b = }", flush=True)  # Divide and print.
              ~~~~~

OverflowError: int too large to convert to float
# 'python3 try_except_else_finally.py' failed with exit code 1.
```

Listing 9.20: The results of static type checking with Mypy of the program given in Listing 9.18.

```
$ mypy try_except_else_finally.py --no-strict-optional --check-untyped-defs
try_except_else_finally.py:26: error: Argument 1 to "divide_and_print" has
    ↪ incompatible type "str"; expected "int | float"  [arg-type]
Found 1 error in 1 file (checked 1 source file)
# mypy 2.4.1 failed with exit code 1.
```

9.4.5 The `with` Block and Context Managers

The `try`-`finally` block allows us to make sure that one action will always be performed, regardless of whether some other intermediate code fails (and raises exceptions). A use case for this is handling resources that need to be explicitly closed or freed. Typical examples of this are network connections or file Input/Output (I/O).

Let us use file I/O as an example in program `file_try_finally.py` given as Listing 9.21. In this program, we will create and open a text file with the name `example.txt`. We will write a line of text into the file and then close it. Then we will open it again, read the text, and print it to the `stdout` stream. At the end, we will delete the file to not leave it laying around.

To implement these steps, we first import the necessary types and functions. We import the type `IO` from the `typing` module. `IO` is the basic type for text-based I/O streams and we will later use it as type hint. We will also need the function `remove` from the module `os`.

We begin by opening the file `example.txt` for writing. To do so, we call the built-in function `open`. We pass in the filename `"example.txt"` as first parameter. The second parameter, `mode`, is set to `"w"`, which means "open for writing." If the file does not yet exist, then the `"w"` tells `open` to automatically create it. If the file already exists, its contents will be discarded and we begin writing at its beginning.

The parameter `encoding` is set to `"UTF-8"`, which indicates that the text should be translated to binary form when it is stored in the file via the usual UCS Transformation Format 8 (UCS) encoding [285, 648]. All stored data is binary and this is the most common format to store text in binary form in the internet.

Anyway, if opening the text file succeeds, we now have a variable `stream_out`, which is an instance of `IO`. We must make sure to definitely close this so-called text stream again, regardless what happens from now on. We know that this can be done with a `try`-`finally` statement. We simply put `stream_out.close()` into the `finally` block. It will thus definitely be called and we thus ensure that `stream_out` will be closed.

Into the `try` block, we put `stream_out.write("Hello world!")`. This line will write the string `"Hello world!"` to the file. This could, of course, fail. Maybe our hard disk does not have enough space left to store this string. But even if it fails, the `finally` block will make sure to close

Listing 9.21: Using `try`-`finally` for closing files after writing to and reading from them. (stored in file `file_try_finally.py`; output in Listing 9.22)

```
"""Use the `try-finally` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.
from typing import IO  # IO is the text stream object

# We open the text file "example.txt" for writing.
stream_out: IO = open("example.txt", mode="w", encoding="UTF-8")
try:  # If we succeed opening the file for writing, then...
    stream_out.write("Hello world!")  # ...we write one line of text
finally:                # ...and we will definitely get here....
    stream_out.close()  # and close the file again.

# We now open the text file "example.txt" for reading.
stream_in: IO = open("example.txt", encoding="UTF-8")
try:  # If we succeed opening the file for reading, then...
    print(stream_in.readline())  # ...we read one line of text
finally:               # ...and we will definitely get here....
    stream_in.close()  # and close the file again.

remove("example.txt")  # Delete the file "example.txt".
```

↓ `python3 file_try_finally.py` ↓

Listing 9.22: The `stdout` of the program `file_try_finally.py` given in Listing 9.21.

```
Hello world!
```

Listing 9.23: The output of Ruff when applied to Listing 9.21: It suggests using context managers instead of `try`-`finally`.

```
$ ruff check --target-version py312 --select=A,AIR,ANN,ASYNC,B,BLE,C,C4,COM
    ↪ ,D,DJ,DTZ,E,ERA,EXE,F,FA,FIX,FLY,FURB,G,I,ICN,INP,ISC,INT,LOG,N,NPY,
    ↪ PERF,PIE,PLC,PLE,PLR,PLW,PT,PYI,Q,RET,RSE,RUF,S,SIM,T,T10,TD,TID,TRY,
    ↪ UP,W,YTT --ignore=A005,ANN001,ANN002,ANN003,ANN204,ANN401,B008,B009,
    ↪ B010,C901,D203,D208,D212,D401,D407,D413,INP001,N801,PLC2801,PLR0904,
    ↪ PLR0911,PLR0912,PLR0913,PLR0914,PLR0915,PLR0916,PLR0917,PLR1702,
    ↪ PLR2004,PLR6301,PT011,PT012,PT013,PYI041,RUF100,S,T201,TRY003,UP035,W
    ↪  --line-length 79 file_try_finally.py
SIM115 Use a context manager for opening files
 --> file_try_finally.py:7:18
6 | # We open the text file "example.txt" for writing.
7 | stream_out: IO = open("example.txt", mode="w", encoding="UTF-8")
  |                  ^^^^
8 | try:  # If we succeed opening the file for writing, then...
9 |     stream_out.write("Hello world!")  # ...we write one line of text
SIM115 Use a context manager for opening files
  --> file_try_finally.py:14:17
13 | # We now open the text file "example.txt" for reading.
14 | stream_in: IO = open("example.txt", encoding="UTF-8")
   |                 ^^^^
15 | try:  # If we succeed opening the file for reading, then...
16 |     print(stream_in.readline())  # ...we read one line of text
Found 2 errors.
# ruff 0.17.3 failed with exit code 1.
```

Listing 9.24: The output of Pylint when applied to Listing 9.21: It suggests using the `with` statement instead of `try`-`finally`.

```
$ pylint file_try_finally.py --disable=C0103,C0302,C0325,R0801,R0901,R0902,
    ↪ R0903,R0911,R0912,R0913,R0914,R0915,R1702,R1728,W0212,W0238,W0703
************* Module file_try_finally
file_try_finally.py:7:17: R1732: Consider using 'with' for resource-
    ↪ allocating operations (consider-using-with)
file_try_finally.py:14:16: R1732: Consider using 'with' for resource-
    ↪ allocating operations (consider-using-with)
-----------------------------------
Your code has been rated at 8.18/10
# pylint 4.1.3 failed with exit code 8.
```

the file.

So after the block, the file is closed. We could now open it in a text editor and would find in there the text that we had written. Instead, we want to use code to read the text again right away.

For this purpose, we open the file again, this time for reading. This works exactly as opening for writing, by using the `open` function. Instead of supplying `mode="w"`, we could write `mode="r"`, meaning "open for reading." However, `"r"` is the default value for the parameter `mode`, so we can just omit it. Therefore, `stream_in = open("example.txt", encoding="UTF-8")` it is. Once the file is opened for reading, we must again make sure to close it eventually. We do this again with `try`-`finally` statement, where we put `stream_in.close()` into the `finally` block.

The line of text that we had written before can now be read in the `try` block. This is done via `stream_in.readline()`. And, as you can see, it gets immediately written to the `stdout` stream via `print`.

At the end of our program, we delete the file `example.txt` by calling `remove("example.txt")`. We had imported the `remove` function from the module `os` exactly for this purpose.

The output of our program, given in Listing 9.21 looks exactly as expected. Since we are diligent

programmers, we will, of course, also perform static code analysis by using tools such as Ruff and Pylint. Their output can be found in Listings 9.23 and 9.24. Oddly enough, they complain: Ruff suggests using a "context manager" instead of the `try`-`finally` statement. Pylint suggests to go with a `with` statement instead. They both mean the same.

> A context manager is an object that defines the runtime context to be established when executing a `with` statement. The context manager handles the entry into, and the exit from, the desired runtime context for the execution of the block of code. [...] Typical uses of context managers include saving and restoring various kinds of global state, locking and unlocking resources, closing opened files, etc.
>
> — *[639], 2001*

The `with` statement has the following syntax. Here the `expression` is an expression that returns a so-called context manager [128, 566, 590, 639] object. We did not yet discuss classes, so right now, we are still lacking some background knowledge needed to discuss what a context manager exactly is or how it works.[3] However, in a nutshell, a context manager is basically an object that has two special methods. The first one will be called right at the beginning of the `with` block, and its result will be stored in a local `variable` (if the `with` block has the `with expression as variable:`-shape). The second special method will be called after the end of the `with` block. This is always done, regardless of whether an exception occurred inside the indented block of code directly under `with`.

```
"""The syntax of with-blocks in Python."""

# The 'expression' is some expression that usually acquires a resource
# which must eventually be released.
# The 'with' block does this automatically, even if an uncaught
# exception is raised within it.
# It is basically a fancy try-finally block.

with expression as variable:
    # 'variable' here usually stores the acquired resource.
    # It could be a handle to a file opened for writing, for example.
    statement 1  # block of code that works with variable.
    statement 2
    # ...

# or

with expression:
    # Here we do not explicitly work with the resource that was
    # acquired. This makes sense, for example, if the resource is a lock
    # to a critical section.
    statement 3  # block of code
    statement 4
```

This makes the syntax roughly equivalent to calling the first special method before a `try` block and the second special method in the corresponding `finally` block. It is just much shorter and more elegant.

Many of Python's resource-related APIs are realized as context managers. This also holds for the file I/O API, just as the linters told us. We now rewrite Listing 9.21 using `with` blocks as `file_with.py` in Listing 9.25. The first thing you will notice is that the file is much shorter. It is now 13 lines of code instead of 20. It is also much easier to read and clearer. We do no longer need to call the `close` methods of the streams. They will automatically be invoked at the end of the `with` statements' bodies.

But one thing after the other. Reading our new program from top to bottom, you realize that we no longer import the type `IO`. The reason is that the syntax of the `with` block, whose first line ends with a colon `:`, does not permit us to type-hint the variables. Therefore, we also do not need to import the datatype. (We would have to rely on Mypy to figure it out by itself...) We still import the function `remove` for deleting files.

[3] We will therefore do this later in Section 13.5.

Listing 9.25: Using a `with` block for closing files after writing to and reading from them. (stored in file `file_with.py`; output in Listing 9.26)

```
"""Use the `with` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.

# We open the text file "example.txt" for writing.
with open("example.txt", mode="w", encoding="UTF-8") as stream_out:
    stream_out.write("Hello world!")  # Write one line of text.

# We now open the text file "example.txt" for reading.
with open("example.txt", encoding="UTF-8") as stream_in:
    print(stream_in.readline())  # Read one line of text.

remove("example.txt")  # Delete the file "example.txt".
```

↓ `python3 file_with.py` ↓

Listing 9.26: The stdout of the program `file_with.py` given in Listing 9.25.

```
Hello world!
```

Then we begin our first `with` block. Its purpose is to create the file and to write the text into it. We put the same call to the `open` function as before, and store its result in variable `stream_out` by writing `as stream_out`. Inside the `with` block, we now place the code that formerly was in the `try` block, i.e., the `stream_out.write(...)`. We can rely on the `with` block to close the stream after its end.

Therefore, we can directly begin with the second `with` block. Here we want to read the single line of text from the file. We specify the call to `open` corresponding to this purpose and remember its result in variable `stream_in`. The body of this `with` statement then does the `stream_in.readline()`. After its end, the stream is automatically closed. And then, the file gets deleted via `remove`.

9.4.6 Summary

Now we can both create and handle exceptions. This allows us to construct code which makes sure that its inputs and outputs are correct. At the same time, we can gracefully handle anticipated errors. The important fact is that we do both things *explicitly* and *clearly*. We clearly express what conditions will cause our code to raise exceptions, e.g., in the docstrings. We explicitly write down which exceptions we can process. Finally, we can also write robust code that makes sure that resources are properly closed or freed, even if unexpected errors occur.

9.5 Interlude: Testing for Exceptions

Back in Section 8.3, we introduced the concept of unit tests. Together, we explored how the tool `pytest` can be used to test our functions. We stated in Best Practice 46 that we should cover all the branches of the control flow inside a function with unit tests. One kind of branch that is often overlooked are `Exception` and exception handling [357].

If our function is supposed to `raise` a certain exception under some conditions, then we should have a unit test that checks if this `Exceptions` is actually raised as it should be. Now, any exception raised by a unit test will cause the test to fail. This seems to contradict our goal to intentionally raise the exceptions. Luckily, `pytest` offers us a device for this.

The `pytest` module offers a context manager [128] named `raises`. We just learned how we can use such context managers via the `with`-statement. If we want to check whether a function indeed raises a certain exception of type `ExceptionType` for a given input, then we can wrap the corresponding function call into `with raises(ExceptionType):`. This block tells `pytest` that in the following indented block an exception of type `ExceptionType` must be raised. If such an exception is not raised, the test fails. If it is raised, the test succeeds.

We also have learned that we can provide a error message as parameter when we raise an exception. The `raises` context manager also allows us to compare the string representation of the exception (which is usually equivalent to that error message) to a certain regular expression (regex). We therefore provide the regex as parameter `match`. Then, the unit test will fail either if no exception of type `ExceptionType` is raised *or* if such an exception is raised, but its error message does not match the regex in `match`.

```
"""The syntax of the `raises` context manager offered by `pytest`."""

from pytest import raises  # Needed checking that exceptions are raised.

# `raises` is a context manager that will cause the test to fail if no
# exception of type `ExceptionType` is raised.
with raises(ExceptionType):
    code that should raise ExceptionType

# We can optionally provide a regular expression with parameter `match`.
# If either no exception of type `ExceptionType` is raised > OR < if the
# string-representation of the exception (usually corresponding to the
# error message) does not match to this regex, then the test will fail.
with raises(ExceptionType, match="error message regex"):
    code that should raise ExceptionType with fitting error message
```

But what is a regex? Regexes offer a simple programming language for specifying text patterns that can be compared with strings. Regexes are supported by a very wide range of tools and programming

Listing 9.27: A unit test checking that our new variant of the `sqrt` function given in `sqrt_raise_2.py` in Listing 9.4 properly raises an `ArithmeticError` if its input is non-finite or negative and a `TypeError` if the input type does not fit. (src)

```
"""Testing our sqrt function that raises an error for invalid inputs."""

from math import inf, nan  # some maths constants

from pytest import raises  # Needed checking that exceptions are raised.

from sqrt_raise_2 import sqrt  # Import our new sqrt function.


def test_sqrt() -> None:
    """Test the `sqrt` function on normal input values."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10    # 1e10^2 = 1e10 * 1e10


def test_sqrt_raises_arithmetic_error():
    """Check that `ArithmeticError` is properly raised."""
    for number in [-1.0, inf, -inf, nan]:  # negative or not finite...
        with raises(ArithmeticError, match="sqrt.* is not permitted."):
            sqrt(number)  # The square root of `number` is not defined.


def test_sqrt_raises_type_error():
    """Check that `TypeError` is properly raised."""
    for number in [True, "x", None]:  # all of these are NOT `float`s.
        with raises(TypeError, match="number must be float!"):
            sqrt(number)  # non-float values are not permitted.
```

Listing 9.28: The output of the unit tests in Listing 9.27: The `ArithmeticErrors` and `TypeErrors` are correctly raised.

```
$ pytest --timeout=10 --no-header --tb=short test_sqrt_raise_2.py
========================== test session starts ==========================
collected 3 items
test_sqrt_raise_2.py ...                                           [100%]
=========================== 3 passed in 0.01s ===========================
# pytest 9.2.0 with pytest-timeout 2.4.0 succeeded with exit code 0.
```

languages. We cannot really discuss them here in-depth, but only provide some very very few examples. In the simplest case, a regex can be a normal string, like `"hello"`. When this string is matched against another string stored in variable `x`, then it matches only if `x` is exactly equal to `"hello"` and does not match otherwise.

However, there are some special characters that make regexes more powerful. For example, the dot `.` stands for any character. The regex `"h.llo"` would match to `"hello"`, `"hallo"`, and `"hXllo"`, or `"h llo"`. The star (`*`) says that the sub-pattern right in front of it can appear zero, one, or multiple times. In the simplest case, the sub-pattern is a single character. The regex `"he*llo"` matches to `"hllo"`, `"hello"`, `"heello"`, `"heeello"`, and so on. The regex `"h.*llo"` matches to `"hllo"`, `"hello"`, `"hallo"`, `"heeeXYZeeeello"`, and so on.

There are many more patterns that we can form with regexes and there are also more special characters that we can use for that. However, for now, we will leave it at that. Feel free to explore [283, 331, 407, 411, 473] for more information. Still, I think that you can already see how using regexes to check whether error messages conform to a certain structure is a very natural idea.

Let us expand our Listing 8.14 for testing the `sqrt` function. This time, we want to test the last variant of our `sqrt` function developed as `sqrt_raise_2.py` in Listing 9.4. This function was an improved variant of our original `sqrt` implementation. It will raise a `TypeError` if its argument is not a `float` with a fixed string as error message. It will raise an `ArithmeticError` if its argument is either negative or not a finite number and its error message will contain the offending value. Of course, we need to be sure that it actually does that. Therefore, we now want to also test for these `ArithmeticErrors` and `TypeErrors`.

In program `test_sqrt_raise_2.py` given in Listing 9.27, we implement a more comprehensive set of unit tests. In the first test function, `test_sqrt`, we perform the same tests for "normal" arguments as back in Listing 8.14. Of course, we still have to test that our function behaves as expected for valid arguments.

In the second test function, `test_sqrt_raises_arithmetic_error`, we check arguments that are floating point numbers but have illegal values. In a `for` loop, we let a variable `number` iterate over the values `[-1.0, inf, -inf, nan]`. The first value is negative and the others are not finite. The sanity check `(not isfinite(number))or (number < 0.0)` inside our function should therefore catch these values, unless we made an error and it somehow does not do what we thing it should be doing. Thus, all of these values should cause our `sqrt` function to raise an `ArithmeticError`. The error message stored in this exception would then be constructed using the f-string `f"sqrt({number})is not permitted."`.

Inside the loop in the test function, we wrap the actual function calls into a `with` block using the `raises` context manager. Here, we specify the expected exception class to be `ArithmeticError`. As pattern to match the string representation, i.e., error message, of the exception against, we provide `"sqrt.* is not permitted."`.

In summary, this requires that *each* `sqrt`-call in the loop must raise an `ArithmeticError`. The error message stored in the exception object should begin with `sqrt` and end with `is not permitted.`. In between these two string fragments, an arbitrary number of arbitrary characters can be contained. In other words, `"sqrt.* is not permitted."` would match against `"sqrt is not permitted."`, `"sqrt(1)is not permitted."`, `"sqrt(inf)is not permitted."`, but not `"sqrt is wrong."`. Even if only for one value of `number` no such exception is raised, the test will fail.

Finally, in our test function `test_sqrt_raises_type_error`, we check for `TypeErrors` being raised. We use exactly the same pattern as in the previous test function. In a `for` loop, we let a variable `number` iterate over the values `[True, "x", None]`. Obviously, none of these values are of

Listing 9.29: Two unit tests checking the original variant of the `sqrt` function from back in Listing 8.11 that does not raise errors. (src)

```
"""Testing the square root implementation that does not raise errors."""

from math import isclose   # Checks if two float numbers are similar.
from pytest import raises  # Expects that a certain Exception is raised.


def sqrt(number: float) -> float:
    """
    Compute the square root of a `number`, but do not raise errors.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v == number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess


def test_raises_arithmetic_error_1():
    """Check that `ArithmeticError` is raised for negative input."""
    with raises(ArithmeticError):  # We can also test without `match`.
        sqrt(-1.0)  # This is not permitted, but no Exception is raised.


def test_raises_overflow_error():
    """Check that large integers cause `OverflowError`s."""
    with raises(OverflowError, match="int too large.*"):  # This works.
        sqrt(10 ** 320)  # Raises OverflowError with right message.


def test_raises_arithmetic_error_2():
    """Check that large integers cause `ArithmeticError`."""
    with raises(ArithmeticError):  # ArithmeticError, any message
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors -> OK.


def test_raises_arithmetic_error_3():
    """Check that an Arithmetic error with (wrong) message is raise."""
    with raises(ArithmeticError, match="sqrt.* is not permitted."):
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors.
```

type `float`. The check `isinstance(number, float)` should prevent these values from entering the actual computation. Unless, of course, we misunderstood how `isinstance` works. If our function `sqrt` works as we think it should, it should raise a `TypeError` with the error message `"number must be float!"`. We therefore place these function calls into a `with` block with context manager `raises(TypeError, match="number must be float!")`.

We now run `pytest` as usual. As you can see in the output given as Listing 9.28, all three tests pass. This means that our `sqrt` function returns the expected results for normal computations. It also means that it raises an `ArithmeticError` with the appropriate message for floating point values that we do not permit. And it also means that it does raise a `TypeError` with appropriate message if we pass in arguments that are not floating point numbers. We can be confident that our implementation is correct.

Listing 9.30: The output of the unit tests in Listing 9.29: No error was raised in it, so the first test fails. The error message in the second test does not fit, so it fails as well.

```
$ pytest --timeout=10 --no-header --tb=short test_sqrt.py
========================== test session starts ==========================
collected 4 items
test_sqrt.py F..F                                                  [100%]
============================== FAILURES ==============================
____________________ test_raises_arithmetic_error_1 ____________________
test_sqrt.py:26: in test_raises_arithmetic_error_1
    with raises(ArithmeticError):  # We can also test without `match`.
         ^^^^^^^^^^^^^^^^^^^^^^^
E   Failed: DID NOT RAISE ArithmeticError
____________________ test_raises_arithmetic_error_3 ____________________
test_sqrt.py:44: in test_raises_arithmetic_error_3
    with raises(ArithmeticError, match="sqrt.* is not permitted."):
         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
E   AssertionError: Regex pattern did not match.
E     Expected regex: 'sqrt.* is not permitted.'
E     Actual message: 'int too large to convert to float'
======================== short test summary info ========================
FAILED test_sqrt.py::test_raises_arithmetic_error_1 - Failed: DID NOT RAISE
    ↪ ArithmeticError
FAILED test_sqrt.py::test_raises_arithmetic_error_3 - AssertionError: Regex
    ↪ pattern did not match.
  Expected regex: 'sqrt.* is not permitted.'
  Actual message: 'int too large to convert to float'
====================== 2 failed, 2 passed in 0.02s ======================
# pytest 9.2.0 with pytest-timeout 2.4.0 failed with exit code 1.
```

We now want to test a few more aspects of the `raises` context manager offered by module `pytest`. We do this in Listing 9.29 presenting file `test_sqrt.py`. This time, we use basically the original version of our `sqrt` function which does not by itself raise any exception. This `sqrt` function is directly copied into the file together with four unit tests, purely for the sake of simplicity.

The test `test_raises_arithmetic_error_1` simply feeds the value `-1.0` to `sqrt`. It expects that an `ArithmeticError` should be raised. It does not provide a specific `match` regex against which the error message should be compared. The output of `pytest` given in Listing 9.30 shows us that this test fails. The reason is that the old implementation `sqrt` simply returns `-1` for negative arguments. It does not raise any error explicitly. Since the function call inside the `with raises(ArithmeticError):` does not actually raise an `ArithmeticError`, the test fails.

In the second test case, `test_raises_overflow_error`, we invoke `sqrt(10 ** 320)`. We place it into a `with` block using context manager `raises(OverflowError, match="int too large.*")`. This means that we expect that the code should raise an `OverflowError`. The exception object needs to hold a message that starts with `int too large` and after that can contain arbitrary additional text.

This indeed happens: The code in the function does floating point arithmetic. If we pass in an integer, it will, at some point, be converted to a `float`. However, a very large integer like 10^{320} is too big to be converted, as we just recently saw in Section 9.4.4. This failure results in an `OverflowError`. The error message stored in that exception object also matches to the text pattern we provide. So this second test will pass.

Then, in test `test_raises_arithmetic_error_2`, we wrap the exact same function call into a `with raises(ArithmeticError):`. We just confirmed that `sqrt(10 ** 320)` will raise an `OverflowError`. Yet, this test passes. The reason is that an `OverflowError` is a *special case* of `ArithmeticError`. We here did not even require a specific error message, as we did not specify the `match` argument. Therefore, the tests asks for an `ArithmeticError`, sees a special case of that, and is happy. The third test thus also passes.

In the fourth test, `test_raises_arithmetic_error_3`, we again use `sqrt(10 ** 320)`. This time, specify the context manager `with raises(ArithmeticError, match="sqrt.* is not permitted.")`. This is just the same condition that we had back in Listing 9.27 when we tested our `sqrt` func-

tion that *does* explicitly raise `ArithmeticErrors`. Of course, the `sqrt` function we are testing here does no such thing. Still, `sqrt(10 ** 320)` does raise an `OverflowError`. We know that `OverflowErrors` are also special `ArithmeticErrors`. However, we also do know that the error message they contain most certainly does not fit to our required `match` regex `"sqrt.* is not permitted."`. Therefore, this unit test fails. The output of Listing 9.30 explains this clearly.

With `pytest`, we can now therefore:

1. Test whether a function computes the expected output for selected (correct) inputs. If it does not, the test fails.
2. Test whether a function does not raise an unexpected exception for the selected (correct) inputs. If it does raise one, the test fails.
3. Raises the expected `Exceptions` for selected (wrong) inputs with the correct error messages. If it does not, the test fails. We do this by using the `with raises(...)` statement.

This allows us to cover both the expected and correct use of our function with tests as well as unexpected incorrect uses. We can then be confident that our code is unlikely to cause harm, neither due to bugs created by ourselves nor due to accidental misuse by other programmers due to misunderstandings (which would be immediately signaled to them by `Exceptions`). Notice that good unit tests go hand in hand with good documentation in docstrings, because good docstrings reduce the chance of such misunderstandings.

Best Practice 53

It is important to cover both the reasonable expected use of our functions as well as unexpected use with incorrect arguments with test cases. The latter case should raise `Exceptions`, which we should verify with unit tests.

9.6 Summary

In this chapter, we have dealt with a very important subject in programming: How we handle errors. Errors can arise from a wide variety of reasons.

They can be caused by invalid or corrupted data being passed to our program. In this case, our program should fail and print an error message to the user.

They can be caused by a programming mistake: Maybe another programmer uses a function that we have written, but passes a parameter of a wrong type to it. For example, maybe they pass in a string where we expect a number. Or maybe they pass a negative number when we expect a positive one. In this case, our program should fail and print an error message to the user.

Failing by raising an `Exception` is a good thing. It clearly indicates that something is wrong. It gives the user or our fellow programmers a chance to become aware of an error and to take action to fix it. Other approaches, like GIGO or overly sanitizing corrupted input instead allow errors to propagate unnoticed.

Maybe some of the readers of this book are graduate or undergraduate students who use `Python` to implement code for experiments. Imagine how annoying it is to run an experiment and to find out one week later that all the data produced is garbage. You then not only feel sad about the waste of time, but now need to waste even more time: Where was the error? Maybe it would take another week to painstakingly debug your code step-by-step to find out that some function was used incorrectly due to a typo. How much better would it have been if the experiment had crashed right at its start, printing an `Exception` and stack trace to the `stdout` stream showing exactly where things went wrong? It would have saved you two weeks and lots of grief.

Of course, there are also situations where it is possible to gracefully recover from an error. For example, maybe our program is trying to delete a file that already has been deleted. The operation would fail, but that does not do any harm. We should notify the user, but we do not need to crash our application. For these scenarios, the `except` blocks exist. They allow us to catch errors which we can reasonably expect and that are not show stoppers.

The `finally` block allows us to properly complete an operation regardless whether an error happened or not. If we send data over an internet connection, we want to close this connection properly after we are done. We also want to close it if something goes wrong. If we write data to a file, then we want to close the file once we are done. We still want to close it properly if an error occurs, because then we can at least preserve the data that was already successfully written.

Compared to the `try`-`finally` block, the `with` statement adds more ease to handling of resources that need to be explicitly closed at some point in time. Such resources can be implemented as context managers, whose special closing routine is automatically called at the end of the body of the `with` statement.

Error handling in Python therefore allows us to develop software that is both robust and that clearly indicates if something goes wrong. Of course, for software to be called *robust*, it has to be tested. Luckily, pytest offers us also unit test capabilities that check whether `Exceptionss` are raised where expected. This completes our discussion of the error-related control flow.

Chapter 10

Iteration, Comprehension, and Generators

In Python, iterating over the items in a sequence is a central concept. In Section 7.5, we learned that we can iterate over collections such as lists, tuples, dictionaries, and sets. We can also iterate over the characters in a string in the same way. These are all data structures whose complete content exists in memory at any given time. In Python, we can also iterate over sequences where the items are constructed only at the time when they are actually needed. A good example of this is the `range` datatype. We can iterate over all the 1 000 000 000 000 `int` elements of `range(100_000_000_000_000)` in a loop. These many integers do not all exist in memory at the same time. Instead, they are allocated and provided one-by-one as needed. From the perspective of a programmer, we can iterate over `ranges` and `lists` in exactly the same way. Matter of fact, many objects in Python support iteration.

Vice versa, we can also create instances of collection datatypes from sequences of items. For example, the datatypes `list`, `tuple`, `set`, and `dict` can also be used like functions that take a sequence of items as parameter and create an instance of the corresponding datatype. In Section 5.1, we learned that `[1, 2, 2, 3]` is a list literal with the specified contents. Passing this list to the `set` function/datatype, i.e., writing `set([1, 2, 2, 3])` will create the set `{1, 2, 3}`. We also learned that collection data structures often have methods that allow us to modify in place by passing in other collections as arguments. Invoking `l.extend({1, 2, 3})` will append the elements `1`, `2`, and `3` to a list `l`, for example.

You will very often encounter situations where you transform, process, or create sequences of data elements. As sketched in Figure 10.1, there are many different manifestations of the concepts of *iterating* over objects that are *iterable* in Python.

The most primitive concept is the `Iterator` [88, 289, 647]. This is an object that represents one visitation of a sequence of items. If you have an `Iterator` object `u`, then you can get the next item from the sequence it represents by calling `next(u)`. If there is no next element, this will raise a `StopIteration` exception. Such iterators are single-use, one-pass objects. A `for` loop, for example, will consume elements from an `Iterator` until the `StopIteration` exception is raised. `Generator` expressions and functions are special `Iterators` allowing us more control and a simpler syntax for defining element sequences, respectively.

Many data structures like collections allow us to visit their elements as often as we wish. They are instances of the `Iterable` interface. We can invoke `iter(coll)` on a collection `coll` implementing this

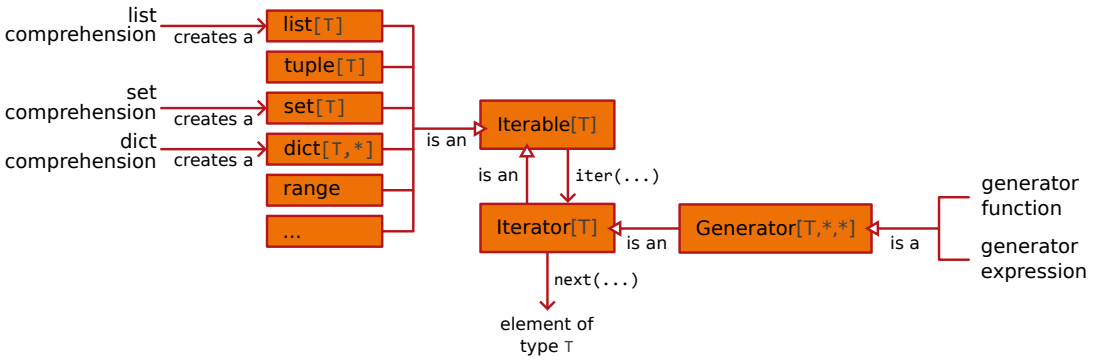


Figure 10.1: The concepts of comprehension, `Iterables`, `Iterators`, and `Generators` in Python.

`Iterable` [288] interface and will get an `Iterator`. Whenever we iterate over a list, a new `Iterator` is created this way.

As a side note, the operator `iter` can also be applied to an `Iterator`, not just to an `Iterable` [88]. If applied to an `Iterator`, it returns the iterator itself [289]. Hence, all APIs that require an `Iterable` as parameter and *use it only once* can also accept an `Iterator`.

Finally, we can also create collections like lists, sets, and dictionaries via so-called *comprehension*. A comprehension basically means writing a `for` loop *into* the corresponding collection literal. It is a more compact and more efficient syntax to construct collections. In this chapter, we will investigate all of these concepts beyond what we did not yet already discuss in Section 7.5 and Chapter 5.

10.1 `Iterables` and `Iterators`

Any object that allows us to access its elements one-by-one, i.e., *iteratively* is an instance of `typing.Iterable`. The actual iteration over the contents is then done by a `typing.Iterator` [88, 289, 647]. This distinction is necessary because we want to allow some objects to be iterated over multiple times.

Let's say you have the list `x = ["a", "b", "c"]`, as in Listing 10.1. We can use this list `x` in `for xi in x`-kind of loops arbitrarily often. We use `x` in two different such `for` loops. `x` is an instance of `list` and every list is also an instance of `Iterable` [288]. We show this by first importing the type `Iterable` from package `typing`. As we already learned, the operator `isinstance(obj, tpe)` returns `True` if object `obj` is an instance of type `tpe`. `isinstance(x, Iterable)` is therefore `True`, because the list `x` can be iterated over.

Every time we do loop over `x`, an `Iterator` instance is created internally by (doing something like) invoking `u = iter(x)`. To verify whether `u` really is an instance of the ominous type `Iterator`, we first import this type from the package `typing`. Then we invoke `isinstance(u, Iterator)`, which returns `True`. More precisely, the actual type of `u` is `list_iterator`, which is a special implementation of `Iterator`. You see, if we want to represent one step-by-step pass over the sequence `x`, then all we have to store in an object `u` is a reference to the list `x` we are iterating over as well as the current position, i.e., the index of the current element, in the iteration sequence. `list_iterator`, does exactly that internally.

Every time a loop needs to advance to the next element `xi` in the sequence represented by `u`, it does (something like) `xi = next(u)`. This will then yield the element at the current iteration index and advance the index by one. The `for` loop basically does this internally.

However, we can also do it "by hand." In Listing 10.1, we perform `u = iter(x)` and `v = iter(x)`. This creates two independent `Iterators`, i.e., two independent instances of `list_iterator`,, which we can use to step through the list independently. Each of them remembers a reference to list `x` as well as its own iteration index. Invoking `next(u)` will yield the first element of the list `x`, namely `"a"`. Calling `next(u)` again gives us the second element, that is `"b"`. If we now call `next(v)`, i.e., apply `next` to the second, independent `Iterator`, we again obtain the first element (`"a"`).

This again shows us why there is a distinction between the two APIs `Iterable` and `Iterator`. The former is interface that objects need to support it they holds or can generate a data sequence that can iteratively be visited. The latter is provided by one independent iteration over such sequence.

The third invocation of `next(u)` gives us `"c"`, the third and last element of `x`. If we now call `next(u)` a fourth time, something interesting happens: A `StopIteration` is raised. Different from the exceptions that we already learned about, this is not an error at all. This instead is how the end of an iteration sequence is signaled. A `for` loop will, for instance, stop when it encounters this exception. If `next(u)` is the way to get the next element from `Iterator` `u`, then there must also be some way to signal that the end of the sequence is reached. Returning `None` would not work, because a sequence may actually contain that value. Therefore, the designers of `Python` simply chose to use the exception mechanism for this.

This approach to iterate over collections `col` by first creating an iterator `it` using the `iter` function as `it = iter(col)` and then applying `next` to that iterator like `next(it)` works for `lists` and `tuples` alike. It also works for `sets`, but be aware that the order in which the elements of a `set` are presented is not defined. Back in Best Practice 28 we already clarified that `sets` are unordered data structures. We explore this with program `set_iteration.py` in Listing 10.2, which we execute twice, giving us the different outputs Listings 10.3 and 10.4. Interestingly, we can also iterate over `dicts`. This iteration

Listing 10.1: Manually iterating over a `list`.

```
Python 3.14.10 (main, Oct 27 2026, 09:35:01) [GCC 15.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> x = ["a", "b", "c"]        # Create a list `x` of three strings.

>>> for xi in x:     # We can loop over this list.
...     print(xi)
a
b
c

>>> for xi in x:     # We can loop over this list as often as we want.
...     print(f"Hello letter {xi!r}!")
Hello letter 'a'!
Hello letter 'b'!
Hello letter 'c'!

>>> from typing import Iterable, Iterator  # Import the two iteration types
>>> isinstance(x, Iterable)  # Check if `x` can be iterated over.
True
>>> u = iter(x)              # Create an Iterator `u` over Iterable `x`.
>>> isinstance(u, Iterator)  # Check if `u` really is an `Iterator`.
True
>>> type(u)                  # The type of `u` is a subclass of `Iterator`.
<class 'list_iterator'>
>>> v = iter(x)              # Create an Iterator `v` over Iterable `x`.
>>> next(u)                  # Get first element in sequence `u`: `"a"`.
'a'
>>> next(u)                  # Get second element in sequence `u`: `"b"`.
'b'
>>> next(v)                  # Get first element in sequence `v`: `"a"`.
'a'
>>> next(u)                  # Get third element in sequence `u`: `"c"`.
'c'
>>> next(u)                  # `u` is exhausted, raises `StopIteration`.
Traceback (most recent call last):
  File "<console>", line 1, in <module>
StopIteration
>>> next(v)                  # Get second element in sequence `v`: `"b"`
'b'
>>> w = iter(x)              # Create an Iterator `w` over Iterable `x`.
>>> next(w)                  # Get first element in sequence `w`: `"a"`.
'a'
>>> next(v)                  # Get third element in sequence `v`: `"c"`.
'c'
>>> next(v)                  # `v` is exhausted, raises `StopIteration`.
Traceback (most recent call last):
  File "<console>", line 1, in <module>
StopIteration
```

returns *only* the dictionary keys however. If we need the values or the key-value pairs of a dictionary `d`, then we have to iterate over `d.values()` or `d.items()`, respectively.

Listing 10.5 shows us that even `ranges` have the exactly same behavior as `lists` with respect to iteration. And they should, of course, like every other object that implements the `Iterable` functionality. Because of this, the `for y in x`-type of loops can be applied to any `Iterable` or `Iterator` instance `x`.A `range` is basically a collection. Different from the other collection types we know, its elements are not all explicitly created and stored. Instead, they are created on the fly by the `Iterator` objects that return them. In our example, we construct a range `x` of the three numbers 0 to 2. We then explore and iterate over it in exactly the same way we applied in Listing 10.5. The only differences are that we now output numbers instead of strings and that the type of the iterator is `range_iterator`.

Listing 10.2: Iterating over a set and the result: Every time we run the program, the output is likely to be different.Compare Listing 10.3 and Listing 10.4 and you will see that they are (probably) different. (stored in file `set_iteration.py`; output in Listing 10.3)

```
"""Iterating over a set: The resulting order is not clear!"""

my_set: set[str] = {      # Create a set of the 26 Latin letters.
    "a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m",
    "n", "o", "p", "q", "r", "s", "t", "u", "v", "w", "x", "y", "z"}
my_list: list[str] = []   # A list to receive the set elements.

for element in my_set:    # Iterate over the set: The order is undefined.
    my_list.append(element)

# Merging all the letters in the order in which they were visited into a
# single string and printing them. Each time we run this program, the
# result is likely to be different.
print("".join(my_list))
```

↓ `python3 set_iteration.py` ↓

Listing 10.3: The stdout of the program `set_iteration.py` given in Listing 10.2.

```
iroxkwgszadlbmujcqpvtfheyn
```

Listing 10.4: Execuring program `set_iteration.py` a second time. This time, the output should be different from Listing 10.3.

```
ladezgmjqpnfkrbxcvhsuiotyw
```

With this, we now know how `for` loops in `Python` actually work. They create an iterator over a sequence and then consume the elements of this iterator, each time executing the loop body, until hitting a `StopIteration` exception. All collection classes in `Python` that offer a sequential view on their data therefore support the `Iterable`/`Iterator`-API [647]. Due to this API structure, it is not even necessary to hold all the elements of a collection in memory at any point in time, as long as we can compute them as needed. An example of this are `ranges`, which provide us with `int`-sequences with arbitrarily many numbers that are created (and discarded) one-by-one during the iteration.

Listing 10.5: Manually iterating over a `range`, in exactly the same way that we used in Listing 10.1.

```
Python 3.14.10 (main, Oct 27 2026, 09:35:01) [GCC 15.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> x = range(3)         # Create a range `x` with the numbers from 0 to 2.

>>> for xi in x:     # We can loop over this range.
...     print(xi)
0
1
2

>>> for xi in x:     # We can loop over this range as often as we want.
...     print(f"Hello number {xi}!")
Hello number 0!
Hello number 1!
Hello number 2!

>>> from typing import Iterable, Iterator  # Import the two iteration types
>>> isinstance(x, Iterable)  # Check if `x` can be iterated over.
True
>>> u = iter(x)              # Create an Iterator `u` over Iterable `x`.
>>> isinstance(u, Iterator)  # Check if `u` really is an `Iterator`.
True
>>> type(u)                  # The type of `u` is a subclass of `Iterator`.
<class 'range_iterator'>
>>> v = iter(x)              # Create an Iterator `v` over Iterable `x`.
>>> next(u)                  # Get first element in sequence `u`: `"0"`.
0
>>> next(u)                  # Get second element in sequence `u`: `"1"`.
1
>>> next(v)                  # Get first element in sequence `v`: `"0"`.
0
>>> next(u)                  # Get third element in sequence `u`: `"2"`.
2
>>> next(u)                  # `u` is exhausted, raises `StopIteration`.
Traceback (most recent call last):
  File "<console>", line 1, in <module>
StopIteration
>>> next(v)                  # Get second element in sequence `v`: `"1"`
1
>>> w = iter(x)              # Create an Iterator `w` over Iterable `x`.
>>> next(w)                  # Get first element in sequence `w`: `"0"`.
0
>>> next(v)                  # Get third element in sequence `v`: `"2"`.
2
>>> next(v)                  # `v` is exhausted, raises `StopIteration`.
Traceback (most recent call last):
  File "<console>", line 1, in <module>
StopIteration
```

10.2 List Comprehension

We can create a list by writing it down as a literal. For example, `[1, 2, 3, 4, 5]` creates a list containing the first five natural numbers. This is very handy, but can also become cumbersome if we either have many elements or want to transform them. Needing to write a literal in the same way that creates a list with the first one hundred natural numbers may be somewhat annoying. If we want to create a list with the logarithms of first five natural numbers. Writing something like `[log(1), log(2), log(3), log(4), log(5)]` looks clunky as well. Luckily, Python offers the much more convenient syntax of list comprehension [610]:

```
"""List Comprehension in Python."""

# Create a list from all the items in a sequence.
# 'expression' is usually an expression whose result depends on 'item'.
[expression for item in sequence]

# Create a list from those items in a sequence for which 'condition'
# evaluates to True.
# 'expression' and 'condition' are usually expressions whose results
# depend on 'item'.
[expression for item in sequence if condition]
```

This syntax creates a new list whose contents are the results of applying a given `expression` to the items `item` of a `sequence`. You can imagine it as a `for` loop where each iteration produces a value which is then stored in a list. For example, `[i for i in range(10)]` creates a list with the integer numbers `0` to `9`. The list comprehension `[i ** 2 for i in range(10)]` instead creates a list with the squares of these numbers (where "squaring" is the before-mentioned expression).

Optionally, we can select the elements that we want to have in the list by adding an `if` clause. `[i for i in range(10)if i != 3]`, for instance, excludes the number `3` from our list. Interestingly, the sequence over which the list creation iterates can itself also be such a comprehension expression. It is totally fine to write `[i * j for i in range(2)for j in range(2)]`, which yields `[0, 0, 0, 1]` because both `i` and `j` will take on the values `0` and `1` independently of each other.

Listing 10.6: Some simple examples of list comprehension. (stored in file `simple_list_comprehension.py`; output in Listing 10.7)

```
"""Simple examples for list comprehension."""

squares_1: list[int] = []  # We can start with an empty list.
for i in range(11):  # Then we use a for-loop over the numbers 0 to 10.
    squares_1.append(i ** 2)  # And append the squares to the list.
print(f" result of construction: {squares_1}")  # Print the result.

# Or we use list comprehension as follows:
squares_2: list[int] = [j ** 2 for j in range(11)]
print(f"result of comprehension: {squares_1}")  # Print the result.

# A very simple example of how to use `if` in list comprehension.
even_numbers: list[int] = [k for k in range(10) if k % 2 == 0]
print(f"even numbers: {even_numbers}")
# Of course, that we just an example, I know, I know, we can also...
print(f"even numbers: {list(range(0, 10, 2))}")  #  ok, ok, yes...

combinations: list[str] = [f"{m}{n}" for m in "abc" for n in "xy"]
print(f"letter combinations: {combinations}")

nested: list[tuple[str, str]] = [(o, p) for o in "abc" for p in "xy"]
print(f"letter combinations as tuples: {nested}")
```

↓ `python3 simple_list_comprehension.py` ↓

Listing 10.7: The stdout of the program `simple_list_comprehension.py` given in Listing 10.6.

```
 result of construction: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
result of comprehension: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
even numbers: [0, 2, 4, 6, 8]
even numbers: [0, 2, 4, 6, 8]
letter combinations: ['ax', 'ay', 'bx', 'by', 'cx', 'cy']
letter combinations as tuples: [('a', 'x'), ('a', 'y'), ('b', 'x'), ('b', '
    ↪ y'), ('c', 'x'), ('c', 'y')]
```

Listing 10.8: The output of `Ruff` for the examples of list comprehension in Listing 10.6: The `for` loop constructing the list `squares_1` can indeed by replaced by a list comprehension.

```
$ ruff check --target-version py312 --select=A,AIR,ANN,ASYNC,B,BLE,C,C4,COM
    ↪ ,D,DJ,DTZ,E,ERA,EXE,F,FA,FIX,FLY,FURB,G,I,ICN,INP,ISC,INT,LOG,N,NPY,
    ↪ PERF,PIE,PLC,PLE,PLR,PLW,PT,PYI,Q,RET,RSE,RUF,S,SIM,T,T10,TD,TID,TRY,
    ↪ UP,W,YTT --ignore=A005,ANN001,ANN002,ANN003,ANN204,ANN401,B008,B009,
    ↪ B010,C901,D203,D208,D212,D401,D407,D413,INP001,N801,PLC2801,PLR0904,
    ↪ PLR0911,PLR0912,PLR0913,PLR0914,PLR0915,PLR0916,PLR0917,PLR1702,
    ↪ PLR2004,PLR6301,PT011,PT012,PT013,PYI041,RUF100,S,T201,TRY003,UP035,W
    ↪  --line-length 79 simple_list_comprehension.py
PERF401 Use a list comprehension to create a transformed list
 --> simple_list_comprehension.py:5:5
3 | squares_1: list[int] = []  # We can start with an empty list.
4 | for i in range(11):  # Then we use a for-loop over the numbers 0 to 10.
5 |     squares_1.append(i ** 2)  # And append the squares to the list.
  |     ^^^^^^^^^^^^^^^^^^^^^^^^
6 | print(f" result of construction: {squares_1}")  # Print the result.
help: Replace for loop with list comprehension
Found 1 error.
# ruff 0.17.3 failed with exit code 1.
```

In Listing 10.6 we provide some examples of list comprehension. First, we want to construct a list with the squares of the integer numbers from 0 to 10. Before learning about list comprehension, we would do this with a good old plain `for` loop. We would start by initially creating an empty list `squares_1`. In a `for` loop letting a variable `i` iterate over the `range(0, 11)`. In the body of the loop, we would then append `i ** 2` to the list `squares_1` by invoking `squares_1.append(i ** 2)`. This will occupy at least three lines of code. But it works. However, instead, we could write the comprehension expression `[j ** 2 for j in range(11)]`, which achieves exactly the same thing only using a single line of code.

We can also select which elements of a sequence we want to insert into our list by using an `if` statement during the list comprehension. In Listing 10.6, we demonstrate this by creating a list of even numbers from the range 0 to 9. We let a variable `k` iterate over the `range(10)`. This lets `k` take on the values `0`, `1`, `2`, ..., `8`, and finally `9`. Out of these values, we select only those for which `k % 2 == 0` via the `if` statement. In other words, we compute the result of the modulo division of `k` and `2`, i.e., the remainder of that division. If it is zero, then `k` is divisible by `2` and thus even. Yes, yes, I know ... of course, we could easily achieve the same result without the `if` here by simply changing the range in the comprehension to `range(0, 10, 2)` ... or by just doing `list(range(0, 10, 2))`. It is just an example. Anyway, we obtain the list `[0, 2, 4, 6, 8]`.

Finally, we play around with "nested" comprehension. Let's say you have two `Iterables` and want to produce all possible combinations of their output. Side note: Interestingly, `str` is also an `Iterable`. You can iterate over the characters of a string.

Assume that the first sequence be `"abc"` and the second one be `"xy"`. How can we create a list of all possible pairs containing one letter from each of these strings? By simply writing two `for` statements! In Listing 10.6, we create the list `combinations` this way. We write `[f"{m}{n}"for m in "abc"for n in "xy"]`. This lets the variable `m` take on, as value, each of the characters in the string `"abc"`, one by one. For each of the values that `m` takes on, the variable `n` iterates over `"xy"` and thus first becomes `"x"` and then `"y"`. The f-string `f"{m}{n}"` is interpolated for each combination of `m` and `n`. The result is thus the list `["ax", "ay", "bx", "by", "cx", "cy"]`.

Of course, we can create lists of arbitrary datatypes using list comprehension. These include also other lists, tuples, sets, dictionaries – whatever we want. We can repeat the above example and, instead of storing the combinations of characters as strings, we could store them as `tuples`. The list comprehension `[(o, p)for o in "abc"for p in "xy"]` does this. It produces `[("a", "x"), ("a", "y"), ("b", "x"), ("b", "y"), ("c", "x"), ("c", "y")]`.

In Listing 10.8, we present the output that `Ruff`, our Useful Tool 5, produces when we apply it to `simple_list_comprehension.py` from Listing 10.6. Interestingly, `Ruff` considers the construction of a list via the `append` function in a loop as a *performance issue*, signified by the error prefix `PERF`. This is,

Listing 10.9: Create a `list` with the even numbers from 0 to 1 000 000 using the `append`. (stored in file `list_of_numbers_append.py`; output in Listing 10.10)

```
"""Measure the runtime of list construction via the append method."""

from timeit import repeat  # needed for measuring the runtime


def create_by_append() -> list[int]:
    """
    Create the list of even numbers within 0..1'000'000.

    :return: the list of even numbers within 0..1'000'000
    """
    numbers: list[int] = []
    for i in range(1_000_001):
        if i % 2 == 0:
            numbers.append(i)
    return numbers


# Perform 50 repetitions of 1 execution of create_by_append.
# Obtain the minimum runtime of any execution as the lower bound of how
# fast this code can run.
time_in_s: float = min(repeat(create_by_append, number=1, repeat=50))
print("==== iterative list construction via append ====")
print(f"runtime/call: {1000 * time_in_s:.3} ms.")  # Print the result.
```

↓ `python3 list_of_numbers_append.py` ↓

Listing 10.10: The stdout of the program `list_of_numbers_append.py` given in Listing 10.9.

```
==== iterative list construction via append ====
runtime/call: 31.3 ms.
```

of course, only the case if the list could as well be constructed without calling `append` in a loop. Well, we do know another way: We can create the list via list comprehension, which is not always possible. When it can be done, as is the case in our example, we noticed that list comprehension is more compact. Code which is more compact is often more readable and in this case, from a software engineering point of view, often preferable. But why would the original loop also be an issue of performance, i.e., execution speed?

OK, let's try to verify this. In order to investigate this issue, we could try to simply create two lists with the same contents. We would create one by using the `append` method in a loop and one by using list comprehension. Very much like what we did just now. Whichever is faster, i.e., needs less time, has the better performance.

Now measuring the runtime needed by something is always a dodgy subject. The runtime of a Python program obviously depends on the machine and CPU it is running on. It is also affected by the operating system, the available RAM, the disk speed, and of course by other processes running on the same machine at the same time [616]. Clearly, it also depends on which version of the Python interpreter we use and our results of the measurement could be different after each software update. So every runtime measurement is always fuzzy and imprecise [616]. Whatever we would measure would have to take with a grain of salt, but we will try it anyway.

Useful Tool 8

`timeit` is a tool for measuring execution time of small code snippets that ships directly with Python. This module avoids a number of common traps for measuring execution times, see [442, 571].

Listing 10.11: Create a `list` with the even numbers from 0 to 1 000 000 using list comprehension. (stored in file `list_of_numbers_comprehension.py`; output in Listing 10.12)

```
"""Measure the runtime of list construction via list comprehension."""

from timeit import repeat  # needed for measuring the runtime


def create_by_comprehension() -> list[int]:
    """
    Create the list of even numbers within 0..1'000'000.

    :return: the list of even numbers within 0..1'000'000
    """
    return [i for i in range(1_000_001) if i % 2 == 0]


# Perform 50 repetitions of 1 execution of create_by_comprehension.
# Obtain the minimum runtime of any execution as the lower bound of how
# fast this code can run.
time_in_s: float = min(repeat(
    create_by_comprehension, number=1, repeat=50))
print("==== list comprehension ====")
print(f"runtime/call: {1000 * time_in_s:.3} ms.")  # Print the result.
```

↓ `python3 list_of_numbers_comprehension.py` ↓

Listing 10.12: The stdout of the program `list_of_numbers_comprehension.py` given in Listing 10.11.

```
==== list comprehension ====
runtime/call: 29.4 ms.
```

`timeit` allows us to measure the runtime of a certain statement. We want to measure how long it takes to create a list containing all even numbers from 0..1 000 000.

In Listing 10.9 we therefore first implement a function `create_by_append` which constructs the list using the `append` method in a loop. In a loop over the `range(1_000_001)`, it appends all the even numbers to the list `numbers`. Finally, it returns the list.

To measure the runtime of this function, we first `import` the function `repeat` from the module `timeit`. We tell `repeat` to call our function `create_by_append` one time (`number=1`) and measure the consumed runtime. It will return the runtime in seconds and stored as a `float`. However, factors like those mentioned above, scheduling by the operating system, maybe garbage collection by the Python iterpreter, and so on, a single measurement would not be very reliable [442]. We thus instruct `repeat` function to take 50 such measurements (argument `repeat=50`). All of the measured runtimes are then return as a `list[float]`. The documentation of `timeit` [571] says:

> *Note:* it's tempting to calculate mean and standard deviation from the result vector and report these. However, this is not very useful. In a typical case, the lowest value gives a lower bound for how fast your machine can run the given code snippet; higher values in the result vector are typically not caused by variability in Python's speed, but by other processes interfering with your timing accuracy. So the `min()` of the result is probably the only number you should be interested in. After that, you should look at the entire vector and apply common sense rather than statistics.
>
> — *[571], 2001*

Best Practice 54

When measuring the runtime of code for one specific set of inputs, it makes sense to perform multiple measurements and to take the *minimum* of the observed values [571]. The reason is that there are many factors (CPU temperature, other processes, ...) that may *negatively* impact the runtime. However, there is no factor that can make your code faster than what your hardware permits. So the minimum is likely to give the most accurate impression of how fast your code can theoretically run on your machine. Notice, however, that there might be effects such as caching that could corrupt your measurements.

So we do just that. `repeat` gives us a list with measured runtimes. The `min` function accepts a sequence of elements and returns the smallest one [83]. Thus, we print the result of `min` applied to the list returned by `repeat`. We format the output to be in milliseconds and rounded to three digits, to be a bit more readable. The result can be seen in Listing 10.10.

Now the Portable Document Format (PDF) document of this book is built automatically using a GitHub Action [109]. This action executes all the `Python` example programs and weaves their output into the book. Every time I update the book, this process is repeated. This means that, when writing this text, I do not know what value you will see in Listing 10.10. On my local machine, I get `runtime/call: 30.1 ms.`

Anyway, in order to test whether list comprehension is really faster than iterative list construction via `append`, we now write the second program Listing 10.11. This program is very similar to Listing 10.9. It defines a function `create_by_comprehension` which creates the very same list as `create_by_append` does in Listing 10.9. However, it uses list comprehension. We measure the runtime of this function in exactly the same way as before. In Listing 10.12, you can see the result. Of course, this result may be different every time the book is compiled using the GitHub Action mentioned above. On my local machine, I get `runtime/call: 28.7 ms.`

This confirms that list comprehension is indeed a bit faster than iterative list construction on `Python` 3.12. The difference may have been bigger on older `Python` versions, but it is there. Five percent of runtime saved are nice. And even if list comprehension was not faster than iteratively constructing a list, it would still be better code, because it is shorter and more readable.

With list comprehension, we learned an elegant, powerful, and concise method to create lists. It generalizes the idea of list literals and combines it with `for` loops that even can be nested. Not only is this method of list construction more compact than using loops and appending items to a list, it is also faster. Comprehension can also be applied to some other collection types.

10.3 Interlude: doctests

We already learned that unit tests are part of all reasonable software development processes and of all reasonable CI pipelines. If we think about it, we realize that unit tests are also part of the documentation of a software. The docstrings of a function tells us what the function basically does, what parameters it expects, and gives information about the exceptions it may raise.

This is complemented by the unit tests, which present us very similar information, but in a more comprehensive way. From unit tests, we can actually see what output we expect for some selected inputs of a function. The docstrings of a `sqrt` function may tell us that it computes the square root of a number. The unit tests show us that it returns `2.0` if the input is `4` and `3.0` for `9`. The docstrings may tell us that it raises an `ArithmeticError` if the input is a negative number. The unit tests show us that it raises a `ArithmeticError` with message *"Invalid input -1."* if we pass in a `-1`.

It is easy to see that both docstrings and unit tests complement each other. Would it not be nice if we could place some of the unit test information directly into the docstring? For instance, that `sqrt(16)` yields `4.0` as result may be something that could neatly fit on a single line. It would be a very nice example for everybody reading the documentation of our function. Of course, we would not write *all* the unit tests into the docstrings, because we usually have many test cases which would become hard to read. But a certain amount of selected tests would probably be quite helpful for the reader.

Well, nothing stops us from just writing this down. However, in an ideal case, `pytest` would also pick up these test cases from the docstrings and then actually execute and check them. Indeed, this

Listing 10.13: A function that flattens `lists` and other `Iterables` using list comprehension. (src)

```
"""A utility for flatten sequences of sequences."""

from typing import Iterable


def flatten(iterables: Iterable[Iterable]) -> list:
    """
    Flatten an :class:`Iterable` of `Iterable`s to a flat list.

    :param iterables: the `Iterable` containing other `Iterable`s.
    :return: a list with all the contents of the nested `Iterable`s.

    >>> flatten([[1, 2, 3], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[1, 2, 3], [], [4, 5, 6]])
    [1, 2, 3, 4, 5, 6]

    >>> flatten([[[1], [2], [3]], [], [[4], [5], [6]]])
    [[1], [2], [3], [4], [5], [6]]

    >>> flatten(([1, 2, 3], (4, 5, 6), {"a": 7, "b": 8}))
    [1, 2, 3, 4, 5, 6, 'a', 'b']
    """
    return [value for subiterable in iterables for value in subiterable]
```

Listing 10.14: The output of pytest executing the doctests for the examples of list comprehension in Listing 10.13: The test succeeded. We used the test execution script given in Listing 16.5.

```
$ pytest --timeout=10 --no-header --tb=short --doctest-modules
    ↪ list_flatten_iterables.py
========================== test session starts ==========================
collected 1 item
list_flatten_iterables.py .                                         [100%]
=========================== 1 passed in 0.01s ===========================
# pytest 9.2.0 with pytest-timeout 2.4.0 succeeded with exit code 0.
```

perfect synthesis of documentation and testing is possible, with so-called doctests [160].

We explore this idea by using one last example of list comprehension. Consider the following scenario: Let's say that you have several lists. You want to create a single new list that contains all the elements of each of these existing lists. In Listing 10.13, we implement a function `flatten` that achieves an even more general variant of this task: You can pass in an `Iterable` of other `Iterables`. Since `listss` are `Iterables`, this allows you to pass in a `list` of `listss`. But you could also pass a `tuple` of `sets` if you want. The return value of `flatten` is a `list` containing the elements of all of the "inner" `Iterables`.

`flatten` creates this list from its parameter `iterables` by simply returning the list comprehension expression `[value for subiterable in iterables for value in subiterable]`. The variable `subiterable` iterates over `iterables`, i.e., becomes one of the, e.g., sub-list, at a time. Then, `value` iterates over the elements of `subiterable`. Thus, it iteratively takes on each of the values in each of the sub-list. Since we return a `list` with all of these values, we effectively flatten the list-of-lists. It may be a bit confusing that the inner `for` loop here is actually the outer `for` loop and vice-versa, but we experienced this already when we computed `[f"{m}{n}"for m in "abc"for n in "xy"]`. in an earlier example.

Normally, we would also provide some code that actually executes `flatten` and presents its output. This time, we do something else: We present doctests for `flatten`.

Useful Tool 9

A doctest is a unit test written directly into the docstring of a function, class, or module. We therefore insert a small snippet of Python code followed by its expected output. The first line of such codes is prefixed py `>>>`. If a statement needs multiple lines, any following line is prefixed by `...`. After the snippet, the expected output is written. The doctests can be performed by modules like `doctest` [160] or tools such as `pytest` [328] (Useful Tool 7). They collect the code, run it, and compare its output to the expected output in the docstring. If they do not match, the tests fail. We use `pytest` in this book, with the default configuration given in Listing 16.5.

Using doctests has another very unique advantage: It allows us to include examples of how our code should be used directly into the docstrings. And these examples can directly serve as unit tests! Let's read the docstrings of our `flatten` function in Listing 10.13.

The first doctest tells us that if we invoke `flatten([[1, 2, 3], [4, 5, 6]])`, we can expect the output `[1, 2, 3, 4, 5, 6]`. In other words, our function will flatten the list of two lists into a single list. The function constructed one flat list of six elements from the list containing two lists with three elements each. Then, we see that `flatten([[1, 2, 3], [], [4, 5, 6]])` should produce `[1, 2, 3, 4, 5, 6]` as well. The single empty list in the list-of-lists disappears, because it has no elements.

`flatten` can only reduce two list levels to one. We pass a list-of-lists-of-lists to `flatten` as in the third test (`flatten([[[1], [2], [3]], [], [[4], [5], [6]]])`). Our function will only remove one list level. It results in a list-of-lists (`[[1], [2], [3], [4], [5], [6]]`).

Since `flatten` works with `Iterables`, it also accepts mixed input. The final doctest symbolizes this as follows: For `flatten(([1, 2, 3], (4, 5, 6), {"a": 7, "b": 8}))`, we get `[1, 2, 3, 4, 5, 6, 'a', 'b']` as the result. Notice that only the dictionary keys appear. As we discussed, applying `iter` to a `dict` yields an `Iterator` over only the keys. Since `for` loops do this implicitly, the keys are what appears in our flat list.

As you can see, the documentation in form of examples also explains what output we should expect. We now execute `pytest` with the additional option `--doctest-modules`. The full command line includes, again, the timeout of ten seconds via `--timeout=10`. We also use two arguments (`--no-header` and `--tb=short`) to shorten the output so that it is looks good as a listing in this book – which is something you would probably not care about when out there in the wild. Either way, `pytest --doctest-modules fileOrDirToTest` would also do the trick on your computer, where `fileOrDirToTest` obviously is to be replaced with the file or directory of files that you want to test. The output in Listing 10.14 shows that our function indeed fulfills the requirements imposed by its doctests.

Best Practice 55

Whereever possible, the docstrings of functions, classes, and modules should contain doctests. This provides unit tests as well as examples as how the code should be used. Since doctests are usually brief, they are a quick and elegant way to complement more comprehensive unit tests in separate files (see Best Practice 44).

Here we execute the doctests using `pytest` from the command line. You can also run them directly in PyCharm. We do this later in Section 13.4.

10.4 Set Comprehension

We already learned about list comprehension. It is a very elegant and powerful tool to create, well, lists. Wouldn't it be strange if such a tool would *only* be available for lists? What about the other collection types? Well, it is also available for sets and dictionaries. We now look at the former and afterwards will consider the latter.

Set comprehension works very much the same as list comprehension. The corresponding syntax is as follows and only differs in the fact that curly braces are used instead of square brackets:

```
"""Set Comprehension in Python."""

# Create a set from all the items in a sequence.
# 'expression' is usually an expression whose result depends on 'item'.
{expression for item in sequence}

# Create a set from those items in a sequence for which 'condition'
# evaluates to True.
# 'expression' and 'condition' are usually expressions whose results
# depend on 'item'.
{expression for item in sequence if condition}
```

In program simple_set_comprehension.py given as Listing 10.15, we provide some simple examples of set comprehension. In Listing 10.16 you can find the output of the program. First, we create a set with the results of the `isqrt` function from the `math` module. This function computes the integer part of a square root, i.e., `isqrt(i)` $= \lfloor\sqrt{\texttt{i}}\rfloor$. We want to create the set with all results of this function for the values `i` from 0 to 99.

We test two methods for creating such a set. We first do this by starting with an empty set `roots_1`. Then we iteratively append the `isqrt`-values by calling `roots_1.add` in a `for` loop. The constructed set contains, obviously, the values 0 to 9. Each value is contained a single time, because this is how sets work.

We now create the same set using set comprehension. The set `roots_2` created by evaluating

Listing 10.15: Some simple examples of set comprehension. (stored in file simple_set_comprehension.py; output in Listing 10.16)

```
"""Simple examples for set comprehension."""

from math import isqrt  # computes the integer parts of square roots

roots_1: set[int] = set()  # We can start with an empty set.
for i in range(100):  # Then we use a for-loop over the numbers 0 to 99.
    roots_1.add(isqrt(i))  # Add the integer part of sqrt to the set.
print(f" result of construction: {roots_1}")  # Print the result.

# Or we use set comprehension as follows:
roots_2: set[int] = {isqrt(j) for j in range(100)}
print(f"result of comprehension: {roots_2}")  # Print the result.

# Compute the set of numbers in 2..99 which are not prime.
not_primes: set[int] = {k for k in range(2, 100)
                        for m in range(2, isqrt(k) + 1) if k % m == 0}
# The set of numbers in 2..99 which are not in not_primes are primes.
primes: set[int] = {n for n in range(2, 100) if n not in not_primes}
print(f"prime numbers 1: {primes}")

# We could also use this method that creates a set from a range and uses
# the set difference operator.
print(f"prime numbers 2: {set(range(2, 100)).difference(not_primes)}")
```

↓ `python3 simple_set_comprehension.py` ↓

Listing 10.16: The stdout of the program simple_set_comprehension.py given in Listing 10.15.

```
 result of construction: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
result of comprehension: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
prime numbers 1: {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,
    ↪ 53, 59, 61, 67, 71, 73, 79, 83, 89, 97}
prime numbers 2: {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,
    ↪ 53, 59, 61, 67, 71, 73, 79, 83, 89, 97}
```

`{isqrt(j)for j in range(100)}` is exactly the same as `roots_1`. Set comprehension works exactly like list comprehension, and our first examples are also quite similar.

Let us now try do something more interesting: We want to create the set `primes` of the prime numbers [135, 481, 626] in the range 2 to 99. We already created a beautiful program doing this efficiently in Listing 7.11. This time, we will use set comprehension.

We first compute the set `not_primes` of numbers that are *not* prime. For this purpose, we let a variable `k` iterate from `2` to `99`. For each value of `k`, a second variable `m` iterates from `2` to `isqrt(k)`. For every single one of the resulting `k`-`m` combinations, we will add the value of `k` to the set *if* the condition `k % m == 0` is met. In other words, every single time we find a number `m` that can divide `k` without remainder, we will insert `k` into the set. If we would be doing a list comprehension, this would yield a huge list where many values of `k` appear repeatedly. For example, 96 would appear five times, because it is divisible by 2, 3, 4, 6, and 8, all of which are smaller than $9 = \lfloor\sqrt{96}\rfloor$. However, we are doing set comprehension, so each value can occur at most once.

This extremely inefficient method gives us the set `not_primes`. If a number can be divided by another one which is larger than 1 and smaller than the number itself, then the number can obviously not be a prime number. Having the numbers which are not primes, we can now use a second set comprehension to get the numbers which are primes. `{n for n in range(2, 100)if n not in not_primes}` lets a variable `n` again iterate from 2 to 99. It includes each value of `n` in the set to be constructed if `n` is not in the set `not_primes`. Indeed, this yields the set of prime numbers correctly (but does in a very inefficient way).

As a small refresher of our knowledge, let us remember the set operations from back in Section 5.4 and that we can create sets by passing sequences to the `set` function. First, we could do `set(range(2, 100))`, which yields the set of all integer numbers from 2..99. If we then invoke `set(range(2, 100)).difference(not_primes)`, we get the set of all of those numbers except the numbers that already appear in `not_primes`. With this expression, we could thus achieve the same result, if, at least, we already have constructed the set `not_primes`.

10.5 Dictionary Comprehension

Dictionary comprehension works almost the same as set and list comprehension [609]. Different from them, it assigns values to keys and therefore has two expressions denoting each entry, separated by `:`. This is also the difference between the syntax of set and dictionary comprehension. Both of them use curly braces, but in dictionary comprehension, keys and values are separated by `:`, whereas in a set comprehension, only single values are given. Dictionary comprehension has the following syntax:

```
"""Dictionary Comprehension in Python."""

# Create a dictionary from all the item pairs in a sequence.
# 'expression1' and 'expression2' are usually expressions whose results
# depend on 'item'.
{expression1: expression2 for item in sequence}

# Create a dictionary from those items in a sequence for which
# 'condition' evaluates to True.
# 'expression1', 'expression2', and 'condition' are usually expressions
# whose results depend on 'item'.
{expression1: expression2 for item in sequence if condition}
```

In Listing 10.17 we provide some examples of dictionary comprehension. We again start by "manually" creating a dictionary and then show how dictionary comprehension is much more concise. Similar to Listing 10.6, where we discussed list comprehension, we want to construct a data structure with the squares of the numbers from 0 to 10. This time, we use a dictionary and assign the squares (as values) to the numbers (the keys).

We start with an empty dictionary `squares_1`. Then we use a `for` loop iterating a variable `i` over the `range(11)`. In the loop body, we assign `squares_1[i] = i ** 2`, i.e., associate each number with its square. The output in Listing 10.18 shows that this produces the expected result. We can shorten this loop into a single dictionary comprehension. `{i: i ** 2 for i in range(11)}` produces the exactly same result.

Listing 10.17: Some simple examples of set comprehension. (stored in file simple_dict_comprehension.py; output in Listing 10.18)

```
"""Simple examples for dictionary comprehension."""

squares_1: dict[int, int] = {}  # We can start with an empty dictionary.
for i in range(11):  # Then we use a for-loop over the numbers 0 to 9.
    squares_1[i] = i * i  # And place the square numbers in the dict.
print(f" result of construction: {squares_1}")  # Print the result.

# Or we use dictionary comprehension as follows:
squares_2: dict[int, int] = {i: i ** 2 for i in range(11)}
print(f"result of comprehension: {squares_2}")  # Print the result.

# Compute the largest divisors of the numbers in 0..20.
maxdiv: dict[int, int] = {k: m for k in range(21)
                          for m in range(1, k) if k % m == 0}
print(f"       largest divisors: {maxdiv}")
```

↓ python3 simple_dict_comprehension.py ↓

Listing 10.18: The stdout of the program simple_dict_comprehension.py given in Listing 10.17.

```
 result of construction: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7:
    ↪ 49, 8: 64, 9: 81, 10: 100}
result of comprehension: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7:
   ↪ 49, 8: 64, 9: 81, 10: 100}
       largest divisors: {2: 1, 3: 1, 4: 2, 5: 1, 6: 3, 7: 1, 8: 4, 9: 3,
           ↪ 10: 5, 11: 1, 12: 6, 13: 1, 14: 7, 15: 5, 16: 8, 17: 1, 18: 9,
           ↪  19: 1, 20: 10}
```

Let us now try something more fancy. We want to create a dictionary `maxdiv` that holds the largest divisor m for each number k from the range 2..20 with $m < k$. We can apply the same (very inefficient) principle that we used in Listing 10.15 when constructing the set of none-prime numbers. First, of course, we need to let the variable `k` iterate over `range(21)`, i.e., let `k` take on the values `0`, `1`, `2`, ..., `19`, `20`. Now we let a second variable `m` iterate over `range(1, k)`, meaning that `m` takes on the values `1`, `2`, ..., `k - 1`. We store the association `k: m` in the dictionary if the condition `k % m == 0` is met, i.e., if `m` divides `k` without remainder.

For most `k`, condition will be met by several `m`. However, a dictionary allows each key to appear at most once. In the dictionary comprehension, the last assignment prevails. This means that the last `m` value that meets the condition is stored for each `k`. Since `m` iterates over strictly increasing values, this will be the largest divisor. Nevertheless, this is the reason why this computation is wildly inefficient ... but it makes for a nice example.

Notice that, different from our prime-number-set example, where `m` started at `2`, it here `m` starts at `1`. For any prime number k', the largest divisor $m < k'$ is then correctly 1. For 1, no such divisor exists, so 1 does not appear as key in the dictionary `maxdiv`. Finally, `maxdiv` is printed and indeed contains the largest divisors for the numbers $k \leq 20$.

With set and dictionary comprehension, we now expanded the nice properties of list comprehension to these two data structures. Both have a very simple syntax, which is very similar to list comprehension. They are quite useful for writing neat and tidy code.

10.6 Generator Expressions

List comprehension gives us the ability to elegantly solidify a sequence of data into an instance of `list`. A list data structure is basically one compact chunk of memory holding all the elements of the list. A list can be extended by adding elements to it, but it will always be managed as a continuous area of memory. It cannot exist in memory partially, but always as a whole. And typically, this is what we want: We want a data structure that can store the elements and that lets us access them in an efficient way. Tuples, sets, and dictionaries are other manifestations of this concept. All of them hold all their

elements in memory.

However, this is not the right solution for all tasks. Let's say you have a sequence of elements and you just want to add them all up. The `sum` function does exactly this [83]. It accepts an `Iterable` as input [83]. It repeatedly invokes `next` to get its elements and adds them in a running sum. `sum([i ** 2 for i in range(100)])` adds up the squares of the values `i` for `i` $\in$ 0..99. The list comprehension therefore first creates a `list` holding all these square values. This `list` is passed to `sum`, which then iterates over it and computes, well, the sum of the values.

This looks fine, but it has one drawback, namely the creation of the complete list in memory. Actually, all that we (or the `sum` function) need to do is to access the elements one-by-one. We need to do this only exactly once. There is no need to keep all the elements in memory. Matter of fact, all we really want is to repetitively invoke `next` on an `Iterator`, as we discussed in Section 10.1. This is how `sum` is implemented. It does not require the complete `list` to exist in memory. All it requires is an `Iterable` as argument to which it will apply `iter`, yielding an `Iterator`. Then it applies `next` to the `Iterable` to get the elements, which it then adds up. At no point does it require all elements to co-exist at the same time in memory.

To allow us to create "lazy" sequences that create and return their elements as needed (instead of keeping them all in memory all the time), *generator expressions* exist [260]. The syntax for generator expressions is basically the same as for list comprehension, except that we use parentheses instead of square brackets. There is one special case, though. If we pass a generator expression as function parameter to a function `my_func`, then we can omit the parentheses.

```
"""Generator Expressions in Python."""

# Create a generator from all the items in a sequence.
# 'expression' is usually an expression whose result depends on 'item'.
(expression for item in sequence)

# Create a generator from those items in a sequence for which
# 'condition' evaluates to True.
# 'expression' and 'condition' are usually expressions whose results
# depend on 'item'.
(expression for item in sequence if condition)

# If generator expressions are single function parameters, then the
# parentheses are unnecessary.
my_func(expression for item in sequence if condition)
```

Notice that, based on what we learned about list, set, and dictionary comprehension, one might mistake this syntax as tuple comprehension. However, generators are something very different, as we will see. Tuple comprehension does not exist in Python.

First, let us investigate a bit how generator expressions work internally. In program generator_expressions_next_1.py given as Listing 10.19, we create the generator expression `(i ** 2 for i in range(1_000_000_000))`. If this was a list comprehension, it would try to fit one billion integer values into our memory. But since it is a generator expression, it exists in memory only as the piece of code in the parentheses. This code will create the next value `i` and compute `i ** 2` only when they are actually needed. The proper type hint for such generator expressions is `Generator[etype]`, where `etype` is the type of elements, which here is `int`.

`Generator` is imported from the `typing` module.

We store the generator expression into a variable `gen`. We then print the contents of `gen`. If `gen` was a `tuple`, this would print a huge string consisting of all billion square numbers. However, instead it just tells us the object type of `gen` and its location in memory. Since the generator expression is just a piece of code in memory, there really is not much that Python can print for us here. As the next line of output shows, the `type` of `gen` is `<class 'generator'>`, neither `list` nor `tuple`.

We then confirm that generator expressions are a special case of `Iterators` using the `isinstance` operator. And it is an iterator that can be used only *once*. We can see this if we apply the `iter` operator to it. This operator returns to us the same object, `gen`, i.e., `iter(gen)is gen` is `True`. We can create arbitrarily many different iterators for lists or sets, each of them being an independent,

Listing 10.19: An investigation of how generator expressions work by using the `next` function. (stored in file generator_expressions_next_1.py; output in Listing 10.20)

```
"""The iteration behavior of generator expressions and `next`."""

from typing import Generator, Iterator  # Import the types.

# Create a generator expression of 1 billion square numbers.
gen: Generator[int] = (i ** 2 for i in range(1_000_000_000))

# If `gen` was a tuple, `print(gen)` would print the tuple contents.
print(gen)  # However, the contents of gen are <generator object...>.
print(f"{type(gen) = }")  # Type of `gen` is `generator`, not `tuple`...

# Every `Generator` is also an `Iterator`, i.e., `Generator`s are
# special cases of `Iterator`s.
print(f"{isinstance(gen, Iterator) = }")

# Doing `iter(gen)` with `Generator` `gen` yields `gen` again.
# This means that can iterate over a `Generator` only once, because we
# cannot create independent `Iterators` of it (as we could for `list`s).
print(f"{iter(gen) is gen = }")

# Now let's manually iterate a bit over the generator expression `gen`.
print(f"{next(gen) = }")  # Returns first element:   0
print(f"{next(gen) = }")  # Returns second element:  1
print(f"{next(gen) = }")  # Returns third element:   4
print(f"{next(gen) = }")  # Returns fourth element:  9
print(f"{next(gen) = }")  # Returns fifth element:  16
# We stop using the generator here, so the remaining 999'999'995
# elements are never generated.
```

↓ `python3 generator_expressions_next_1.py` ↓

Listing 10.20: The stdout of the program generator_expressions_next_1.py given in Listing 10.19.

```
<generator object <genexpr> at 0x7f8e19e64c70>
type(gen) = <class 'generator'>
isinstance(gen, Iterator) = True
iter(gen) is gen = True
next(gen) = 0
next(gen) = 1
next(gen) = 4
next(gen) = 9
next(gen) = 16
```

different object that has its own state information. However, `iter(gen)` yields `gen` itself and thus cannot be independently advanced.

And iterating over `gen` is what we will do. We can obtain the first element of the sequence, `0`, by invoking `next(gen)`. The second invocation of `next(gen)` yields `1`. The third call to `next(gen)` gives us `4`. Then `next(gen)` returns `9`. Finally, as expected, `next(gen)` yields `16`. In our example program, we stop here. This means that the 999 999 995 elements are never created.

To better understand this lazy evaluation behavior, we write a second example program named generator_expressions_next_2.py here printed as Listing 10.21. We now want to construct a generator expression which explicitly tells us exactly when an element is created. For this, we first create the function `as_str`. `as_str` accepts an integer parameter `a` as input. We want to know when exactly this function is invoked, so we add a so-called side effect. When it is invoked, the function immediately writes `f"as_str({a})"` to the `stdout` stream. It also passes `flush=True` to `print`, which forces all buffered output to be written, i.e., we really *really* immediately get to see that our function

Listing 10.21: An investigation of the lazy evaluation in generator expressions by using the `next` function. (stored in file generator_expressions_next_2.py; output in Listing 10.22)

```
"""The lazy iteration behavior of generator expressions and `next`."""

from typing import Generator  # Import the generator type hint.

def as_str(a: int) -> str:
    """
    A function with side effect printing `a` and returning it as `str`.

    :param a: the input number
    :return: the output

    >>> as_str(5)
    as_str(5)
    '5'
    """
    print(f"as_str({a})", flush=True)  # Shows when function is called.
    return str(a)


# List comprehension immediately performs all `as_str` calls.
lst: list[str] = [as_str(j) for j in range(3)]  # Prints 3 lines!
print("list created")  # This text appears after the 3 lines.
print(f"{next(iter(lst)) = }\n")  # This just prints '0'.
# Generator invokes `as_str` only when required during iteration.
gen: Generator[str] = (as_str(j) for j in range(3))
print("generator created")  # Nothing new is printed until now.
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"0"`
print(f"{next(gen) = }")    # 2 prints: first in `as_str` and then `"1"`
print(f"{next(gen) = }", flush=True)  # Last 2 prints `as_str` + `"2"`
print(f"{next(gen) = }", flush=True)  # Raises StopIteration
```

↓ `python3 generator_expressions_next_2.py` ↓

Listing 10.22: The stdout, stderr, and exit code of the program generator_expressions_next_2.py given in Listing 10.21.

```
as_str(0)
as_str(1)
as_str(2)
list created
next(iter(lst)) = '0'

generator created
as_str(0)
next(gen) = '0'
as_str(1)
next(gen) = '1'
as_str(2)
next(gen) = '2'
Traceback (most recent call last):
  File "{...}/iteration/generator_expressions_next_2.py", line 30, in <
    ↪ module>
    print(f"{next(gen) = }", flush=True)  # Raises StopIteration
              ~~~~~~~~~
StopIteration
# 'python3 generator_expressions_next_2.py' failed with exit code 1.
```

was called. This means that whenever `as_str` is called, we will immediately know. Then, the function simply returns the string representation of `a`, i.e., `str(a)`.

Let us first use this function in a list comprehension `lst = [as_str(j)for j in range(3)]`. The list comprehension creates the entire list right away. This means in the moment this line of code is executed and `lst` is created, our function `as_str` will be invoked three times, with `a = 0`, `a = 1`, and `a = 2`. We can see that this happens when the list is created, because all the `as_str`-output happens before the `print("list created")` in the next line completes.

In the following line, we print the value of `next(iter(lst))`. Here, `iter(lst))` creates an `Iterator` over `lst`. The `next` operator then returns the first element of this iterator. As you can see, the corresponding text appears in `stdout`. The function `as_str` is not invoked again at this stage. Of course not, because it was already used when the list was created. This shows that the `list` has indeed been created in its entirety before we began processing it in the next lines.

What happens if we use `as_str` in a generator expression instead? We do this by setting `gen = (as_str(j)for j in range(3))`, with the proper type hint, of course. When this line of code is executed, nothing happens. This is because the generator expression is just a piece of code in memory at this stage. The next line, `print("generator created")` prints its output. Obviously, `as_str` has not yet been invoked. We did not yet pull any element out of the sequence, so the generator expression is just sitting peacefully in memory.

We now iteratively query `gen` with `next`. Indeed, the first time we do this, `as_str(0)` appears in `stdout` *before* the result of the interpolated f-string `f"next(gen): {next(gen)}"` is printed. This is because the element to be returned by `next` first must be created. The generator expression therefore invokes `as_str(0)`. At this point, `"as_str(0)"` is written to the `stdout` stream (and all output buffers are flushed). Then, the string `"0"` is returned to the calling code, which, in this case, is the (string) interpolation. The f-string is interpolated to `"next(gen): '0'}"` and printed. It thus appears in the output *after* `"as_str(0)"`.

Notice that `as_str` was invoked only once. The other elements in the sequence are not yet generated at this stage. When we query `next(gen)` again in the next line, `as_str(1)` appears, and so on. Clearly the elements of the generator expression are indeed created when needed. This is called *lazy* evaluation. No memory is wasted during this process, as only one element exists in memory at a time.

Eventually, after the third `next(gen)` call, the `range(3)` is exhausted. Calling `next(gen)` a fourth time leads to a `StopIteration` exception being raised. This is not an error, but signals the end of the iteration. We can iterate over a generator expression only a single time.

Therefore, generator expressions cannot be used if we need to iterate over a sequence multiple times. They also do not work when we want to access elements in a random order, for example, via indices. In such cases, the collection data structures are the better choice. However, in several use cases, generator expressions excel:

Best Practice 56

Generator expressions shall be preferred over list comprehension if the sequence of items only needs to be processed once. Generator expressions require less memory. If the iteration over the elements can stop early, which can happen, e.g., when using the `all` or `any` functions, they may also be faster.

Generator expressions come in especially handy when we want to reduce or aggregate a sequence of data to a single number. Listing 10.23 shows several examples of such computations.

First, we want to sum up the squares of all numbers from 0 to 999 999. This was the example we mentioned at the beginning of this section. We can compute this by passing the generator expression `(j ** 2 for j in range(1_000_000))` to the `sum` function. Notice that we do not need to write `sum((generator expression...))` but single parentheses are sufficient. The result of the summation is 333 332 833 333 500 000.

We now want to find the largest and smallest value that $\sin k$ takes on for any $k \in -99..99$. We use the generator expression `(sin(k)for k in range(-100, 100))`. We have to write it twice: once to pass it to `max` in order to get the maximum value, and once to pass it to `min` to get the minimum value of the sequence. Again it is not necessary to write double-parentheses.

Listing 10.23: The use of generator expressions in *reducing functions*, such as `sum`, `min`, `max`, `all`, and `any`. (stored in file `generator_expressions_in_reduction.py`; output in Listing 10.24)

```
"""Generator expressions in functions reducing them to single values."""

from math import sin

sum_of_squares: int = sum(j ** 2 for j in range(1_000_000))
print(f"sum of squares for j in 0..1'000'000: {sum_of_squares}")

largest: float = max(sin(k) for k in range(-100, 100))
print(f" largest sin(k) for k in -99..99:  {largest}")

smallest: float = min(sin(m) for m in range(-100, 100))
print(f"smallest sin(m) for m in -99..99: {smallest}")

words: list[str] = ["hello", "how", "are", "you"]
print(f"The words are: {words!r}")

e_in_all: bool = all("e" in word for word in words)
print(f"Does every word contain an 'e': {e_in_all}")

w_in_any: bool = any("w" in word for word in words)
print(f"Does any word contain a 'w': {w_in_any}")
```

↓ `python3 generator_expressions_in_reduction.py` ↓

Listing 10.24: The stdout of the program `generator_expressions_in_reduction.py` given in Listing 10.23.

```
sum of squares for j in 0..1'000'000: 333332833333500000
 largest sin(k) for k in -99..99:  0.9999902065507035
smallest sin(m) for m in -99..99: -0.9999902065507035
The words are: ['hello', 'how', 'are', 'you']
Does every word contain an 'e': False
Does any word contain a 'w': True
```

As next example, we first create a list `words` of, well, words `["hello", "how", "are", "you"]`. We want to know if *all* words in the list contain the letter "e". The generator expression `"e" in word for word in words` evaluates `"e" in word` for every `word` in the list `words`. It would produce the sequence `True`, `False`, `True`, and `False`.

The function `all` returns `True` if and only if all the elements in the sequence that it receives as argument are `True`. Otherwise it returns `False`. We pass our generator expression as argument to `all`. It will obviously return `False` here as well. The interesting thing is that `all` can stop calling `next` on our generator as soon as it hits the first `False`. In other words, our generator expression will never evaluate `"e" in "are"`, because `"e" in "how"` is already `False`. It is clear, at this point, that `all` must return `False` regardless of the rest of sequence and so it will do just that immediately.

If we had used list comprehension, all the words would have been checked and the corresponding `bool` values would have been stored in a `list`. Using the generator expression, we only generated the part of the data that was actually used, one item at a time, and stored it nowhere.

A complement to the `all` function is the `any` function. `any` returns `True` if at least one of the elements of its argument sequence also is `True`. It returns `False` if *all* of the elements in the argument sequence are also `False`.

We now want to know whether the letter "w" appears in any of the words. For this, we write the generator expression `("w" in word for word in words)`. If evaluates `"w" in word` for every `word` in the list `words`. We pass this generator expression to the function `any`.

We immediately see that `"w" in "how"` is `True`. Thus, the result of `any` will also be `True`. And again, the evaluation of the sequence can stop at the point that the first `True` value is reached. Lazy evaluation here again reduces both the runtime and memory footprint.

Listing 10.25: The behavior of generator expressions in `for` loops. (stored in file generator_expressions_loops.py; output in Listing 10.26)

```
"""Generator expressions in `for` loops."""

from math import sin
from typing import Generator

# The generator expression produces tuples of the form `(n, sin(n))`.
gen: Generator[tuple[int, float]] = (
    (n, sin(n)) for n in range(1_000_000_000))

# The for loop iterates over the generator expression and unpacks the
# tuples into the variables `o` and `p`.
for o, p in gen:  # `o` = original `n` and `p = sin(o) = sin(n)`
    if p < -0.99999:  # We look for an n with sin(n) < -0.99999.
        print(f"sin({o})={p}")  # We found it and print it.
        break  # And stop the iteration, not using the full range.
else:  # This happens only if `break` was never invoked.
    print("No value p < -0.99999 found.")
```

↓ `python3 generator_expressions_loops.py` ↓

Listing 10.26: The stdout of the program generator_expressions_loops.py given in Listing 10.25.

```
sin(11)=-0.9999902065507035
```

We said before that Python `for` loops actually work on `Iterators`. Every `Generator` is also an `Iterator`. Therefore, we can of course also iterate over generator expressions using a `for` loop. This is illustrated in generator_expressions_loops.py illustrated as Listing 10.25.

Here, we create a generator expression `gen` which creates the tuples of the form $(n, \sin n)$ for all $n \in 0..1\,000\,000\,000$. This is done by writing `((n, sin(n))for n in range(1_000_000_000))`. Assume that for some reason, we want to find the first $n \in \mathbb{N}_0$ for which $\sin n < -0.99999$. We therefore iterate over this generator expression `gen`. We directly unpack the tuples it delivers in the same way we did back in Section 7.4 when looping over collections.

The variable `o` gets the original value of `n`, which is stored in the first tuple elements, and the variable `p` will take on $\sin$ `o`. We place the conditional `if p < -0.99999` to check for the number we want to find. If the expression `p < -0.99999` evaluates to `True`, we print the discovered values and leave the loop via `break`.

If the `for` loop is never terminated with a `break`, then the condition for `else` after the loop body is fulfilled. In this case, we print a message that we never found the value we were looking for.

The output `sin(11)=-0.9999902065507035` tells us that we did, however, indeed find a natural number fulfilling our condition. We also know that only 12 loop steps were done (as n starts with 0).

Using a generator expression here was not just more memory efficient than using a list, it also was faster, because it did not need to compute all the values. Well, of course, we would not have needed any generator or data structure in the first place. We could have just iterated over the `range` directly, which would have been even more efficient ... but then we could not have used this as an example.

Of course, generator expressions can also be passed to the constructors of collection data structures or other functions that create such data structures. Assume that we are processing numbers stored in a comma-separated values (CSV) format. Often, the rows of text files with tabular or matrix data are in this format. Here, ever line of text corresponds to a row of data elements. The single data elements in a row are usually separated by commas (",").

In program generator_expressions_to_collection.py given as Listing 10.27, we first define a string `csv_text` with the value `"22,56,33,67,43,33,12"`. It thus holds one row of CSV data.

Invoking `csv_text.split(",")` will split the string into a list of single strings based on the delimiter `","`. This will thus yield the list `["22", "56", "33", "67", "43", "33", "12"]`.

We can use this list in a generator expression. For example, `(int(i)for i in csv_text.split(","))`

Listing 10.27: Using generator expressions when creating collection data structures. (stored in file generator_expressions_to_collection.py; output in Listing 10.28)

```
"""Using generator expressions to create collections."""

csv_text: str = "22,56,33,67,43,33,12"

# This could be more efficiently replaced by list comprehension.
as_list: list[int] = list(int(i) for i in csv_text.split(","))
print(f"a generator converted to a list: {as_list}.")

# This cannot be replaced, because there is no tuple comprehension.
as_tuple: tuple[int, ...] = tuple(int(i) for i in csv_text.split(","))
print(f"a generator converted to a tuple: {as_tuple}.")

# This could be more efficiently replaced by set comprehension.
as_set: set[int] = set(int(i) for i in csv_text.split(","))
print(f"a generator converted to a set: {as_set}.")

# This cannot be replaced with a single line of code.
as_sorted_list: list[int] = sorted(int(i) for i in csv_text.split(","))
print(f"a generator converted to a sorted list: {as_sorted_list}.")

# This could be more efficiently replaced by dictionary comprehension.
as_dict: dict[str, int] = dict((i, int(i)) for i in csv_text.split(","))
print(f"a generator of tuples converted to a dict: {as_dict}.")
```

↓ `python3 generator_expressions_to_collection.py` ↓

Listing 10.28: The stdout of the program generator_expressions_to_collection.py given in Listing 10.27.

```
a generator converted to a list: [22, 56, 33, 67, 43, 33, 12].
a generator converted to a tuple: (22, 56, 33, 67, 43, 33, 12).
a generator converted to a set: {33, 67, 43, 12, 22, 56}.
a generator converted to a sorted list: [12, 22, 33, 33, 43, 56, 67].
a generator of tuples converted to a dict: {'22': 22, '56': 56, '33': 33,
    ↪ '67': 67, '43': 43, '12': 12}.
```

is a generator expression that converts the strings in the list into integers. Passing this generator expression to the `list` constructor, i.e., calling `list(int(i)for i in csv_text.split(","))`, will create a list with all the elements generated by the expression.

This also works exactly the same when passing it to the `tuple` or `set` constructors which then create a `tuple` or a `set`, respectively. Notice, however, that the constructed set only contains each element at most once, i.e., duplicates are removed. We can also pass a generator expression to the `sorted` function mentioned back in Section 5.4. This function takes a sequence and returns a sorted list with the same contents.

Dictionaries can be created from sequences of `tuples`. The first element of each tuple is used as key and the second one as value. We pass a generator expression that returns the tuples `(i, int(i))` for each `i` in `csv_text.split(",")` to `dict`. The keys of the new dictionary are thus the original strings in our CSV data and the values are their integer representation. Notice that this dictionary also does not contain duplicate keys.

We now apply the trusty tool Ruff to generator_expressions_to_collection.py. It tells us that using generator expressions to create lists, sets, and dictionaries is unnecessary. We could use the corresponding comprehension instead directly. And it is right. There is no tuple comprehension, though, so using a generator expression in `tuple(...)` is OK. The same holds for passing the generator expression to the `sorted` function.

With this, we have reached the end of our discussion of generator expressions. We have learned that we can use them as an efficient source of data. Different from collection data structures, they

do not hold all of their elements in memory at once. They instead offer us sequences that are lazily evaluated, i.e., their elements are computed and returned when queried. This is much more efficient in cases where we only need to access the data elements one-by-one. It is doubly efficient when we maybe want to abort the iteration over the data earlier. In that case, only the data elements up to this point are generated, saving both memory and runtime.

Listing 10.29: The output of `Ruff` for collection creation-expressions in Listing 10.27: Passing generators to `list`, `set`, or `dict` constructors could be replaced by list-, set-, or dictionary comprehension.

```
$ ruff check --target-version py312 --select=A,AIR,ANN,ASYNC,B,BLE,C,C4,COM
    ↪ ,D,DJ,DTZ,E,ERA,EXE,F,FA,FIX,FLY,FURB,G,I,ICN,INP,ISC,INT,LOG,N,NPY,
    ↪ PERF,PIE,PLC,PLE,PLR,PLW,PT,PYI,Q,RET,RSE,RUF,S,SIM,T,T10,TD,TID,TRY,
    ↪ UP,W,YTT --ignore=A005,ANN001,ANN002,ANN003,ANN204,ANN401,B008,B009,
    ↪ B010,C901,D203,D208,D212,D401,D407,D413,INP001,N801,PLC2801,PLR0904,
    ↪ PLR0911,PLR0912,PLR0913,PLR0914,PLR0915,PLR0916,PLR0917,PLR1702,
    ↪ PLR2004,PLR6301,PT011,PT012,PT013,PYI041,RUF100,S,T201,TRY003,UP035,W
    ↪  --line-length 79 generator_expressions_to_collection.py
C400 Unnecessary generator (rewrite as a list comprehension)
 --> generator_expressions_to_collection.py:6:22
5 | # This could be more efficiently replaced by list comprehension.
6 | as_list: list[int] = list(int(i) for i in csv_text.split(","))
  |                      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
7 | print(f"a generator converted to a list: {as_list}.")
help: Rewrite as a list comprehension
5 | # This could be more efficiently replaced by list comprehension.
  - as_list: list[int] = list(int(i) for i in csv_text.split(","))
6 + as_list: list[int] = [int(i) for i in csv_text.split(",")]
7 | print(f"a generator converted to a list: {as_list}.")
note: This is an unsafe fix and may change runtime behavior
C401 Unnecessary generator (rewrite as a set comprehension)
  --> generator_expressions_to_collection.py:14:20
13 | # This could be more efficiently replaced by set comprehension.
14 | as_set: set[int] = set(int(i) for i in csv_text.split(","))
   |                    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
15 | print(f"a generator converted to a set: {as_set}.")
help: Rewrite as a set comprehension
13 | # This could be more efficiently replaced by set comprehension.
   - as_set: set[int] = set(int(i) for i in csv_text.split(","))
14 + as_set: set[int] = {int(i) for i in csv_text.split(",")}
15 | print(f"a generator converted to a set: {as_set}.")
note: This is an unsafe fix and may change runtime behavior
C402 Unnecessary generator (rewrite as a dict comprehension)
  --> generator_expressions_to_collection.py:22:27
21 | # This could be more efficiently replaced by dictionary comprehension.
22 | as_dict: dict[str, int] = dict((i, int(i)) for i in csv_text.split
    ↪ (","))
   |
      ↪ ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
23 | print(f"a generator of tuples converted to a dict: {as_dict}.")
help: Rewrite as a dict comprehension
21 | # This could be more efficiently replaced by dictionary comprehension.
   - as_dict: dict[str, int] = dict((i, int(i)) for i in csv_text.split
     ↪ (","))
22 + as_dict: dict[str, int] = {i: int(i) for i in csv_text.split(",")}
23 | print(f"a generator of tuples converted to a dict: {as_dict}.")
note: This is an unsafe fix and may change runtime behavior
Found 3 errors.
3 unsafe fixes available with `--fix --unsafe-fixes`.
# ruff 0.17.3 failed with exit code 1.
```

10.7 Generator Functions

The final element in Figure 10.1 that we did not yet discuss are *generator functions* [494]. From the perspective of the user of a generator function, it is a function that behaves like an `Iterator` of values. We can process the sequence of values provided by this `Iterator` in exactly the same ways already discussed. We can iterate over it using a `for` loop. We can use it as a comprehension or pass it to the constructor of a collection, if we want to.

From the perspective of the implementor, however, a generator function looks more like a function that can return values several times. Instead of using the `return` keyword, this is achieved by using the `yield` keyword. Each element of the sequence that we generate is produced by returning it via `yield`. This feels like we can define a function that can return a value, which is then processed by some outside code, and then the function *resumes* to return more values.

Since this sounds quite confusing, let's begin by looking at a very simple example. In Listing 10.30, we create a generator which should produce only the values 1, 2, and 3. It is implemented as a function `generator_123`, which is declared with `def` like any normal Python function. The return type is annotated with type hint `Generator[int]`, meaning that this is a generator function which produces `int` values. The function body consists only of the three statements `yield 1`, `yield 2`, and `yield 3`.

We can use the `Generator` returned by this function to populate a `list`: `list(generator_123())` results in the list `[1, 2, 3]`. Of course we can also manually iterate over the `Generator` in exactly the same way we could iterator over any normal `Iterator`. We therefore set `gen = generator_123()`. The

Listing 10.30: A very simple generator function yielding the numbers 1, 2, and 3. (stored in file simple_generator_function.py; output in Listing 10.31)

```
"""A simple example of generator functions."""

from typing import Generator  # The type hint for generators.


def generator_123() -> Generator[int]:
    """A generator function which yields 1, 2, 3."""
    yield 1  # The first time next(...) is called, the result is 1.
    yield 2  # The second time next(...) is called, the result is 2.
    yield 3  # The third time next(...) is called, the result is 3.


print(f"{list(generator_123()) = }")  # The list is [1, 2, 3].

gen: Generator[int] = generator_123()  # Use directly.
print(f"{next(gen) = }")               # First time next:  1
print(f"{next(gen) = }")               # Second time next: 2
print(f"{next(gen) = }", flush=True)   # Third time next:  3
print(f"{next(gen) = }", flush=True)   # raises StopIteration
```

↓ `python3 simple_generator_function.py` ↓

Listing 10.31: The stdout, stderr, and exit code of the program simple_generator_function.py given in Listing 10.30.

```
list(generator_123()) = [1, 2, 3]
next(gen) = 1
next(gen) = 2
next(gen) = 3
Traceback (most recent call last):
  File "{...}/iteration/simple_generator_function.py", line 19, in <module>
    print(f"{next(gen) = }", flush=True)  # raises StopIteration
          ~~~~~~~~~~
StopIteration
# 'python3 simple_generator_function.py' failed with exit code 1.
```



Figure 10.2: Leonardo Pisano / Fibonacci. Sculpture by Bertel Thorvaldsen, 1834/1838. Source: Thorvaldsens Museum exhibit A187, photographer Jakob Faurvig, under the Creative Commons Zero (CC0) license.

first time we invoke `next(gen)`, it returns `1`. The second time we invoke `next(gen)`, it returns `2`. The third time we invoke `next(gen)`, it returns `3`. The fourth call to `next(gen)` raises a `StopIteration`. This indicates that the end of the sequence is reached. Indeed, we queried the generator function's result exactly like a normal `Iterator`.

The interesting thing is that the function code is really disrupted by every `yield`. The value returned via `yield` is then received by the outside code, which do whatever it likes with it. The function is resumed after the `yield` only when `next` is called. When the control flow reaches the end of the function, the sequence ends as well. This becomes visible when we create a generator function that returns an infinite sequence.

Fibonacci, illustrated in Figure 10.2, was a medieval Italian mathematician who lived in the 12th and 13th century CE [97]. He wrote the book *Liber Abaci* [512], which introduced Arabic numbers to Europe. He is also known for the Fibonacci number sequence, which follow the sequence $F_n = F_{n-1} + F_{n-2}$ where $F_0 = 0$ and $F_1 = 1$ [512, 623].

We want to iterate over this sequence. We therefore define the function `fibonacci`, which is annotated to return a `Generator`. It begins by setting `i` $= F_0 = 0$ and `j` $= F_1 = 1$. It then begins a `while` loop which will never stop, as the loop condition is imply set to `True`. In this loop, it always yields `i`.

This means that the current function execution will be interrupted. The value of `i` is provided to the outside code which iterates over our sequence. After that code invokes `next`, the function resumes. Then, it assigns `j` and `i + j` to `i` and `j`, respectively. This stores the old value of `j` in `i`. This also stores the sum of the old `i` and `j` values in `j`. In the next loop iteration, `yield i` will then produce the next Fibonacci number.

We can now loop over the `Generator` returned by `fibonacci()` with a normal `for` loop. This would result in an endless loop, unless we insert some additional termination condition. In our loop, we print the Fibonacci numbers `a` we get. However, we stop the iteration via `break` once `a > 30`.

It should be mentioned that doing something like `list(fibonacci())` would be a very bad idea. It would attempt to produce an infinitely large list, which would lead to an `MemoryError`.

As the final example of generator functions, let us wrap our code for enumerating prime numbers [135, 481, 626] from back in Section 7.4 into a generator function. We do this in `prime_generator.py` given as Listing 10.34. Back in Listing 7.11, when we produced prime numbers for the last time, we used nested `for` loops. Back then, the outer loop would iterate at most to 199.

Now we do not need such a limit anymore. We simply assume that whoever will call our prime number enumeration code will stop iterating whenever they have seen sufficiently many primes. A `while True` loop therefore will be more appropriate. We also do not need to produce a list, so we do not need to store the only even prime number 2 anywhere. Instead, we just `yield 2` at the beginning of our generator and move on. The last important difference between the old and new code is that,

Listing 10.32: A generator function yielding the infinite sequence of Fibonacci numbers [512, 623]. (stored in file `fibonacci_generator.py`; output in Listing 10.33)

```
"""A generator functions iterating over the Fibonacci Sequence."""

from typing import Generator  # The type hint for generators.


def fibonacci() -> Generator[int]:
    """A generator returning Fibonacci numbers."""
    i: int = 0   # Initialize i.
    j: int = 1   # Initialize j.
    while True:  # Loop forever, i.e., generator can continue forever.
        yield i  # Return the Fibonacci number.
        i, j = j, i + j  # i = old_j and j = old_i + old_j


for a in fibonacci():  # Loop over the generated sequence.
    print(a)  # Print the sequence element.
    if a > 30:  # If a > 300, then
        break    # we stop the iteration.

# list(fibonacci()) <-- This would fail!!
```

↓ `python3 fibonacci_generator.py` ↓

Listing 10.33: The stdout of the program `fibonacci_generator.py` given in Listing 10.32.

```
0
1
1
2
3
5
8
13
21
34
```

once we confirm a number to be prime, we do not just append it to the list `found` of odd primes, we also need to `yield` it.

Apart from this, the function works pretty much the same as last time. We iterate over the odd numbers `candidate` that could be a prime number. For each `candidate`, we test all previously identified prime numbers `check` (except 2, of course), whether they can divide `candidate` with a remainder of zero. The first time we encounter such a number, we know that `candidate` cannot be a prime number. If no number `check` exists that is a divisor of `candidate`, then `candidate` is prime.

We only need to test values of `check` that are less or equal to `isqrt(candidate)`, i.e., $\leq \lfloor\sqrt{\texttt{candidate}}\rfloor$. Numbers larger than that do not need to be tested. They would have resulted in a quotient smaller than `check` that would have already been tested. In summary, our function will find one prime after the other and `yield` it.

Every time we `yield` a value, our function is interrupted. The first time, this happens when we `yield 2`. Then, it happens in each iteration of the main loop that discovers a prime number. Every time our function is interrupted like this, the corresponding value is returned to the outside code iterating over our sequence. While implementing the function, we have no idea what that code does, well, except that it will apply `next` to the generator... But we also do not need to know what it does. All we care is that, if it invokes `next`, our function will continue with the instruction following right after the `yield` and run until it hits the next `yield`.

We demonstrate how our generator function works with a doctest. The test begins by instantiating the `Generator` as `gen = primes()`. The first `next(gen)` call is supposed to return `2`. The second such call shall return `3`, the third one `5`, the fourth one `7`. The fifth and last `next(gen)` invocation

Listing 10.34: A generator function yielding the infinite sequence of prime numbers [135, 481, 626]. (src)

```
"""A generator function for prime numbers."""

from math import isqrt  # The integer square root function.
from typing import Generator  # The type hint for generators.


def primes() -> Generator[int]:
    """
    Provide a sequence of prime numbers.

    >>> gen = primes()
    >>> next(gen)
    2
    >>> next(gen)
    3
    >>> next(gen)
    5
    >>> next(gen)
    7
    >>> next(gen)
    11
    """
    yield 2  # The first and only even prime number.

    found: list[int] = []  # The list of already discovered primes.
    candidate: int = 1     # The current prime candidate
    while True:            # Loop over candidates.
        candidate += 2  # Move to the next odd number as prime candidate
        is_prime: bool = True  # Let us assume that `candidate` is prime
        limit: int = isqrt(candidate)  # Get maximum possible divisor.
        for check in found:  # We only test with the odd primes we got.
            if check > limit:  # If the potential divisor is too big,
                break          # then we can stop the inner loop here.
            if candidate % check == 0:  # division without remainder
                is_prime = False  # check divides candidate evenly, so
                break  # candidate is not a prime. Stop the inner loop.

        if is_prime:  # If True: no smaller number divides candidate.
            yield candidate  # Return the prime number as next element.
            found.append(candidate)  # Store candidate in primes list.
```

in the doctest should return `11`. You can use `pytest` by yourself to check whether the code works as expected... Well, OK, I tell you: It does, as shown in Listing 10.35.

From the perspective of a user, a generator function works like a generator expression, which, in turn, works like an iterator. Compared to generator expressions, generator functions are much more powerful. We can package arbitrarily complex code into such a function. This code can have one or multiple points where it returns (that is, yields) results to the outside caller.

A normal function can have arbitrarily many `return` statements. However, the execution of a normal function is completed and ends when the first `return` is executed. A generator function can have arbitrarily many `yield` statements. Each statement returns a value to the outside caller. However, the generator function can continue until either its end is reached or until the outside user stops iterating over the sequence it presents.

Listing 10.35: The output of pytest executing the doctests for the prime number generator function given as in Listing 10.34: The test succeeded.

```
$ pytest --timeout=10 --no-header --tb=short --doctest-modules
    ↪ prime_generator.py
========================== test session starts ==========================
collected 1 item
prime_generator.py .                                                [100%]
=========================== 1 passed in 0.01s ===========================
# pytest 9.2.0 with pytest-timeout 2.4.0 succeeded with exit code 0.
```

10.8 Operations on Iterators

Sequences play a very important role in Python programming. The fact that generator functions and the `yield` keyword were added to the language just to provide a natural way to construct complex sequences speaks for itself [494]. Naturally, there also are several utilities for working with and transforming sequences. Some of these tools are directly provided as built-in functions, some ship with the module `itertools` [290]. Here we want to discuss a few of them.

The first two tools we will look at are the built-in function `filter` and the `takewhile` function from the `itertools` module. In the previous section, we implemented a generator function returning the endless sequence of prime numbers in file `prime_generator.py` in Listing 10.34. What would we do if we wanted a convenient way to create a list of all prime numbers which are less than 50 by using this never-ending generator? The answer can be found in our program `filter_takewhile.py` given in Listing 10.36.

`takewhile` is a function with two parameters [290]. The second parameter is an `Iterable`. Let's say that this `Iterable` provides a sequence of elements of some type `T`. The first parameter then is a predicate, which is a function accepting one element of type `T` and returning a `bool` value. Then,

Listing 10.36: An example of `takewhile` and `filter`. (stored in file `filter_takewhile.py`; output in Listing 10.37)

```
"""Examples of `takewhile` and `filter`."""

from itertools import takewhile  # takes items while a condition is met
from math import isqrt  # the integer square root

from prime_generator import primes  # prime number generator function

# First, we want to create a list with all prime numbers less than 50.
# This can be done using `takewhile` and a lambda expression.
less_than_50: list[int] = list(takewhile(lambda z: z < 50, primes()))
print(f"primes less than 50: {less_than_50}")

# Now we want to find the prime numbers `x` < 1000 that have the form
# `x = y² + 1` where `y` must be an integer number. For the latter
# condition, we use a lambda expression and the `filter` function.
sqrs_plus_1: tuple[int] = tuple(
    filter(lambda x: x == isqrt(x) ** 2 + 1,  # check if x has form y²+1
           takewhile(lambda z: z < 1000, primes())))  # Only z < 1000
print(f"primes less than 1000 and of the form x²+1: {sqrs_plus_1}")
```

↓ `python3 filter_takewhile.py` ↓

Listing 10.37: The stdout of the program `filter_takewhile.py` given in Listing 10.36.

```
primes less than 50: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43,
    ↪ 47]
primes less than 1000 and of the form x²+1: (2, 5, 17, 37, 101, 197, 257,
    ↪ 401, 577, 677)
```

Listing 10.38: An example of the function `map`. (stored in file `map.py`; output in Listing 10.39)

```
"""Examples of `map`: Transform the elements of sequences."""

# A string with comma-separated values (csv).
csv_text: str = "12,23,445,3,12,6,-3,5"

# Convert the csv data to ints by using `split` and `map`, then filter.
for k in filter(lambda x: x > 20, map(int, csv_text.split(","))):
    print(f"found value {k}.")

# Obtain all unique square numbers using `map` and `set`.
csv_text_sqrs: set[int] = set(map(
    lambda x: int(x) ** 2, csv_text.split(",")))
print(f"csv_text_sqrs: {csv_text_sqrs}")

# Get the maximum word length by using `map`, `len`, and `max`.
words: list[str] = ["Hello", "world", "How", "are", "you"]
print(f"longest word length: {max(map(len, words))}")
```

↓ `python3 map.py` ↓

Listing 10.39: The stdout of the program `map.py` given in Listing 10.38.

```
found value 23.
found value 445.
csv_text_sqrs: {36, 198025, 9, 144, 529, 25}
longest word length: 5
```

`takewhile` constructs a *new* `Iterator`, which returns the elements from the original `Iterable` as long as the predicate function returns `True` for them. As soon as it hits an element from the original `Iterable` for which the predicate returns `False`, it will stop the iteration.

Back in Section 8.5, we learned that we can also pass functions or `lambda`s as arguments to other functions. This is a practical example where `lambda`s come in especially handy. Therefore, the answer to "How can I extract all the numbers less than 50 from the prime sequence?" is simply to call `takewhile(lambda z: z < 50, primes())`. This sequence is now no longer infinitely long and can conveniently be converted to a `list`.

The built-in function `filter` works quite similarly [83]. It, too, accepts a predicate and an `Iterable` as input. Different from `takewhile`, the new `Iterator` created by `filter` does not stop if the predicate returns `False`. However, it returns only those elements for which the predicate returned `True`. In Listing 10.36, we use this to select prime numbers x for which an integer y exists such that $x = y^2 + 1$. We again implement the predicate as `lambda`. Since the underlying `Iterator` given by `primes()` is infinite, we again use `takewhile` to limit the sequence to those primes that are less than 1000. We pass the result of our construct to the function `tuple`, which creates an immutable and indexable sequence. The results can be seen in Listing 10.37: There are ten such primes. The smallest one is $1^2 + 1 = 2$ and the largest one is $26^2 + 1 = 677$.

Another important utility function when dealing with sequences is the built-in function `map` [83]. We explore its use in `map.py` given as Listing 10.38. Back in Listing 10.27, we used a generator expression to process data that we extracted from a CSV-formatted string. Instead of doing `int(s) for s in csv_text.split(",")` we can simply write `map(int, csv_text.split(","))`. The first argument to `map` is a function that should be applied all of the elements in the `Iterable` passed in as its second argument. The result of `map` is a new `Iterator` with the return values of this function.

In `map.py` given Listing 10.38, we first split `csv_text` at all `","`. We then translate the elements of the resulting list to `int`s via `map`. Finally, we `filter` the sequence to retain only values greater than 20. We can conveniently iterate over the resulting filtered and mapped sequence using a `for` loop.

How about we now obtain all the squares of the values in the CSV data, but each value only once. In other words, we discard all duplicates. First, we again use `split` to divide the text into chunks based on the separator `","`. Then we map these chunks to integers and return their squares using the `map` function, but this time we provide a `lambda` that does the transformation.

Listing 10.40: An example of the function `zip`. (src)

```
"""An examples of `zip`: Compute the distance between two points."""

from math import sqrt
from typing import Iterable


def distance(p1: Iterable[int | float],
             p2: Iterable[int | float]) -> float:
    """
    Compute the distance between two points.

    :param p1: the coordinates of the first point
    :param p2: the coordinate of the second point
    :return: the point distance

    >>> distance([1, 1], [1, 1])
    0.0

    >>> distance((0.0, 1.0, 2.0, 3.0), (1.0, 2.0, 3.0, 4.0))
    2.0

    >>> distance([100], [10])
    90.0

    >>> try:
    ...     distance([1, 2, 3], [4, 5])
    ... except ValueError as ve:
    ...     print(ve)
    zip() argument 2 is shorter than argument 1
    """
    return sqrt(sum((a - b) ** 2 for a, b in zip(p1, p2, strict=True)))
```

Listing 10.41: The output of pytest executing the doctests for the `zip` example from Listing 10.40.

```
$ pytest --timeout=10 --no-header --tb=short --doctest-modules zip.py
============================ test session starts ============================
collected 1 item
zip.py .                                                              [100%]
============================ 1 passed in 0.01s =============================
# pytest 9.2.0 with pytest-timeout 2.4.0 succeeded with exit code 0.
```

Now we want to retain only the unique values, i.e., want to get a duplicate-free collection of these numbers. This can be done by passing the resulting `Iterator` into the `set` constructor. A set, by definition, only contains unique values. In the resulting output in Listing 10.39, we can see that `9` indeed only appears once and so does `144`.

Finally, the `map` function also plays nicely together with aggregating functions like `sum`, `min`, or `max`. In the final example of `map`, we have a list `words` of words and want to know the length of the longest word. We can first map each word to its length via `map(len, words)`. This produces an `Iterator` of word lengths, which we can directly pass to `max`. `max` will then iterate over this sequence and return the largest value it encountered.

Notice that `map` does not generate a data structure with all the transformed elements in memory. Instead, the elements are constructed as needed (and thereafter disposed by the garbage collection when no longer needed). This makes `map` an elegant and efficient approach to transforming sequences of data.

As last example of sequence processing we play a bit with the `zip` function [83]. This function accepts several `Iterables` as arguments and returns a new `Iterator` which steps through

all of input iterables in synch, returning tuples with one value of each of them. For example, `zip([1, 2, 3], ["a", "b", "c"])` returns an `Iterator` that produces the sequence `(1, "a")`, `(2, "b")`, and `(3, "c")`. Sometimes, the input `Iterables` may be of different length. To make sure that such an error is properly reported with a `ValueError`, we must always supply the named argument `strict=True` [80].

In `zip.py`, provided as Listing 10.40, we use `zip` to implement a function `distance` that computes the Euclidean distance of two n-dimensional vectors or points `p1` and `p2`. The two points are supplied as `Iterables` of either `float` or `int`. We could, for example, provide them as `listss` The Euclidean distance is defined as

$$\texttt{distance(p1, p2)} = \sqrt{\sum_{i=1}^{n}(\texttt{p1}_i - \texttt{p2}_i)^2} \tag{10.1}$$

This means that we need to iterate over both points in lockstep. This is exactly what `zip` does. If both points were provides as `lists`, then `zip(p1, p2, strict=True)` will, step by step, give us the tuples `(p1[0], p2[0])`, `(p1[1], p2[1])`, ..., until reaching the ends of the lists. We can now write the generator expression `(a - b)** 2 for a, b in zip(p1, p2, strict=True)`. It uses tuple unpacking to extract the two elements `a` and `b` from each of the tuples that `zip` creates. It then computes the square of the difference of these two elements. By passing the generator expression to the `sum` function as-is, we can get the sum of these squares. Finally, the `sqrt` function from the `math` completes the computation of the Euclidean distance as prescribed in Equation 10.1.

Instead of testing this new function `distance` with a small example program, we do so with doctests. The doctest shows that the expected distance of two identical vectors with the same value `[1, 1]` should be `0.0`. `distance((0.0, 1.0, 2.0, 3.0), (1.0, 2.0, 3.0, 4.0))`, which basically is $\sqrt{1+1+1+1}$ should be `2.0`. The distance of the two one-dimensional vectors `[100]` and `[10]` should be `90.0`. If we, however, pass in two vectors with different dimensions, this should result in a `ValueError`. The output of `pytest` in Listing 10.41 shows that the example cases all return their expected results.

This concludes our treatment of operations on `Iterators`. We could only scratch the surface here. The module `itertools` [290] which ships with `Python` offers many more useful functions. However, an understanding of the principles of `map`, `filter`, and `zip` will enable the reader to explore these tools by themselves.

10.9 Summary

Working with sequences is a very important aspect of `Python` programming. The programming language provides a simplified syntax for working with loops in form of list-, set-, and dictionary comprehension. Different from comprehension, generator expressions allow us to provide sequences of data that can be processed without storing all elements in memory first or at once. Instead, the elements are created when needed. If this creation of elements is more complicated than what simple generator expressions can, well, express, we can use generator functions. With their `yield` statement, they allow us to write functions that perform a computation, pass the result to their output and allow the calling code outside to process them. Different from normal functions, these generator functions can then resume their execution until they return more results via `yield` or reach the end of their sequence. Sequences of data can be processed by aggregating and transforming functions. These functions can process containers, comprehensions, generator expressions, and generator functions alike. This is possible because all of the sequence API boils down to two basic components: `Iterator` and `Iterable`. An `Iterable` is an interface supported by any object whose elements can be accessed one-by-one. An `Iterator` is a single pass, a single such access sequence.

Part III

Classes

Chapter 11

Basics of Classes

We learned about simple datatypes in Chapter 3. We then also got to know different kinds of collections, i.e., datatypes that can hold multiple elements, in Chapter 5. However, in many situations, we deal with data that cannot satisfyingly represented by either of them alone. Many datatypes are compound structures of different elements that are semantically related to each other. For example, a the elements of a list or tuple are related only in the sense that they appear in the same collection. If we look at a *date* however, then there is a much more meaningful relationship between its components *day*, `month`, and `year`. Such datatypes and the operations on them also often form a semantic unit.

Imagine, for example, you would like to implement mathematical operations for complex numbers.[1] Of course, you could represent complex numbers as `tuple[float, float]`. This, however, comes with several drawbacks.

On the one hand, not *every* tuple of two `floats` is to be understood as a complex number. So just from the signature of your functions, i.e., just based on its parameters and return types, it is not immediately clear that your functions are for complex numbers. All we can directly see is that they are for tuples of two `floats`.

On the other hand, the two parts of a complex numbers, the real part and the imaginary part, have two distinct and different meanings. It would not immediately be clear whether the first number of the tuple is the real or the imaginary part. Indeed, we could also represent complex numbers in polar form, in which case the two tuple elements would have yet different meanings. Also, the standard textual representation of tuples of two `floats` would be something like `"(3.0, 4.0)"`, whereas we would probably prefer something like `"3+4i"` for complex numbers.

The first important use-cases of `classes` in `Python` is that they offer us a way to define a data structure together with the operations for it as one semantic unit [120]. This allows us to define a `class` for complex numbers which has attributes with the names `real_part` and `imaginary_part`. We can define operators like addition and subtraction that work with instances of this `class`, making it immediately clear how and when they are to be used. And this `class` can then have default textual representations of our liking.

A second situation where ability to define functions and we have learned so far hits a limit arises with APIs that have different implementations. Let us be ambitious and imagine that you wanted to create a versatile system that can produce documents. On the output side, you want to support different formats, say LibreOffice [215, 356], Microsoft `Word` [166, 396], and Adobe PDF [205, 630]. On the input side, you would like to provide the user/programmer with a uniform way to create documents.

This input side API should be the same for all output formats. It would not just consist of a single function, but several groups of functions. There could even be nested hierarchies of operations that you would like to support, e.g., the add chapters, paragraphs, and maybe runs of text in different fonts. Obviously, these operations will be implemented differently for the different output formats.

We could try to solve this by making different modules for different output formats. Each modul could contain functions of the same name and signatures, realizing the required behaviors. But this would be a huge hassle. The bibbest problem would be that there would be no way to define a central blueprint of "how the API looks like." It can easily lead to inconsistencies during the software life cycle. If we slightly change the signature of one function, we need to implement this in all of the modules. There also is no way that a linter like `Ruff` could tell is if some module is no longer synchronized due

[1] `Python` already offers the datatype `complex` for this purpose, but for the sake of the example, assume it does not.

to the lack of a central blueprint.

Classes offer us the necessary abstraction. We could specify a base class `Document` for document objects that provides methods for the necessary operations, from adding text to aligning figures. These operations could just `raise` a `NotImplementedError`. Then, for each output format, we could derive a new subclass from this base class with the actual implementation of the methods. The code on the user side could treat all these different document types the same, because all of them would be instances of `Document` with the same operations. The format-specific stuff would all be invisible to the user, exactly as it should be. Linters then can also tell us if one certain implementation of the subclass does not correctly adher to the API defined in the base class. So the second important use case for classes are that they offer us a very nice abstraction for defining and implementing APIs.

Classes therefore can solve two important problems where the basic datatypes, collections, and plain functions we learned about are not sufficient: First, they allow us to semantically and clearly group data and operations together. Second, they offer us a simple way to group several operations into one API, which then – transparently to the user – can be implemented in different ways. In this chapter, we discuss `classes` in Python. Syntactly, `classeses` follow the general blueprint below [120].

```
"""The basic syntax for defining classes in Python."""

class MyClass:    # or `class MyClass(MyBaseClass)`
    """The docstring of the class."""

    def __init__(self, param1: type_hint) -> None:
        """The docstring of the initializer __init__."""
        # In this method, we initialize all the attributes of the class.
        # Each attribute should get an initial value, `None` if need be.

        #: Documentation of the meaning of attribute 1 (notice the ":"!)
        self.attribute_1: type_hint = initial value
        #...

    def my_method(self, param1: type_hint, param2: type_hint) -> result:
        """
        Docstring of my_method.

        :param param1: the documentation of the first parameter.
        :param param2: the documentation of the second parameter.
        :returns: the documentation of the result of the method.
        """
        # compute something using the attributes
        self.attribute_1 = ...    # Assign value to attribute.
        x = self.attribute_1      # Use the value of an attribute.
        self.my_other_method(12)  # Call other methods of the class.

    #... more methods


# Instantiating MyClass creates a new instance of MyClass.
# We can use MyClass as type hint for variables.
newVar: MyClass = MyClass(value for param1 of __init__)
```

Classes are datatypes that combine data elements with the code working on them [120]. A class is basically a blueprint or a concept, whereas objects are the concrete instances of the classes. For example `int` is basically a class of integer numbers, whereas `5` is an instance of this class.

Classes are declared with the keyword `class` followed by the class name, followed by a colon (":"). The class body is then indented by four spaces. It contains everything that belongs to the class, including the documentation, methods, and attributes. The first item after the class declaration is usually the docstring of the class. It can be a single descriptive line or a multi-line documentation, in which case it begins with a single summary line, followed by an empty line, followed by the comprehensive documentation text.

Best Practice 57

Class names should follow the CapWords convention (often also called *camel case* [656]), i.e., look like `MyClass` or `UniversityDepartment` (not `my_class` or `university_department`) [594].

A class can have a initializer method `__init__`. This special method can take arbitrarily many parameters, but never returns a value. It is annotated with type hints and with a docstring like a normal function. Only the special parameter `self` is not annotated.

The initializer `__init__` is used to declare all of the classes' (non-inherited) instance attributes by assigning them their initial values. We also provide attribute type hints in this step. In all methods of the class, the current object/instance of the class is referenced by the name `self`. If we access an attribute or a method of a class, we always use the prefix `self.`. We therefore always declare `self` as the first parameter of every method. Thus, a line like `self.x: int = 5` in `__init__`, creates the instance attribute `x`, type-hints it as integer, and assigns to it the initial value 5. We can put a short comment describing the meaning of the attribute in the line *before* its declaration. This comment will always have a colon right after the hash, i.e. always begins with `#:`[525]. For example, writing `#: This is the x-coordinate.` in the line before the declaration of an attribute `self.x` would annotate this attribute with documentation stating that it is, well, an x-coordinate.

Classes can have arbitrarily many methods. A method is a function that works on or with the attributes of a class instance. Each such method has as first parameter `self`, denoting the object instance of the class the method works on. A method can have arbitrarily many other parameters and, if you want, also a return value. All parameters except `self` are of course annotated with type hints. Methods also have docstrings, like normal functions. In the method, we can access the both the attributes and methods of the instance again via the `self.`-prefix.

After defining the class, we can now instantiate it. For this, we use the class name like a normal function. We need to provide arguments, i.e., values, for all parameters of `__init__` except `self`. We can use objects of this class like normal values, and, e.g., store them in variables. We can also use the class name as a type hint, because it represents a normal type.

So much about the general structure of `classes`. Now, without further ado, let's explore this structure with some examples.

11.1 Design Principle: Immutable Classes

We begin directly with an example. Assume that we want to write a program that processes points in the two-dimensional Euclidean plane. Every point is defined by its x and y-coordinate. We could use a `tuple` of two numbers, say a `tuple[int |float, int |float]`, to represent such points. This is totally fine and a quick solution.

However, this solution lacks semantics, i.e., it lacks a clear and obvious meaning. There is nothing that says that a `tuple[int |float, int |float]` must be a point in the two-dimensional Euclidean plane. It could just as well be a tuple of travel time and an associated cost for a train ride from Hefei to Beijing, taken from the train schedule of a particular day. Basically, it just is a grouping of two numbers. The same lack of semantics appears when we try to implement operations that process our points. A function that computes the distance between two such points would just take two `tuples` as input. Of course, I should pass in only `tuples` that actually represent points in the two-dimensional plane. But nothing really can stop me from passing in other tuples, such as `tuples` holding the travel times and associated costs for train rides from Hefei to Beijing. Of course, the results would probably not make any sense. Yet, such situations may arise from misunderstandings or lack of documentation.

In the ideal case, if I am working with points in the two-dimensional Euclidean plane, then I have a data structure that clearly and unambiguously is designed for such points and such points only. The operations for the points should only accept instances of this data structure as input (and raise exceptions if fed something else as arguments). If I am accessing the x-coordinate of a point, it should be absolutely clear from the semantics and names involved that this is, indeed, the x-coordinate and not something else, not just the first number of a `tuple` of two numbers.

Such clear semantics can be achieved with `classes` in Python. In Listing 11.1, you can find exactly this example as file `point.py`. We create the class `Point` by writing `class Point:`. We then create

Listing 11.1: A class for representing points in the two-dimensional Euclidean plane (src)

```
"""A simple class for points."""

from math import isfinite, sqrt
from typing import Final


class Point:
    """
    A class for representing a point in the two-dimensional plane.

    >>> p = Point(1, 2.5)
    >>> p.x
    1
    >>> p.y
    2.5

    >>> try:
    ...     Point(1, 1e308 * 1e308)
    ... except ValueError as ve:
    ...     print(ve)
    x=1 and y=inf must both be finite.
    """

    def __init__(self, x: int | float, y: int | float) -> None:
        """
        The constructor: Create a point and set its coordinates.

        :param x: the x-coordinate of the point
        :param y: the y-coordinate of the point
        """
        if not (isfinite(x) and isfinite(y)):
            raise ValueError(f"x={x} and y={y} must both be finite.")
        #: the x-coordinate of the point
        self.x: Final[int | float] = x
        #: the y-coordinate of the point
        self.y: Final[int | float] = y

    def distance(self, p: "Point") -> float:
        """
        Get the distance to another point.

        :param p: the other point
        :return: the distance

        >>> Point(1, 1).distance(Point(4, 4))
        4.242640687119285
        """
        return sqrt((self.x - p.x) ** 2 + (self.y - p.y) ** 2)
```

the class body, which we indent by four spaces. Of course the first thing we place in the class body is always a docstring.

Best Practice 58

At the beginning of a `class`, a docstring is placed which describes what the class is good for. This docstring can include doctests to demonstrate the class usage. Such doctests can also be placed in the docstring of the module.

Afterwards, we will place all the *methods* of the `class` in the class body. Methods are like functions,

Listing 11.2: An example of using our new class `Point` from Listing 11.1. (stored in file `point_user.py`; output in Listing 11.3)

```
"""Examples of using our class :class:`Point`."""

from point import Point             # Import our class from its module.

p1: Point = Point(3, 5)             # Create a first instance of Point.
print(f"{p1.x = }, {p1.y = }")      # p1.x = 3, p1.y = 5

print(f"{type(p1) = }")                 # <class 'point.Point'>
print(f"{isinstance(p1, Point) = }")    # Hence, this is True.
print(f"{isinstance(5, Point) = }")     # This is obviously False.
print(f"{isinstance(p1, int) = }")      # This is obviously False, too.

p2: Point = Point(x=7, y=8)         # Create a second instance of Point.
print(f"{p2.x = }, {p2.y = }")      # p2.x = 7, p2.y = 8
print(f"{type(p2) = }")             # <class 'point.Point'>

print(f"{p1 is p1 = }")  # True, because p1 is the same as p1.
print(f"{p1 is p2 = }")  # False, as these are two different instances.

print(f"{p1.distance(p2) = }")   # sqrt(4^2 + 3^2) = 5.0
print(f"{p2.distance(p1) = }")   # sqrt(4^2 + 3^2) = 5.0

point_list: list[Point] = [  # Create list of points via comprehension.
    Point(x, y) for x in range(3) for y in range(2)]
print(", ".join(f"({p.x}, {p.y})" for p in point_list))
```

↓ `python3 point_user.py` ↓

Listing 11.3: The stdout of the program `point_user.py` given in Listing 11.2.

```
p1.x = 3, p1.y = 5
type(p1) = <class 'point.Point'>
isinstance(p1, Point) = True
isinstance(5, Point) = False
isinstance(p1, int) = False
p2.x = 7, p2.y = 8
type(p2) = <class 'point.Point'>
p1 is p1 = True
p1 is p2 = False
p1.distance(p2) = 5.0
p2.distance(p1) = 5.0
(0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)
```

but their first parameter is always called `self` and always is an instance of the class, i.e., an object. Either way, all these methods go into the class block.

Our class `Point` will have two attributes, `x` and `y`. A attribute is a variable that every single instance of the class has. We later want to be able to create one instance of `Point` with the x-coordinate 5 and the y-coordinate 10 and then another instance with the x-coordinate 2 and the y-coordinate 7. So each instance of `Point` needs to have these two attributes.

Therefore, `Point` needs a initializer, i.e., a special method that creates and initializes these attributes. This method is called `__init__`. As said, every method of a class must have a parameter `self`, which is the instance of the class (the object) upon which the method is called. The initializer `__init__` is a special method, so it also has this parameter `self`. Additionally, we demand that values for the two parameters `x` and `y` be passed in when we create an instance of `Point`. We allow the values to be either `ints` or `floats`.

Inside every method of a class, the attributes of objects are accessed via the parameter `self`. We can read the attribute `x` of an object inside a method of the class by writing `self.x`. Here, `self.x` can

be used just like a normal local variable. We can store the value `a` in a (mutable) attribute `x` of the current object in a method of the class by writing `self.x = a`. This value will then remain the same until it is changed in exactly that way, even after the execution of the method is completed.

Best Practice 59

Object attributes must only be created inside the initializer `__init__`. There, an initial value must immediately be assigned to each attribute.

We only want to allow `Points` that have finite coordinates. As explained in the Chapter 9 on `Exceptions`, it is better to immediately signal errors if we encounter invalid data. So we want to sort out non-finite coordinates right when someone attempts to create the `Point` object. Thus, the first thing we do in the initializer is to use the `isfinite` function from the `math` module. If either `x` or `y` is not finite, then we raise a `ValueError`.[2]

Otherwise, we set `self.x: Final[int |float] = x` and `self.y: Final[int |float] = y`. These lines create the attributes `self.x` and `self.y` of the object which was passed in via the parameter `self`. The type hint `Final` from the `typing` module annotates a variable or attribute as immutable [546]. In other words, we do not allow the coordinates of our `Points` to change after object creation.

Best Practice 60

Every attribute of an object must be annotated with a type hint when created in the initializer `__init__` [348]. Here, type hints work exactly like with normal variables.

Best Practice 61

The type hint `Final` marks a variable or attribute as immutable. All attributes that you do not intend to change should be annotated with `Final`. Notice that this is a type hint, i.e., it will not be enforced by the Python interpreter [546] and malicious code can *still* change the attribute value. Type checkers like Mypy can detect such incorrect changes, though, and issue warnings.

We will explore this issue a bit later in detail.

Best Practice 62

An attribute is documented in the line *above* the attribute initialization by writing a *comment* starting with `#:`, which explains the meaning of the attribute [525]. (Sometimes, the documentation is given as string directly below the attribute definition [235], but we stick to the former method, because it has proper tool support, e.g., by Sphinx.)

After properly defining our initializer, we can now do something like `p = Point(1, 2)`. This creates a new object as an instance of our `class` `Point`. Therefore, first, the necessary memory for `p` is allocated. Then, the initializer is invoked as `__init__(p, 1, 2)`. After the initializer is completed, the object is stored in the varible, i.e., `p` now refers to a `Point` object. The attribute `p.x` has the value `1` and `p.y` has value `2`.

From the knowledge that `p` is an instance of `Point`, we can immediately see that `p.x` and `p.y` are its x- and y-coordinate, respectively. There is no way to mistake the meaning of these variables. Of course, our docstrings with doctests and the type hints further help the reader to understand their meaning.

Having a new class for points in the two-dimensional plane is already nice. But such a `class` also allows us to define operations on points in form of methods. As an example, we implement a method

[2] Strictly speaking, it could also make sense to check whether `x` and `y` are indeed either `int` or `float`. For example, we could check if `isinstance(x, int |float)` and this is `False`, raise a `TypeError`. However, that would make the example overly long, so I am not doing this here.

`distance` that computes the distance between two points. You would have a point `p1` and invoke `p1.distance(p2)` to compute the distance to another point `p2`. The computation itself will follow Equation 10.1 from our recent endeavor to operations on iterations in Section 10.8. We therefore need to import the `sqrt` function from the `math` module.

Our new method `distance` will have two parameters, `self`, which will be the object upon which we invoke the method (`p1` in the above example) and `p`, the other object (or `p2` above). It then just has to compute the Euclidean distance `sqrt((self.x - p.x)** 2 + (self.y - p.y)** 2)`. Inside a method of an object, `self` always refers to the object itself. Therefore, `self.x` is the x-coordinate of the current object and `self.y` is its y-coordinate. `p.x` is the x-coordinate of the point `p` that was passed in as actual parameter of the method, and `p.y` is its y-coordinate. Notice that the docstring not just explains how this method is used, but also provides a simple example in form of a doctest. If you compute `Point(1, 1).distance(Point(4, 4))`, then the expected result is something like 4.243...

In this doctest – `Point(1, 1).distance(Point(4, 4))` – we only provided a single parameter to the method `distance`. When calling the method `distance`, we never need to provide a value of the parameter `self` directly. Instead, it will be provided indirectly: If we have two points `p1` and `p2` and invoke `p1.distance(p2)`, then `self = p1` will be set automatically. Hence, even though we declared our method as `def distance(self, p: "Point")-> float`, which looks as if we need to provide two parameters (`self` and `p`), we only need to provide one, namely `p`.

Reading this again, we notice that the parameter `p` is annotated with a very strange type hint: One would expect that we would annotate it with `Point`, instead it is annotated with the string `"Point"`. This has the simple reason that the complete class `Point` is only defined *after*, well, the complete definition of class `Point`. Therefore, `Point` is not yet available as type *inside* its definition. Using the string `"Point"` here is therefore just a crutch with no real other effect.

Best Practice 63

All methods of `classes` must be annotated with docstrings and type hints.

Best Practice 64

When using a class `C` as type hint *inside* the definition of the class `C`, you must write `"C"` instead of `C`. (Otherwise, static code analysis tools and the Python interpreter get confused.)

We could now go on and add more methods that do reasonable computations with instances of `Point`. For now, this simple example will suffice. Let us instead take a look on how one would use our new class with program `point_user.py` given as Listing 11.2.

Before we can do anything, we need to import our class `Point`. Class `Point` is defined in file `point.py`. The file name without the `.py` is the module name (`point`) from where we can import the class. Hence, we write `from point import Point`.

We now can create one instance of `Point` and store it in a variable `p1`. Since `p1` is supposed to reference an instance of `Point`, its type hint should denote this. Here we can use `Point` just like any other datatype. We write `p1: Point = Point(3, 5)`. The initializer `__init__` is automatically invoked when we write `Point(3, 5)`. The two arguments that we pass in will be provided to it as values for parameters `x` and `y`, respectively. The first parameter, `self`, of `__init__` will be a newly allocated and yet un-initialized instance of `Point`.

After `__init__` completes, the resulting new instance of `Point` that we get should have its attribute `x` set to `3` and its attribute `y` set to `5`. We can access them via `p1.x` and `p1.y`. Of course we can also use them in an f-string. We find that `f"{p1.x = }, {p1.y = }"` will be interpolated to `"p1.x = 3, p1.y = 5"`.

The type of `p1` is `Point`. The class `Point` is defined in file `point.py`. This file name is interpreted as module name `point`. Hence, the full name of the type is `point.Point`. And it is a `class`. Hence, printing `type(p1)` yields `<class 'point.Point'>`.

We can check if an object `o` is an instance of our class `Point` by writing `isinstance(o, Point)`. For `p1`, this returns `True`, as one would expect. As a test, we check whether `isinstance(5, Point)`, which returns `False` for similarly obvious reasons. `isinstance(p1, int)` is `False`, too.

Listing 11.4: An example program that uses our new class `Point` incorrectly: We violate the `Final` annotation of attribute `x` by overwriting it. (stored in file `point_user_wrong.py`; output in Listing 11.5)

```
"""Example of using our class where we change the `Final` attributes."""

from point import Point          # Import our class from its module.

p1: Point = Point(3, 5)          # Create a first instance of Point.
print(f"{p1.x = }, {p1.y = }")   # p1.x = 3, p1.y = 5

p1.x = 5                         # This is not allowed, but possible!
print(f"{p1.x = }, {p1.y = }")   # p1.x = 5, p1.y = 5
```

↓ `python3 point_user_wrong.py` ↓

Listing 11.5: The `stdout` of the program `point_user_wrong.py` given in Listing 11.4.

```
p1.x = 3, p1.y = 5
p1.x = 5, p1.y = 5
```

We now create a second instance, `p2`, of the class `Point`. We this time, just for fun, pass the arguments by parameter name, i.e., write `x=8` and `y=7`. These arguments will then again be passed to `__init__`. This assigns `7` to `p2.x` and `8` to `p2.y`. We can again print the values of these attributes using an f-string.

The `type` of `p2` is again the `class` `point.Point`. Our objects can be used with the `is` operator, which checks for object identity. `p1` is the same object as itself, which means that `p1 is p1` yields `True`. While being instances of the same class, `p1` and `p2` are different objects. Therefore, `p1 is p2` yields `False`.

We can now also use our method `distance`. `p1.distance(p2)`, i.e., the distance from `p1` to `p2`, is the same as `p2.distance(p1)`, i.e., the distance from `p2` to `p1`. Both are equal to 5, because $\sqrt{(7-3)^2+(8-5)^2} = \sqrt{4^2+3^2} = \sqrt{25} = 5$.

`Point` can indeed be used like any other datatype that we discussed before. For example, we can have lists of instances of `Points`. The appropriate type hint for such a list would be `list[Point]`. We can even create such a list with list comprehension, exactly as we learned back in Section 10.2.

We can then process this list by writing a generator expression which transforms the points to strings, just as we discussed in Section 10.6. The expression interpolates the f-string `f"({p.x}, {p.y})"` for each `Point` `p` in our list `point_list`. This means that a sequence of strings of the form `"(x, y)"` is created. These are then concatenated by the `join` method of the string `", "` (see Section 7.4). The result can be seen at the bottom of Listing 11.3.

`Point` is a datatype like any other datatype. Actually, it is the very first datatype in `Python` that we have created. This is quite cool, if you think about it. The programming language has datatypes like `str` or `list`. Now we are now able to extend the programming language with our own datatypes, that make sense in our own scenarios.

The objects of our `Point` class are *immutable*. Once created, the attributes cannot be changed anymore, at least not without violating the language rules. The thing that (should) stop anyone from changing them is the `Final` type hint. As said before, in `Python`, type hints are just hints and not enforced by the interpreter [546]. Thus, the attributes `x` and `y` of instances of `Point` can actually be changed. However, tools like `Mypy` will detect such mistakes and report them [546].

Program `point_user_wrong.py` explores this problem in Listing 11.4. After creating the `Point` object `p1` exactly as in our first example, it frivolously sets `p1.x = 5`. The `Final` type hint tells us explicitly that this should not be done. However, as the output in Listing 11.5 shows, we can do it anyway.[3]

That this is not a good idea becomes already clear when we open our program in `PyCharm`. Figure 11.1 illustrates that `PyCharm` highlights the violating line of code with a yellow mark. Hovering the mouse over this note tells us in no uncertain terms that we are doing something that should be

[3] Notice that we explored this ability back in Section 4.1.3, where we changed the values of `pi`, `nan`, and `inf` from the `math` module. Indeed, these are also annotated with the `Final` type hint and it did not stop us back then either.

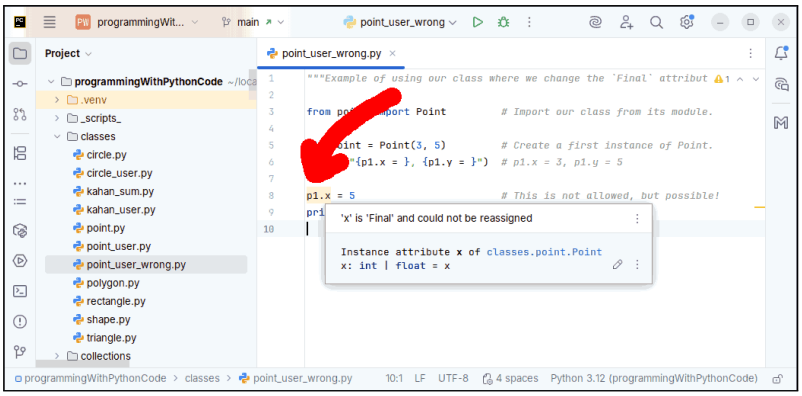


Figure 11.1: We open our program `point_user_wrong.py` from Listing 11.4 that uses the class `Point` incorrectly in `PyCharm`. If we hover the mouse over the yellow mark in the line that violates the `Final` annotation, `PyCharm` informs us that doing that is wrong.

Listing 11.6: The result of applying `Mypy` to the program `point_user_wrong.py` from Listing 11.4 that violates the `Final` annotation of the attributes of our class `Point`. `Mypy` reports the issue.

```
$ mypy point_user_wrong.py --no-strict-optional --check-untyped-defs
point_user_wrong.py:8: error: Cannot assign to final attribute "x"  [misc]
Found 1 error in 1 file (checked 1 source file)
# mypy 2.4.1 failed with exit code 1.
```

forbidden. `Mypy` gives us a similar warning in Listing 11.6.

The attributes `x` and `y` are initialized by `__init__`. There, they are also marked with the type hint `Final` and thus changing them is an error. But why did we do that? Why did we start this topic so early when delving into classes? Because in many cases, making objects immutable is a good design pattern.

> Classes should be immutable unless there's a very good reason to make them mutable.... If a class cannot be made immutable, limit its mutability as much as possible.
>
> — *Joshua Bloch [61], 2008*

Definition 11.1: Immutable

After their initialization, the attributes of *immutable* objects cannot be modified anymore.

Creating classes whose instances are immutable has several benefits, e.g.: Code gets easier to understand, because we do not need to think about whether, when, or how an object changes (because it cannot). The elements of sets and the keys of dictionaries in `Python`, for example, must be immutable objects because these collection store objects based on their hash codes which, in turn, are based on the attributes of these objects. If the attributes would change, so would the hash codes, which means that the objects could no longer be found. Immutable objects are especially useful in parallel programming, where mutable shared state could lead to complex bugs and race conditions.

Still, there also are many cases where we want to create objects whose attributes can change. We will explore one of them in the next section.

But before we do that, let's summarize what we have learned and practically played with so far: By defining our own classes, we can create our own data structures. We can use classes just like any other datatype.

Classes can have their own variables, which are called attributes. They also can have associated functions, which are called methods. The methods access the attributes to compute stuff. Each class can have arbitrarily many attributes and methods.

A special method is the initializer `__init__`. It is called automatically when we create a new instance of the class. Here, all the attributes of a class instance created and receive their initial value.

Of course, we use type hints and docstrings as well as doctests also with classes. For example, ee can mark attributes as immutable with the `Final` type hint. Sadly, `Python` does not strictly enforce that, i.e., we can technically still change `Final` attributes. This, however, is a sin that luckily is detected by tools like `Mypy` and even IDEs like `PyCharm`. Marking attributes as immutable is still a good idea, because it prevents many possible bugs.

11.2 Design Principle: Encapsulation

We often want our objects to be immutable. This obviously cannot always be the case. There also are many situations, where we need objects whose state can change. Then the question is *How should the state of an object be changed?* The vast majority of our classes will have more than one instance attributes. All the attributes of an object together make up its state and form a semantic unit.

If an object has multiple attributes, these attributes may be inter-related and their values depent on each other. We would not want that somebody can change the attributes willy-nilly. Often we want to design mutable objects such that the values of their mutable attributes can only be accessed via methods. The methods and attributes of a class from a semantic unit, too. The methods are written by the same programmer(s) who designed the attributes. These programmers know exactly how the attributes can be changed in a consistent and meaningful way.

We now explore exactly such a secario. We have a class whose attributes are tightly inter-related, such that it would not make any sense to allow a user to change them separately. Instead, we implement methods that can change the attributes in a consistent way and query information from the object. Of course, being the math nerds we are, we will do this on the example of a mathematical algorithm.

We already implemented some interesting mathematical algorithms. We implemented the method of LIU Hui (刘徽) for approximating π in Section 4.1.2, Heron's Method for approximating the square root in Section 7.6, and Euclid's algorithm for the greatest common divisor in Section 8.1. All of these algorithms you likely have seen in high school. Let's now implement an algorithm that you probably

Listing 11.7: Computations with `float` that hit the digit limit, which explored back in Section 3.3.1.

```
Python 3.14.10 (main, Oct 27 2026, 09:35:01) [GCC 15.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
# Python's float supports 15 to 16 digits of precision.
>>> 1e15 + 1          # This works and will give us 1_000_000_000_000_001
1000000000000001.0
>>> 1e16 + 1          # This gives us 1e16, since we would need 17 digits
1e+16

# While 1e16 + 1 cannot be represented exactly with a `float`, one
# may wonder what happens if the computation produces results that
# can be represented exactly ... but has intermediate steps that
# would exceed the range that can be covered by a `float`.
>>> 1e16 + 1 - 1e16  # Since 1e16 + 1 == 1e16, this does yield 0.
0.0

# The problems with the range of `float` can also occur if we have
# two very large numbers ... as long as one is much larger than the
# other.
>>> 1e18 + 1e36       # 1_000_000_000_000_000_001 * 1e18 needs 19 digits
1e+36

# We here can again observe the problem with intermediate results.
# The final result may fit well into a `float`, but if precision is
# lost in intermediate steps, the final result can be off quite a
# bit.
>>> 1e18 + 1e36 - 1e36  # Since 1e18 + 1e36 == 1e36, this gives us 0.
0.0
>>> 1e18 + 1 + 1e36 - 1e36 - 1e18  # The exact result would be 1, but...
-1e+18
```

never heard about: A practical method that tries to mitigate the limitations of the datatype `float`, that we discussed way back in Section 3.3.

We learned that the datatype `float` can represent numbers accurately to about 15 to 16 digits. This is illustrated in Listing 11.7: If we add `1` to 10^{15} = `1e15`, the correct result is `1000000000000001.0`. As you can count, there are 16 digits before the decimal dot, two `1`s and 14 `0`s. Consequenly, adding `1` to 10^{16} = `1e16` would require 17 digits. This exceeds the capacity of our `floats`. Hence, the least-significant digit just "falls off." The result of `1e16 + 1` computed mit `floats` is still `1e16`. It is not possible to represent the number $1+10^{16}$ exactly with the 64 bit double precision floating point numbers that `Python` offers [204, 267, 282].

Usually, that itself is totally fine: There are only very few application were we really need more than 15 digits of precision. Still, let's continue this example for a bit. What happens if we try to compute $10^{16}+1$ and then subtract 10^{16} from this sum? Obviously, in an ideal world, the result should be `1.0`. While `1e16 + 1` would be `10_000_000_000_000_001.0` which does not fit in a `float`, `1.0` is a number that we can correctly and accurately represent as a `float`. The actual result of this computation in `Python`, however, is `0.0`. The reason is that the result of the intermediate computation of `1e16 + 1 == 1e16` and then `1e16 - 1e16 == 0`.

Similarly, computing `1e18 + 1 + 1e36 - 1e36 - 1e18` yields `-1e18`, while the "correct" result would again be `1.0`. The reason is that first, `1e18 + 1 == 1e18` is computed. Then `1e18 + 1e36` yields `1e36`, from which we subtract `1e36` and get `0.0`. The final subtraction of `1e18` then results in `1e-18`. If we had infinite precision, the first step would instead yield $10^{18}+1 = 1\,000\,000\,000\,000\,000\,001$ Adding `1e36` to this number should then yield $1000\,000\,000\,000\,000\,001\,000\,000\,000\,000\,000\,001$. Subtracting `1e36` from this humongous number would bring us back to $10^{18}+1$ and the final subtraction of `1e18` would give us `1.0`. It is easy to see why this does not work out. The designers of the floating point number arithmetic unit in CPUs had to choose some limited number of bits per `float`. They concluded that 52 bits of significand giving us $52/\log_2 10 \approx 15.7$ digits are reasonable and neatly fit into a 8 bytes of memory. On the other hand, 36 digits, requiring a significand of $36\log_2 10 \approx 120$ bits, would make the datatype much bigger and are probably never actually needed. Well, never, unless you try to add up large and small numbers.

The interesting question that we may ask after reviewing the example is: *Is there a way to more accurately add large and small numbers?* Of course, we can never represent `1e16 + 1` exactly as `float`. However, we want to be able to represent the final result of a sum at least as accurately as possible. For example, we want a way to compute the sum `(1e18 + 1)+ -1e18` such that the result is `1.0`. Is this possible?

Luckily, in the 1960s, Kahan [310] and Babuška [22] had an idea how to extend the precision of additions [227, 334]: We add numbers up in a `sum` variable and carry with us a variable `cs` where we remember how far off this sum is. Imagine again that we want to compute `1e18 + 1 - 1e18`. How could we do that to arrive at the result `1.0`? By following this simple algorithm that maintains the overall sum of the numbers in a variable `sum` and keeps track of an error term in a variable `cs`:

1. We start with `sum = 0` and `cs = 0`.
2. For each number to add to the sum, we perform the following steps:
   a) We first compute `t = sum + value`. `t` thus is the sum as precisely as a `float` can represent it.
   b) We then calculate `error = (sum - t)+ value`. Since `t = sum + value`, we could assume that `error` should be `0.0`, because it looks as if it effectively is `(sum - (sum + value))+ value`. However, some digits could have been "lost" when computing `sum + value`. They will re-appear in `error`.
   c) We then set `sum = t`.
   d) We accumulate the errors in `cs`, i.e., set `cs += error`.
3. The final result of the summation is then `sum + cs`.

In the example `1e18 + 1 - 1e18`, we have three numbers that we want to sum up. Let's follow through with this simple algorithm above.

1. We start with `sum = 0` and `cs = 0`.

2. The first number that we want to add to the sum is `1e18`.
   a) In the first step, we have to compute `t = sum + value`, i.e., `t = 0 + 1e18`, meaning that `t = 1e18`.
   b) Then, `error = (sum - t)+ value` becomes `error = (0 - 1e18)+ 1e18`, which gives us `error = 0.0`.
   c) We set `sum = t`, also `sum = 1e18`.
   d) `cs`, which was `0`, becomes `0 + 0.0`, which means that it now is `cs = 0.0`.
3. The second number that we want to add to the sum is `1`.
   a) We compute `t = sum + value`. Now this mean `t = 1e18 + 1` and this results in `t = 1e18`. The `1` is lost.
   b) Well, almost: `error = (sum - t)+ value` becomes `error = (1e18 - 1e18)+ 1`, i.e., `error = 1.0`.
   c) After this step, `sum = 1e18`.
   d) And `cs = 0 + 1.0` becomes `1.0`.
4. We now subtract `1e18` from the sum, which is equivalent to adding `-1e18`.
   a) This means that we first compute `t = sum + value`, i.e., `t = 1e18 + -1e18`, which leads us to `t = 0.0`.
   b) For the error term, we get `error = (1e18 - 0.0)+ -1e18`, which leads to `error = 0.0`.
   c) We now got `sum = 0.0`.
   d) And `cs = 1.0 + 0.0`, which still means that `cs = 1.0`.
5. The final result will be `sum + cs` and this indeed gives us `0.0 + 1.0`, i.e., `1.0`!

Although we cannot represent the intermediate results of the addition exactly, we still got the right result. With the Kahan-Sum, we can compute the correct result of `1e18 + 1 - 1e18`, namely `1.0`. The trick was to keep track of the total error in the variable `cs`. The price thus is that we needed additional variables and needed to update them in each step.

There exists a variety of different implementations of this so-called Kahan-Sum or Kahan-Babuška-Sum. Neumaier, for example, came up with a tweak to improve the precision in 1974 in the case that the running sum is smaller than the numbers we add [409]. Klein [316] then improved the precision further by, basically, maintaining an error sum for the error sum, i.e., created a second degree error sum (and more). This more advanced and accurate algorithm is given in Algorithm 1.

It is not really necessary to discuss the details of the algorithm. It basically is a more advanced version of the simple Kahan-Sum we already tried out above. I intentionally choose this more advanced and scary looking variant of the algorithm. Because I want to make an important point here: Yes, sometimes algorithms look complicated and scary. But if we follow the definition properly, we still can implement them. Even if we may not understand everything perfectly at the beginning.

Algorithm 1: The second-order Kahan-Babuška-Neumaier summation algorithm [22, 310, 316, 409] over an array x of n numbers as defined by Klein in [316].

$sum \leftarrow 0$; $cs \leftarrow 0$; $ccs \leftarrow 0$; ▷ *Initialze sum, 1st order- and 2nd order error.*
for $i \in 0..n-1$ **do** ▷ *For each of the n numbers to add.*
  $t \leftarrow sum + x[i]$; ▷ *Compute the sum and (below) the first-order error term [22, 310].*
  if $|sum| \geq |x[i]|$ **then** $c \leftarrow (sum - t) + x[i]$; ▷ *the tweak by Neumaier [409]*
  else $c \leftarrow (x[i] - t) + sum$;
  $sum \leftarrow t$; ▷ *The overall sum is updated.*
  $t \leftarrow cs + c$; ▷ *The second-order error summation by Klein [316] begins.*
  if $|cs| \geq |c|$ **then** $cc \leftarrow (cs - t) + c$; ▷ *the tweak by Neumaier [409] again*
  else $c \leftarrow (c - t) + cs$;
  $cs \leftarrow t$; $ccs \leftarrow ccs + cc$;
  return $sum + cs + ccs$

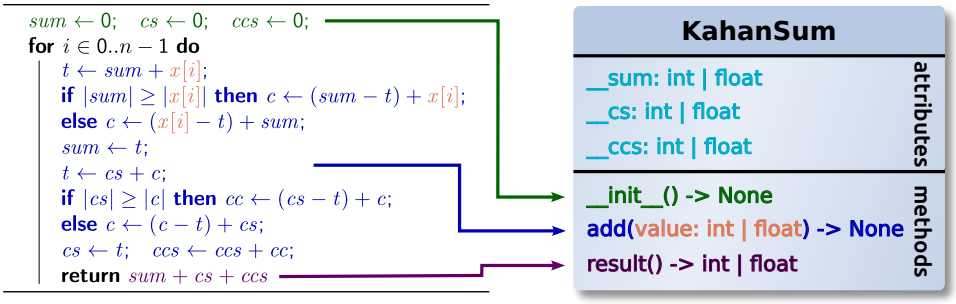


Figure 11.2: A transformation of the Kahan-Babuška-Neumaier summation algorithm specified in Algorithm 1 to a Python class design. All state variables of the algorithm become attributes of the class. The initialization step setting the variables to their starting values is placed into the initializer `__init__`. The part of the algorithm body that adds numbers to the sum becomes method `add`. The final step returning the total sum can be performed at any time by querying method `result`.

While implementing the algorithm, while writing down its components and mapping them to primitives of the programming language, we may be able to understand it better. Actually, we can even use our implementation to understand things better. We could let our code print intermediate results for an example chosen by us. Then we would be able to trace its steps and observe its behavior. Or we could even step over our code using the debugger (see later in Section 13.4). If we have a clear algorithm specification, we can translate it to Python. And we can learn something from that.

And we are now going to implement this algorithm in file `kahan_sum.py` given as Listing 11.8. The question is how should we implement this algorithm? We could, for example, implement it as a function, maybe taking an `Iterable` `x` with the numbers to add up as parameter. It is defined like that in Algorithm 1, after all.

However, we decide to do something else: We want to implement it as a class `KahanSum` that supports the creating of running sums. It should be possible to iteratively add values to the sum and to query the current total. Such a design is presented in Figure 11.2.

A class has attributes and methods, right? To figure out how we can map Algorithm 1 to that, let's look at it more closely. It begins by initializing the state variables sum, cs, and ccs. These state variables should clearly be attributes of the objects. In this case, their initialization would happen in the initializer `__init__`. This is where we will put the code corresponding to this very first setup step of the algorithm. And it is fairly easy to do that.

The loop in Algorithm 1 adds one number $x[i]$ to the internal running sum at a time. We want to implement a method `add` that does exactly that. It is supposed to add one new number to the internal sum and to update the first- and second-order error terms. This method could then, too, be called in a loop. The parameter `value` of that method then takes the place of the loop variable $x[i]$. We can now add new numbers to this sum by invoking `add`.

After the end of the loop, at the very end of the algorithm, the final result is computed by adding the sum variable sum to the first- and second-order error terms cs and ccs. We want to put the code computing the final result, the error-corrected sum, into a method `result`. Actually, when we look more closely at the algorithm, we realize that there is no reason why the result should be *final*. No state variable is changed during its computation. If we wanted, we could query `result` and then continue adding numbers and then query `result` again. This makes this a *running* sum.

What we did not yet discuss are the attributes of this class. As you can see, we should have attributes for the total sum sum, the first-order error term cs, and the second-order error term ccs. The values of these attributes clearly change during summation, so they won't be `Final`. They represent the *internal* state of our sum. They are meaningless for any outside code. Most likely, nobody but us, who write this code, can understand their meaning.

Therefore, nobody should be able to see their values or even change them. Because there is no reason for that. The attributes of our class should be *"hidden"*. Different from the attributes `x` and `y` of our `Point` class, the user (another programmer) has no business accessing the (internal) state of our sum. Instead, the user should interact with our objects only via the methods `add` and `result`.

Listing 11.8: An implementation of the second-order Kahan-Babuška-Neumaier summation algorithm given in Algorithm 1. (src)

```
"""
The second-order Kahan-Babuška-Neumaier-Summation by Klein.

[1] A. Klein. A Generalized Kahan-Babuška-Summation-Algorithm.
    Computing 76:279-293. 2006. doi:10.1007/s00607-005-0139-x

>>> kahan_sum = KahanSum()
>>> for xi in [1e18, 1, 1e36, -1e36, -1e18]:
...     kahan_sum.add(xi)
>>> kahan_sum.result()
1.0
"""


class KahanSum:
    """The second-order Kahan-Babuška-Neumaier sum by Klein."""

    def __init__(self) -> None:
        """Create the summation object."""
        #: the running sum, an internal variable invisible from outside
        self.__sum: float | int = 0
        #: the first correction term, another internal variable
        self.__cs: float | int = 0
        #: the second correction term, another internal variable
        self.__ccs: float | int = 0

    def add(self, value: int | float) -> None:
        """
        Add a value to the sum.

        :param value: the value to add
        """
        s: int | float = self.__sum  # Get the current running sum.
        t: int | float = s + value   # Compute the new sum value.
        c: int | float = (((s - t) + value) if abs(s) >= abs(value)
                          else ((value - t) + s))  # The Neumaier tweak.
        self.__sum = t  # Store the new sum value.
        cs: int | float = self.__cs  # the current 1st-order correction
        t = cs + c  # Compute the new first-order correction term.
        cc: int | float = (((cs - t) + c) if abs(cs) >= abs(c)
                           else ((c - t) + cs))  # 2nd Neumaier tweak.
        self.__cs = t  # Store the updated first-order correction term.
        self.__ccs += cc  # Update the second-order correction.

    def result(self) -> int | float:
        """
        Get the current result of the summation.

        :return: the current result of the summation
        """
        return self.__sum + self.__cs + self.__ccs
```

Listing 11.9: The output of `pytest` executing the doctests for our `KahanSum` class in module `kahan_sum.py`: The test succeds without error.

```
$ pytest --timeout=10 --no-header --tb=short --doctest-modules kahan_sum.py
========================== test session starts ==========================
collected 1 item
kahan_sum.py .                                                    [100%]
=========================== 1 passed in 0.01s ===========================
# pytest 9.2.0 with pytest-timeout 2.4.0 succeeded with exit code 0.
```

Listing 11.10: Some use cases of our Kahan-summation class from Listing 11.8 and a comparison with Python's `sum` and the function `fsum` from the `math` module. (stored in file `kahan_user.py`; output in Listing 11.11)

```
"""Examples for using our class :class:`KahanSum`."""

from math import fsum  # An (even more) precise summation algorithm.

from kahan_sum import KahanSum  # Import our new own class.

# Iterate over four example arrays.
for numbers in [[1e-15, 1e-14, 1e-13, 1e-16, 1e-12], [1e18, 1, -1e18],
                [ 1e36, 1e18, 1, -1e36, -1e18],
                [1e36, 1e72, 1e18, -1e36, -1e72, 1, -1e18],
                [1, -1e-16, 1e-16, 1e-16]]:
    print(f"======= numbers = [{', '.join(map(str, numbers))}] =======")
    k: KahanSum = KahanSum()  # Create our Kahan summation object.
    for n in numbers:         # Iterate over the numbers...
        k.add(n)              # ...and let our object add them up.
    print(f"sum(numbers)  = {sum(numbers)}")    # the normal sum
    print(f"Kahan sum     = {k.result()}")      # our better result
    print(f"fsum(numbers) = {fsum(numbers)}")   # the exact result
```

↓ `python3 kahan_user.py` ↓

Listing 11.11: The stdout of the program `kahan_user.py` given in Listing 11.10.

```
======= numbers = [1e-15, 1e-14, 1e-13, 1e-16, 1e-12] =======
sum(numbers)  = 1.1111e-12
Kahan sum     = 1.1111e-12
fsum(numbers) = 1.1111e-12
======= numbers = [1e+18, 1, -1e+18] =======
sum(numbers)  = 1.0
Kahan sum     = 1.0
fsum(numbers) = 1.0
======= numbers = [1e+36, 1e+18, 1, -1e+36, -1e+18] =======
sum(numbers)  = 0.0
Kahan sum     = 1.0
fsum(numbers) = 1.0
======= numbers = [1e+36, 1e+72, 1e+18, -1e+36, -1e+72, 1, -1e+18] =======
sum(numbers)  = -1e+18
Kahan sum     = 0.0
fsum(numbers) = 1.0
======= numbers = [1, -1e-16, 1e-16, 1e-16] =======
sum(numbers)  = 1.0
Kahan sum     = 1.0
fsum(numbers) = 1.0
```

Definition 11.2: Encapsulation

Encapsulation means that all the attributes of an object can only be accessed its methods. Under complete encapsulation, it is impossible to read or to modify the attributes of an object in any way other than using the methods of an object. Encapsulation therefore allows the programmer to ensure that the state of an object can only be modified in a consistent and correct way.

We want to encapsulate our objects completely. We create the three attributes `__sum`, `__cs`, and `__ccs`. These names are directly taken from the variable names in Algorithm 1, but notice the double leading underscores in front of the names.

Best Practice 65

Names of attributes and methods that start with a double leading underscore (`__`) are to be treated as *private* [457, 458]. They should not be accessed or modified from outside the class. All internal attributes and methods of a class that should not be exposed to the outside therefore should be named following this convention (with two leading underscores). Python applies some internal name mangling to make such attributes harder to access from outside the class[594]

In other words, nobody should access our attributes `__sum`, `__cs`, or `__ccs` from the outside. Everybody should only use our methods to access the state of our `KahanSums`. Sadly, like all such things, Python does *not* enforce this... ...This naming convention is thus not an absolute protection. It is only a very clear hint to other programmers to not touch these attributes.

We could implement the algorithm as a simple function applied to an `Iterable` of numbers. That would have been the most natural way to implement it. However, we chose the class design because it allows for a much greater flexibility. We can, of course, still use instances of our class to sum up sequences. However, we can also use them to sum up numbers in any other way that pleases us. We can query intermediate sums. Or we can use multiple `KahanSums` at the same time. For example, if we have a stream of data, we could use two instances of `KahanSum` to add up the data values (in the first instance) as well as their squares (in the second instance), which may come in handy when computing a sample variance.

With that out of the way, we can begin the actual implementation. A new instance of `KahanSum` will start with all of its three summation attributes initialized to `0`, which is done in the initializer `__init__`. We first implement the `add` method, which corresponds to the body of the loop in Algorithm 1. Looking at the algorithm, in each iteration, a value $x[i]$ is coming in and should be added to the sum. Our `add` method therefore has the parameter `value`, with the value that we want to add to the sum. Then we just have to write down the loop body from Algorithm 1 into `add`. We replace sum, cs, ccs, and $x[i]$ with `self.__sum`, `self.__cs`, `self.__ccs`, and `value`, respectively. We write the conditional for the tweak by Neumaier [409] as an inline `if...else` statement (see Section 6.4). For computing the absolute value $|a|$ of a number a, we can use Python's `abs` function. For the sake of simplicity, we can let And that's already it. There is nothing in the loop body of Algorithm 1 that we cannot write down almost directly as Python code.

We now can place the last line of the algorithm that returns the final result of the summation into the new method `result`. Obviously, it will again access the attributes `self.__sum`, `self.__cs`, and `self.__ccs` in place of sum, cs, and ccs, respectively. But that's it. And we are done. We now have translated a relatively intricate mathematical algorithm into a piece of Python code. We packaged a monolithic algorithm into a API that can iteratively be invoked.

But does it also work? First, we add some doctest to the docstring the module. This doctest computes the sum of `1e18 + 1 + 1e36 - 1e36 - 1e18`, which was our example from back in Listing 11.7. There, we saw that writing the sum down like this will yield `0.0`. We know that the correct result should be `1.0`, though. Using our `KahanSum` class to add the elements `[1e18, 1, 1e36, -1e36, -1e18]`, however, is *supposed* to yield this correct result. As we can see in Listing 11.9, it does exactly that.

In program `kahan_user.py` given as Listing 11.10, we now use our new `KahanSum` class to sum up some numbers. We compare its result with the built-in function `sum` and the exact summa-

tion function `fsum` from the `math` module. We find that all three summation algorithms can correctly add up `[1e-15, 1e-14, 1e-13, 1e-16, 1e-12]` to `1.1111e-12`. `sum` returns `0.0` for the sum of `[1e+18, 1, -1e+18]` as well as for `[1e+36, 1e+18, 1, -1e+36, -1e+18]`. `KahanSum` and `fsum` both correctly compute `1.0` for both cases. If we include even larger numbers and compute the sum over `[1e+36, 1e+72, 1e+18, -1e+36, -1e+72, 1, -1e+18]`, our `KahanSum` returns `0.0` instead of the correct result `1.0`. `fsum` indeed computes the correct result, whereas `sum` is way off and returns `-1e18`. Finally, we add up `[1, -1e-16, 1e-16, 1e-16]`. All three methods return `1.0`, while the exact result would be $1+10^{-16}$. This number, however, cannot be represented with the datatype `float`. Therefore, the results are right.

We find that `fsum` gives us the most precise result. Our `KahanSum` object is better than the built-in `sum`. This leads to two questions:

1. Why can our `KahanSum` not always give us the exact results?
2. How is `fsum` better?

The answer to the first question is fairly simple: If you use only a single summation variable, as `sum` does, then you can be precise to about 15 to 16 digits and lose all decimals beyond that. If you have one summation variable and a first-order error term `cs`, then you basically gain about another 15 or 16 digits for representing intermediate results more precisely. With the second-order error term `ccs`, we can mantain an intermediate result with 45 to 48 digits. This allows us to add `1` to `1e36` and then subtract `1e36` again to get `1`. But if we try to add `1e72`, then we will clearly exceed this 48 digit window and the `1` gets lost. So with our `KahanSum` with two error terms, we can have intermediate results of maybe up to 48 digits.

The second question is *What does `fsum` do differently?* Actually, not very much. It is based on the algorithm by Shewchuk [259, 507], which dynamically allocates a list of helper variables to obtain a provably exact sum (within the result range of `float`). It basically is a dynamic version of the Kahan Sum, which uses more error term variables as needed and also drops them when they are no longer needed. The general principle is the same, just that a list of values replaces `cs` and `ccs`.

Our `KahanSum` uses exactly three variables for the whole summation process. It does not dynamically allocate more memory. As the result, the number of steps it performs are constant for each addition. The algorithm by Shewchuk implemented as `fsum` performs a variable number of steps that depends on the current length of its internal data structure. Thus, `KahanSum` is indeed a very nice compromise solution, which offers higher precision than normal summing but retains the same constant memory and time complexity.

Another advantage of our `KahanSum` over the function `fsum` is that we can use it in a more versatile fashion. We do not need to have all values ready in an `Iterable`. If we had a generator of a sequence of values and wanted to compute the sum of the values themselves as well as the sum of their squares, we could simply use two instances `KahanSum` while iterating over the sequence. With `fsum`, we would need to iterate over the sequence twice. This may not be practical if the sequence is generated from some outside source and was too long to just store it in memory.

11.3 Summary

Classes allow us to solve two problems in programming.

1. Wan semantically group data and the operations on the data together.
2. We can define interfaces, i.e., APIs, that consist of multiple operations. We can then create different realizations of the iterface that implement these operations in different ways.

So far, we focussed on the first concept. For this, we have seen two examples in this section.

We also learned about two very fundamental principles of how classes can be designed.

1. We can created immutable classes as containers for fixed information and whose attributes that does not change. Our `Point` class is an example of that.
2. We can also created encapsulated classes, which are classes where all access to attributes is only possible via methods of the class. This ensures that all state change is consistent and correct. The `KahanSum` class is an example of that.

Most classes that either actually store data or realize some behavior can be implemented in one of these two styles. Of course, there can also by hybrid designs, e.g., classes where some attributes are `Final` and can never change and thus are made readable for outside code without the detour over methods, while some other attributes can only be accessed and changed via methods.

Let us revisit the two example scenarios that we discussed so far. Instances of our `Point` class store a pair of coordinates in the two-dimensional Euclidean plane. The operation `distance` is inseparably linked to this data structure. When developing it, we also learned that the first design principle mentioned above, mking objects *immutable*, is a good idea. Here, the values of the attributes of an object are set only during its initialization (in the `__init__` method) and never change. Then there can never be any confusion about the values of the attributes. It cannot happen that one part of our process has a reference to a `Point` variable and "thinks" that it's coordinates are $(0, 1)$, but some other code changed the coordinates to something else. This cannot happen precisely because the values of the coordinates in instances of `Point` can never change.

If the attributes need to change, then it is often a good idea to *encapsulate* them. Encapsulation means that the attributes of an object can only be read or changed via methods of the object. Our `KahanSum` class is an example of this. This class allows us to add numbers more accurately by internally keeping track of errors resulting from normal summation. The user never gets to see these internal attributes and thus also can never modify them directly. Instead, they are changed by passing new numbers to the `add` method. Via the method `result`, the user can get a consistent view of the state of the summation without getting confused about its internal state.

Now, Python is a very lenient language. It is very hands-off in terms what is permitted and what not. For example, you know that type hints are only hints for tools and programmers. The Pythoninterpreter basically does not care about them. It is possible to do something like `a: str = 5`. This leniency also concerns both design principles above.

Attributes and variables are made "immutable" by annotating them with the type hint `Final`. Since this, again, is only a type hint, it is just information for the programmer and for tools. The Python interpreter ignores it. We can do `a: Final[int] = 5` and then `a = 6` without punishment. Similarly, marking attributes as private by using names starting with a double underscore (`__`) does not actually make them private. The Python interpreter applies some name mangling [594] to make them harder to access, but a crafty programmer can still read and write them. Tools like Mypy, however, will also spot such misconduct.

Python places the responsibility into our hands to properly adher to standards and coding rules. But it does not enforce such behavior.

In the next section, we will see how inheritance of classes in Python can be used to implement APIs, which is the second big use-case of classes.

Chapter 12

Inheritance

Inheritance or subclassing is one of the most important concepts of Object-Oriented Programming (OOP). A subclass can be derived from one existing class. It will inherit all the methods and attributes of that class. It can add new methods and attributes. It also can overwrite methods, i.e., replace them with other code. This mechanism of inheritance can be viewed from several different angles.

For example, we could look at this from the perspective of set theory. We can consider a class as a set of all objects that belong to that class. Subclasses are then subsets of these sets. Remember the tree illustration Figure 9.1 of built-in `Exception`-types back in Section 9.3. The root was formed by `BaseException`. All exception objects instances of this class. It has a subclass `Exception`. All instances of `Exception` are `BaseExceptions`, but not all `BaseExceptions` are `Exceptions`. `ArithmeticError` is a subclass of `Exception`, meaning that all `ArithmeticErrors` are `Exceptions` (but not vice versa). In this sense, one could say that if a class `ClassA` is a subclass of a class `ClassB`, i.e., if `issubclass(ClassA, ClassB)`, then `ClassA` is also a subset of `ClassB`.

$$\texttt{ClassA} \subseteq \texttt{ClassB} \Leftrightarrow \texttt{issubclass(ClassA, ClassB)} \tag{12.1}$$

Another perspective is the concept of specialization. In a reasonable design, subclasses are not just some random or unstructured subsets of their base classes. Usually, but they describe specific features that all of their instances have and the others not. If `Animal` was a class, then `Bird` would be a subclass. But it would not just be a random group of animals. It would represent the animals that add special traits such as two feet, wings, feathers, and laying eggs to their animal character. It could inherit the attribute `weight` and the method `eat()` from its base class `Animal`. It could have the methods `walk()` and `fly()`. `Sparrow` and `Ostrich` would be special cases, i.e., subclasses, of `Bird`, each having peculiar special traits distinguishing from other `Birds`. `Ostrich` could overwrite `fly()` to raise a `NotImplementedError`, for example.

The concepts of inheritance and overwriting of methods makes classes particularly suitable to define and implement APIs. They allow us to create a base class with methods meant for particular tasks. But we would define the methods as empty hulls, without implementing them. These methods, in the base class, could do nothing or raise a `NotImplementedError`. They would just define the API. Subclasses then can implement the methods with meaningful behavior. Differenz subclasses could implement the API differently. We could, for example, imagine an API for rendering of documents, which could be implemented in one subclass to produce PDF output and in another subclass to produce Scalable Vector Graphics (SVG) output. A user of the API then only needs to understand the documentation and signatures of the methods in the base class. They can then use any subclass, depending on what output they want.

12.1 A Hierarchy of Geometric Objects

We already learned that classes are a proper tool for grouping data and operations on the data together into one semantic unit. Classes offer the concept of inheritance, which basically enables specialization.

Imagine that we would wanted to represent all closed geometrical shapes in a two-dimensional Euclidean plane. The overall structure of this example illustrated in Figure 12.1.

We could begin by creating a base class `Shape`. Each closed shape has an associated *area* as well as a *perimeter*. Therefore `Shape` defines two methods, `area` and `perimeter`, returning the area in area

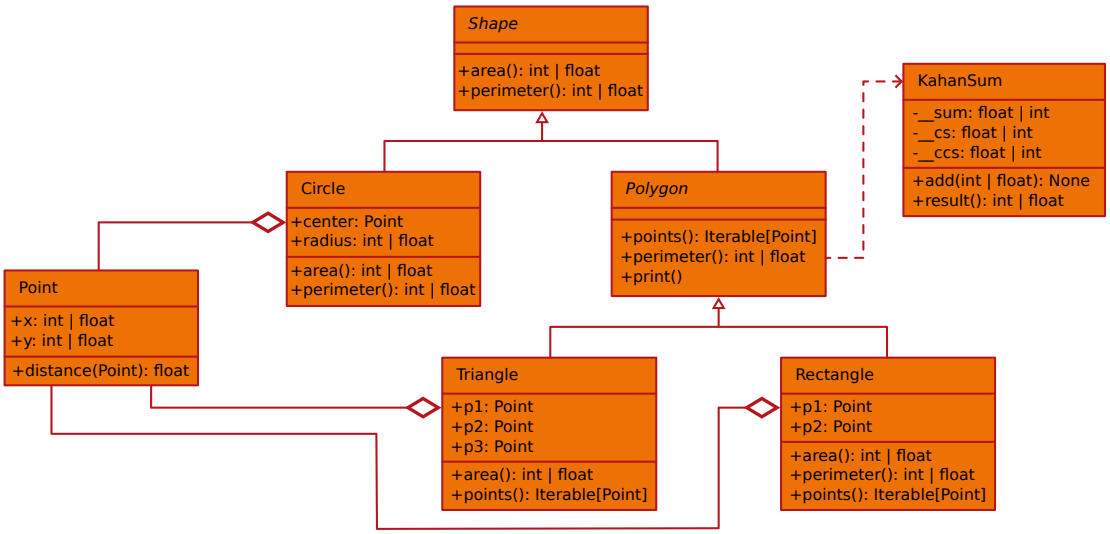


Figure 12.1: An example of class inheritance. The base class `Shape` offers two abstract methods, `area` and `perimeter`. The class `Circle` inherits from `Shape` and implements these methods. It also has an attribute `center`, which is an instance of `Point`, and an attribute `radius`. The class `Polygon` extends `Shape` as well, and offers, amongst others, an `Iterable` of its `Points`. It also uses our `KahanSum` to implement the `perimeter` method. The class `Polygon` is realized by the classes `Triangle` and `Rectangle`.

units and the perimeter length in length units, respectively. It will not implement them yet, though, because how to compute area and perimeter is different for different types of shapes. But by defining the methods, we make clear that these two things can be computed for any actual instance of `Shape`.

In Listing 12.1 with file `shape.py`, we create this base class `Shape`. The class `Shape` is not intended for being instantiated directly. Instead, we just want it to be the base class for the shape types that we learned about in primary school. It does not have any attribute. But, as said, this class has two methods, `area` and `perimeter`. Of course, we cannot really compute the area or the perimeter of such an abstract object. So both methods raise an `NotImplementedError`. If someone would like to actually instantiate `Shape` by doing, say, `s = Shape()` and then invoke `s.perimeter()`, this would fail. This behavior is documented with a doctest in the header of the module. The important thing is that these methods exist. All non-abstract subclasses must overwrite and implement them properly. The user thus can query the area and perimeter of any real instance of `Shape` in exactly the same way.

A circle is a special two-dimensional shape. It does have a center as well as a radius. Knowing these two attributes, we can draw a circle at its correct location and also compute the area and perimeter. If we wanted to express this in terms of classes, we could define the class `Circle` as a subclass of class `Shape`. Not all `Shapes` are `Circles`, but each `Circle` is a `Shape`. The class `Circle` would have two attributes. The attribute `center` could be an instance of our first ever class `Point`. The attribute `radius` could be an `int` or a `float`. The methods `area` and `perimeter` could then be implemented appropriately.

While `Shape` itself is useless, it allows us to create different specialized subclasses that do implement `area` and `perimeter`. The user of these classes could then treat all of these different subclasses in the same way, because all of them support the interface defined by `Shape`.

In file `circle.py` given as Listing 12.2, we define the class `Circle`. By writing `class Circle(Shape)`, we declare it as a subclass of `Shape`. Via its initializer `__init__`, we supply two parameters. The first one is `center`, which is an instance of our class `Point` from Listing 11.1. Der zweite ist `radius`, which can either be an `int` or a `float`. The initializer first checks if `radius` is a finite and positive number. Otherwise, it raises a `ValueError`. The initializer of the class `Point` already ensures that both coordinates of `point` will be finite, so we do not need to check these. We thus ensured that we only create valid circles.

We store `center` and `radius` in two attributes of the same names. We annotate them with the type hint `Final`, which means that they should not be changed after the object is constructed. We thus design our class following the *immutable* approach.

Listing 12.1: A class for representing shapes in the two-dimensional Euclidean plane. (src)

```
"""
A base class for shapes defines a general interface for defining shapes.

It has no attributes, but two methods, `area` and `perimeter`, which
non-abstract subclasses must implement to return the area and perimeter
of the shapes they represent.

>>> from pytest import raises
>>> s = Shape()
>>> with raises(NotImplementedError):
...     s.area()
>>> with raises(NotImplementedError):
...     s.perimeter()
"""


class Shape:
    """A closed geometric shape has an area and a perimeter."""

    def area(self) -> int | float:
        """
        Get the area of this shape.

        :return: the area of this shape
        :raises NotImplementedError: You must overwrite this method.
        """
        raise NotImplementedError  # must be implemented by subclasses

    def perimeter(self) -> int | float:
        """
        Get the perimeter of this shape.

        :return: the perimeter of this shape
        :raises NotImplementedError: You must overwrite this method.
        """
        raise NotImplementedError  # must be implemented by subclasses
```

We can now implement the method `area` to return π`radius`^{2}. We set `perimeter` to return 2π`radius`. With this, the complete interface defined by the superclass `Shape` is now filled with meaning. We also include comprehensive doctests in our file, to immediately make sure that the methods work as advertised.

In program `circle_user.py` given as Listing 12.3, we explore how this new class can be used. We create the instance `circ` of `Circle` by providing the `point=Point(2, 3)` and `radius=5`. We confirm that these parameters are indeed reflected by the corresponding attributes.

The operator `isinstance(a, B)` checks whether an object `a` is an instance of class `B`. If yes, it returns `True` and otherwise, it returns `False`. We confirm that `isinstance(cir, Circle)` is `True`. It also holds that `isinstance(cir, Shape)`. Every instance of `Circle` is also an instance of `Shape`. Because `Circle` is a special case of `Shape`.

It also holds that `isinstance(cir, object)`. `object` is the ultimate base class of any class in `Python` [85, 422]. If you do not provide a base class in the class definition, `object` is used. This is the case for our class `Shape` and also for our older classes `Point` and `KahanSum`. Any `Circle` is there also an `object`.

While the operator `isinstance` relates objects to classes, the operator `issubclass` does the same thing for two classes. `issubclass(A, B)` is `True` if class `A` is a subclass, i.e., inherits from, class `B`. Therefore, `issubclass(Circle, Shape)` is `True` whereas `issubclass(Shape, Circle)` is `False`. `issubclass(Shape, object)` and `issubclass(Circle, object)` are also `True` by definition.

There are, of course, more shapes than just circles. Another very general class of shapes are

polygons. Polygons are shapes enclosed by straight lines. This means that every polygon can be defined by its corner points. If we design a class for polygons, then we can appreciate this fact by defining a method `points()` returning a sequence of these corner points.

In file `polygon.py` presented as Listing 12.5, we define `Polygon` as base class for such shapes. It is a special case of (and thus inherits from) `Shape`. `Polygon` extends the interface of `Shape` by offering the method `points`. This method returns an `Iterable` of instances of `Point`, which are the corner points of the polygon. Our goal here is to provide a base class for different types of `Polygons`. We do not actually want to implement a data structure for arbitrary polygons directly. So the method `points` raises an `NotImplementedError` and thus must be implemented by the subclasses that we will develop later.

So we assume that non-abstract sublclasses of `Polygon` will always implement method `points` appropriately, to return the sequence of corner points of the polygon. If we know the sequence corner points and also know that they are connected by straight lines, then we can easily compute the perimeter of such a shape. We can thus implement the method `perimeter` as follows: We iterate over the instances of `Point` returned by `points()`. In a summation variable, we add up the distance between each point and its successor in the sequence. Finally, we add the distance of the last point to the first point.

The distances can be computed with the `distance` method offered by the `Point` class. While `Polygon` itself does not implement `points`, its subclasses will. The method `perimeter` will then *automatically* use the actual implementation of the subclass. If we call `a.perimeter()`, then the implementation of `points()` of the subclass to which `a` belongs will be used. We therefore can implement `perimeter` here, even if `points` is not yet implemented. If we create a subclass that does implement `points` and call `perimeter` upon an instance of this subclass, it will work.

The summation of the distances could be easily done with a normal variable of type `float`. However, since we implemented the second-order Kahan-Babuška-Neumaier summation algorithm in Listing 11.8, we instead use that one. It should give us a very accurate result. Notice that we could not easily use `fsum` here without first storing all distances in a list. The reason is that we need to add up over the sequence of points and *also* the distance from the last to the first point. So our `KahanSum` does have indeed some advantages.

For convenience sake, we also add a method `print` to `Polygon`, which just prints the sequence of points. So we learned that subclasses can also add new methods and behavior. We also learned that subclasses themselves can be designed to be abstract subclasses. And we learned that if we call `a.method()` (or `self.method()` inside a method), then this will always invoke the "most recent" implementation of `method` in the hierarchy of classes to which object `a` belongs.

Rectangles and triangles are special cases of polygons. We now implement them as classes `Rectangle` and `Triangle` in Listings 12.6 and 12.7, respectively, which both are subclasses of `Polygon`.

The class `Rectangle` for representing rectangles is implemented in file `rectangle.py`. A rectangle can be defined using its bottom-left and top-right corner point. The initializer `__init__` of the class `Rectangle` thus accepts two points `p1` and `p2` as parameters. We raise a `ValueError` if these to points are the same and don't actually form a rectangle. Otherwise, we store the minimum x- and y-coordinate in the bottom-left point attribute `p1`. We store the maximum x- and y-coordinate in the attribute `p2`, marking the top-right corner of our rectangle.

We implement the method `points` to return all four corners of the rectangle. We only needed to store two of them, but we here need to return all four to comply with the definition of the method as given in class `Polygon`. We simply create the points on the fly and return them in a tuple.

The area of the rectangle is easily computed. The method `area` therefore just has to compute the height and width of the rectangle and multiply them with each other. The width is the difference of the x-coordinates of the two corner points. The height is the difference of the y-coordinates of the two corner points.

While we already inherit a perfectly fine method `perimeter` computing the perimeter of our rectangle from `Polygon` based on the result of `points`. But we still override it. This makes sense because we can compute the perimeter faster and more exactly by simply returning twice the sum of the width and height of the rectangle. The inherited `perimeter` method would instead iterate over all four points and compute Euclidean distances using `sqrt`. This is both slower and less accurate, especially if our coordinates would be `ints`.

The class `Triangle` for representing triangles is implemented in file `triangle.py`. This time, we

Listing 12.2: A circle is a special shape, namely the set of all points whose distance from the center of the circle is exactly the radius. (src)

```
"""
A class for circles.

>>> c = Circle(Point(1, 2), 3)
>>> print(f"{c.center.x}, {c.center.y}, {c.radius}")
1, 2, 3
"""

from math import isfinite, pi
from typing import Final

from point import Point
from shape import Shape


class Circle(Shape):
    """A circle is a round shape with a center point and radius."""

    def __init__(self, center: Point, radius: int | float) -> None:
        """
        Create the circle.

        :param center: the center coordinate
        :param radius: the radius

        >>> try:
        ...     Circle(Point(2, 3), -1)
        ... except ValueError as ve:
        ...     print(ve)
        radius=-1 must be finite and >0.
        """
        if not (isfinite(radius) and (radius > 0)):  # sanity check
            raise ValueError(f"radius={radius} must be finite and >0.")
        #: the center point of the circle
        self.center: Final[Point] = center
        #: the radius
        self.radius: Final[int | float] = radius

    def area(self) -> int | float:
        """
        Get the area of this cirlce.

        :return: the area of this cirlce

        >>> Circle(Point(3, 4), 10).area()
        314.1592653589793
        """
        return pi * self.radius ** 2

    def perimeter(self) -> int | float:
        """
        Get the perimeter of this cirlce.

        :return: the perimeter of this cirlce

        >>> Circle(Point(4, 1), 5).perimeter()
        31.41592653589793
        """
        return 2 * pi * self.radius
```

Listing 12.3: An example of using our class `Circle` from Listing 12.2. (stored in file `circle_user.py`; output in Listing 12.4)

```
"""Examples for using our class :class:`Circle`."""

from circle import Circle   # Our new class `Circle`.
from point import Point     # Our very first class ever: `Point`.
from shape import Shape     # Our base class `Shape`.

circ: Circle = Circle(Point(2, 3), 5)  # Create a new circle instance.
print(f"                    center: ({circ.center.x}, {circ.center.y})")
print(f"                    radius: {circ.radius}")
print(f"                 perimeter: {circ.perimeter()}")
print(f"                      area: {circ.area()}")

# Do some instance tests.
print(f"  isinstance(circ, Circle): {isinstance(circ, Circle)}")
print(f"   isinstance(circ, Shape): {isinstance(circ, Shape)}")
print(f"  isinstance(circ, object): {isinstance(circ, object)}")

# Explore the class hierarchy.
print(f" issubclass(Circle, Shape): {issubclass(Circle, Shape)}")
print(f" issubclass(Shape, Circle): {issubclass(Shape, Circle)}")
print(f" issubclass(Shape, object): {issubclass(Shape, object)}")
print(f"issubclass(Circle, object): {issubclass(Circle, object)}")
```

↓ `python3 circle_user.py` ↓

Listing 12.4: The stdout of the program `circle_user.py` given in Listing 12.3.

```
                    center: (2, 3)
                    radius: 5
                 perimeter: 31.41592653589793
                      area: 78.53981633974483
  isinstance(circ, Circle): True
   isinstance(circ, Shape): True
  isinstance(circ, object): True
 issubclass(Circle, Shape): True
 issubclass(Shape, Circle): False
 issubclass(Shape, object): True
issubclass(Circle, object): True
```

Listing 12.5: Polygons are special shapes delimited by straight lines between corner points. (src)

```
"""A polygon is a figure described by its corner points."""

from typing import Iterable

from kahan_sum import KahanSum
from point import Point
from shape import Shape


class Polygon(Shape):
    """Polygons are shapes delimited by straight lines."""

    def points(self) -> Iterable[Point]:
        """
        Get a :class:`Iterable` over the points describing this polygon.

        :return: the points describing the polygon
        :raises NotImplementedError: Must be implemented by subclasses.
        """
        raise NotImplementedError  # must be implemented by subclasses

    def perimeter(self) -> int | float:
        """
        Get the perimeter of this polygon.

        :return: the perimeter of this polygon
        """
        previous: Point | None = None  # the previous point
        first: Point | None = None     # the first point
        total: KahanSum = KahanSum()   # the total perimeter length sum
        for current in self.points():  # Iterate over the points.
            if previous is None:       # We got the first point.
                previous = first = current  # Remember it for last step.
            else:  # We now have previous != None, so we can add length.
                total.add(previous.distance(current))  # Add length.
                previous = current  # Current point becomes previous.
        total.add(previous.distance(first))  # distance back to start
        return total.result()  # Return the perimeter.

    def print(self) -> None:
        """Print the points of this polygon."""
        print(", ".join(f"({p.x}, {p.y})" for p in self.points()))
```

Listing 12.6: Two different points in a plane span a rectangle, which is a special polygon. (src)

```
"""
A class for rectangles.

>>> Rectangle(Point(22, 1), Point(4, 12)).print()
(4, 1), (4, 12), (22, 12), (22, 1)
"""

from typing import Final

from point import Point
from polygon import Polygon


class Rectangle(Polygon):
    """A rectangle defined by its bottom-left and top-right corners."""

    def __init__(self, p1: Point, p2: Point) -> None:
        """
        Create a rectangle.

        :param p1: the first point spanning the rectangle
        :param p2: the second point spanning the rectangle
        """
        if (p1.x == p2.x) or (p1.y == p2.y):  # check for non-emptiness
            raise ValueError(f"{p1.x},{p1.y},{p2.x},{p2.y} is empty.")
        #: the bottom-left point spanning the rectangle
        self.p1: Final[Point] = Point(min(p1.x, p2.x), min(p1.y, p2.y))
        #: the top-right point spanning the rectangle
        self.p2: Final[Point] = Point(max(p1.x, p2.x), max(p1.y, p2.y))

    def area(self) -> int | float:
        """
        Get the area of this rectangle.

        :return: the area of this rectangle

        >>> Rectangle(Point(7, 3), Point(12, 6)).area()
        15
        """
        return (self.p2.x - self.p1.x) * (self.p2.y - self.p1.y)

    def perimeter(self) -> int | float:
        """
        Get the perimeter of this rectangle.

        :return: the perimeter of this rectangle

        >>> Rectangle(Point(10, 5), Point(4, 9)).perimeter()
        20
        """
        return 2 * ((self.p2.x - self.p1.x) + (self.p2.y - self.p1.y))

    def points(self) -> tuple[Point, Point, Point, Point]:
        """
        Get the four corner points of this rectangle.

        :return: a tuple with the four corners of this rectangle
        """
        return (self.p1, Point(self.p1.x, self.p2.y), self.p2,
                Point(self.p2.x, self.p1.y))
```

Listing 12.7: A triangle is a polygon spanned by three points in a plane. (src)

```
"""
A class for triangles.

>>> Triangle(Point(22, 1), Point(4, 12), Point(6, 3)).print()
(22, 1), (4, 12), (6, 3)
>>> Triangle(Point(0, 0), Point(0, 3), Point(4, 0)).perimeter()
12.0
"""

from typing import Final

from point import Point
from polygon import Polygon


class Triangle(Polygon):
    """The class for triangles."""

    def __init__(self, p1: Point, p2: Point, p3: Point) -> None:
        """
        Create a triangle.

        :param p1: the first point spanning the triangle
        :param p2: the second point spanning the triangle
        :param p3: the third point spanning the triangle
        """
        if (p1.distance(p2) <= 0) or (p2.distance(p3) <= 0) or (
                p3.distance(p1) <= 0):  # check for non-emptiness
            raise ValueError("empty triangle")
        #: the first point spanning the triangle
        self.p1: Final[Point] = p1
        #: the second point spanning the triangle
        self.p2: Final[Point] = p2
        #: the third point spanning the triangle
        self.p3: Final[Point] = p3

    def area(self) -> int | float:
        """
        Get the area of this triangle.

        :return: the area of this triangle

        >>> Triangle(Point(-1, 2), Point(2, 3), Point(4, -3)).area()
        10.0
        """
        return 0.5 * abs(self.p1.x * (self.p2.y - self.p3.y)
                         + self.p2.x * (self.p3.y - self.p1.y)
                         + self.p3.x * (self.p1.y - self.p2.y))

    def points(self) -> tuple[Point, Point, Point]:
        """
        Get the three points describing this triangle.

        :return: a tuple with the three corners of this triangle
        """
        return self.p1, self.p2, self.p3
```

Listing 12.8: An example of how all subclasses of class `Shape` can be used in exactly the same way via the methods that are defined in `Shape`. (stored in file shape_user.py; output in Listing 12.9)

```
"""Examples for using the different :class:`Shape` classes."""

from circle import Circle          # Our new class `Circle`.
from point import Point            # Our very first class ever: `Point`.
from rectangle import Rectangle    # Our new class `Rectangle`.
from triangle import Triangle      # Our new class `Triangle`.
from shape import Shape            # Our base class `Shape`.


shapes: list[Shape] = [    # We create list of shapes.
    Circle(Point(2, 3), 5),
    Rectangle(Point(2, 3), Point(3, 5)),
    Triangle(Point(2, 3), Point(3, 5), Point(7, 4)),
]

for s in shapes:  # Print shape classes, areas, and perimeters.
    print(f"{type(s)} instance with A={s.area()} and P={s.perimeter()}")
```

↓ `python3 shape_user.py` ↓

Listing 12.9: The stdout of the program shape_user.py given in Listing 12.8.

```
<class 'circle.Circle'> instance with A=78.53981633974483 and P
    ↪ =31.41592653589793
<class 'rectangle.Rectangle'> instance with A=2 and P=6
<class 'triangle.Triangle'> instance with A=4.5 and P=11.458193116710234
```

need to store all three corner points. Of course, in the initializer `__init__`, we first check if any side has a zero length, which would mean that the points do not describe a valid triangle. We raise an `ValueError` if so.

We return all three points in the `points` method implementation. There is no better way for computing the perimeter than what `Polygon` already provides, so we this time do not override `perimeter`. The area computation is implemented using the formula $A = \frac{1}{2}\left(x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2)\right)$ that you may remember from school maths.

Notice that both subclasses of `Polygon` offer some doctests in the docstrings. While I needed to keep the docstrings short to be able to fit the listings on pages, these doctests still are instructive for the user.

In program shape_user.py illustrated as Listing 12.8, we now use all the classes derived from `Shape`. All of them support the methods `area` and `permeter`.

We declare a variable `shapes` to hold a list of one instance for each of these subclasses. We can annotate the list with the type hint `list[Shape]`. This type hint states that the list can only include instances of type `Shape`. Since all instances of `Circle`, `Rectangle`, or `Triangle` are also instances of class `Shape`, this works OK. We then iterate over the list and print the area and perimeter or each shape.

12.2 Summary

In this section, we have discussed how inheritance with classes in Python works. We used this knowledge to construct a hierarchy of geometric objects in the two-dimensional Euclidean plane. While doing this, we saw how methods can be defined in an abstract way in a base class and then implemented and filled with life in a subclass.

What we have seen here, of course in a very abridged manner, is how complex APIs can be define and implemented. In a way, the class `Shape` defined an interface for geometric objects. This interface defined that each object shall support the two operations `area()` and `perimeter()`. The class `Circle` then implements this API, and so do the classes `Triangle` and `Rectangle`.

OK, that was a bit far-fetched. But in reality, it actually could be a bit like this.

Imagine that you develop an API for creating graphics programmatically. Your objects would basically act as a blank canvas. They would probably provide a method to draw a rectangle with certain coordinates in a certain color. The would probably provide another method to draw a line of a certain width and color connecting two points. You could define this as an abstract base class with (at least) two methods `draw_line` and `draw_rectangle`. We could then go an implement these methods in a subclass in such a way that constructs an SVG graphic [141] in memory. Another subclass could instead construct an Adobe PDF [205, 630] graphic. Doing this would be more complex and exceed the space that we can reasonable use for an example in this book. Yet, the principle of the approach would not be very different from what you have already learned here.

Chapter 13

Dunder Methods

In Python, *everything is an object* [295, 423]. Functions, modules, classes, datatypes, values of simple datatypes, and so on – all are objects. Many of these objects have special functionality. For example, we can add, multiply, and divide numerical objects. We can get string representations for all objects that we can print to the console. We can iterate over the elements of objects that represent sequences. We can execute objects that represent functions. These special functionalities are implemented by so-called *dunder* methods. *Dunder* stands for "double underscore", i.e., `__`, with which all the names of these special methods begin and end. A typical example is the initializer method `__init__`, that creates the attributes of an object.

We already learned that, if we create a subclass of a class, we can define new methods and override existing ones. We can do the same with dunder methods. This means that we can implement, create, change, and customize all of the functionalities listed above!

13.1 `__str__`, `__repr__`, and `__eq__`

In Python, we can distinguish two forms of string representations of a given object `o`:

1. `str(o)` should return a concise and brief representation of the object `o`. This representation is mainly for end users [33]. `str(o)` invokes the `__str__` dunder method of `o`, if it has been implemented. Otherwise, `o.__repr__()` is used instead.

2. `repr(o)` should ideally return a string representation that contains all the information that is needed to re-create the object [32]. The target audience here are programmers who are working on the code, who may need to write precise information into log files, or who are searching for errors. `repr(o)` invokes the `__repr__` dunder method, if it has been implemented. Otherwise, it returns the default representation, which is the type name and ID of the object.

These two functions are compared in program `str_vs_repr.py` given as Listing 13.1. Here, we first create an integer variable `the_int` with value `123`. Both `str(the_int)` and `repr(the_int)` `repr` are `"123"`. This is to be expected, since this is all the information that is needed to completely recreate this value and, at the same time, it is also the most concise way to present the value.

We then create another variable `the_str` with value `"123"`. Printing `the_str` to the `stdout` stream, which means doing `print(str(the_str))`, will make the text `123` appear on the console. Printing `repr(the_str)`, however, produces `'123'`. Notice the added single quotation marks on each side? These are necessary. Without them, `repr(the_str)` and `repr(the_int)` would be the same. We could not distinguish whether the value we printed was a string or an integer. This, of course, matters only if we care about the internal workings of our program. This is the purpose for the existence of `repr`.

Next, we create two collections. First comes the list `l1`, which contains the three integers `1`, `2`, and `3`. Then we create the list `l2`, which contains the three strings `"1"`, `"2"`, and `"3"`. Then we print both lists, which will use `str(l1)` and `str(l2)` internally. The result of `print(f"{l1 = }, but {l2 = }")` is `l1 = [1, 2, 3], but l2 = ['1', '2', '3']`. Notice that the single quotation marks around the string elements of `l2` are printed? When obtaining the string representations of the standard Python collections with either `str` or `repr`, the elements of the collections

Listing 13.1: Comparing the `str` and `repr` representations of integers, strings, lists, and Python's `datetime` class. (stored in file `str_vs_repr.py`; output in Listing 13.2)

```
"""An example comparing `str` and `repr`."""

from datetime import UTC, datetime

the_int: int = 123  # An integer with value 123.
print(the_int)  # This is identical to `print(str(the_int))`
print(repr(the_int))  # Prints the same as above.

the_string: str = "123"  # A string, with value "123".
print(the_string)  # This is identical to `print(str(the_string))`.
print(repr(the_string))  # Notice the added `'` around the string.

l1: list[int] = [1, 2, 3]  # A list of integers.
l2: list[str] = ["1", "2", "3"]  # A list of strings.
print(f"{l1 = }, but {l2 = }")  # str(list) uses repr for list elements.

# Get the date and time when this program was run.
right_now: datetime = datetime.now(tz=UTC)

# Print the human-readable, concise string representation for users who
# want to know that the object means but do not necessarily need to know
# its detailed content.
print(f" {str(right_now) = }")
print(f"     right_now  =  {right_now!s}")

# Print the format for programmers who need to understand the exact
# values of all attributes of `right_now`.
print(f"{repr(right_now) = }")
print(f"     right_now  =  {right_now!r}")
```

↓ `python3 str_vs_repr.py` ↓

Listing 13.2: The stdout of the program `str_vs_repr.py` given in Listing 13.1.

```
123
123
123
'123'
l1 = [1, 2, 3], but l2 = ['1', '2', '3']
 str(right_now) = '2026-11-05 00:14:13.267959+00:00'
     right_now  =  2026-11-05 00:14:13.267959+00:00
repr(right_now) = 'datetime.datetime(2026, 11, 5, 0, 14, 13, 267959, tzinfo
    ↪ =datetime.timezone.utc)'
     right_now  =  datetime.datetime(2026, 11, 5, 0, 14, 13, 267959, tzinfo
        ↪ =datetime.timezone.utc)
```

are converted to strings using `repr`, not `str` [78]. Otherwise, we could not distinguish `l1` and `l2` in the output.

Another good example of the difference between `str` and `repr` is Python's `datetime` class. We will not discuss this class here in any detail. It suffices to know that instances of this class represent a combination of a date and a time. In the program, we first import the class `datetime` from the module of the same name. We create a variable `right_now` and assign to it the result of the function `datetime.now`, which returns an object representing, well, today and the current time.[1]

If we want to print the result of the `str` function applied to an object `o` in an f-string, then we can do this using the format specifier `!s`, i.e., by writing `f"{o!s}"` We find that the simple string

[1] In the output of our program given in Listing 13.2, you cannot see the time of your reading, but the time when this book was compiled.

Listing 13.3: Investigating string representations and equality for the class `Point`. (stored in file point_user_2.py; output in Listing 13.4)

```
"""Examples for using our class :class:`Point` without dunder."""

from point import Point

p1: Point = Point(3, 5)  # Create a first point.
p2: Point = Point(7, 8)  # Create a second, different point.
p3: Point = Point(3, 5)  # Create a third point, which equals the first.

print(f" {str(p1) = }")  # should be a short string representation of p1
print(f"     p1  =  {p1!s}")  # (almost) the same as the above
print(f"{repr(p1) = }")  # should be a representation for programmers
print(f"     p1  =  {p1!r}")  # (almost) the same as the above

print(f"{(p1 is p2) = }")  # False, p1 and p2 are different objects
print(f"{(p1 is p3) = }")  # False, p1 and p3 are different objects
print(f"{(p1 == p2) = }")  # False, because without dunder `==` = `is`
print(f"{(p1 == p3) = }")  # False, but should ideally be True
print(f"{(p1 != p2) = }")  # True, because without dunder `==` = `is`
print(f"{(p1 != p3) = }")  # True, but should ideally be False

print(f"{(p1 == 5) = }")  # comparison with the integer 5 yields False
```

↓ `python3 point_user_2.py` ↓

Listing 13.4: The stdout of the program point_user_2.py given in Listing 13.3.

```
 str(p1) = '<point.Point object at 0x7fa92e350c20>'
     p1  =  <point.Point object at 0x7fa92e350c20>
repr(p1) = '<point.Point object at 0x7fa92e350c20>'
     p1  =  <point.Point object at 0x7fa92e350c20>
(p1 is p2) = False
(p1 is p3) = False
(p1 == p2) = False
(p1 == p3) = False
(p1 != p2) = True
(p1 != p3) = True
(p1 == 5) = False
```

representation of a `datetime` object is, well, a simple human readable date and time string. The result of the function `repr` for an object `o` can be obtained in an f-string by using the format specifier `!r`, i.e., by writing `f"{o!1}"`. Doing this with a `datetime` object gives us all the information that we need to manually recreate the object. We could copy the output of `repr` from Listing 13.2 into the `Python` console! This would create a `datetime` object with exactly the same data as `right_now`. This would also work with the string representations that we printed for our lists `l1` and `l2` above.

Let us now move a bit backwards and revisit a previous example we created by ourselves. In Section 11.1 presenting file point.py, we created the class `Point` for representing points in the two-dimensional Euclidean plane (see Listing 11.1). This class turned out to be quite useful when we went on to implement classes for different two-dimensional geometric shapes. Here, we already implemented one dunder method, the initializer `__init__`. Let us play with this class a bit more.

In Listing 13.3 showing program point_user_2.py, we create three instances of this class. `p1` represents the coordinates $(3, 5)$, `p2` stores $(7, 8)$, and `p3` has the same coordinates as `p1`. In this program, we first print the `str` and `repr` results for `p1`. We immediately find them very unsatisfying. Since we implemented neither `__str__` nor `__repr__`, the default result for `str` falls back to the result of `__repr__` which then falls back to just the type name and object ID. This gives us basically no useful information.

While we are on the subject of "not useful," there is another aspect of our `Point` class that does not show useful behavior. Way back in Section 4.5, we discussed the difference between object identity

and object equality. All three variables `p1`, `p2`, and `p3` point to different objects. While `p1 is p1` is obviously `True`, `p1 is p2` and `p1 is p3` are obviously `False`. The three objects are not all different instances of `Point`, so this is expected.

However, we find it annoying that `p1 == p3` is `False`, too. `p1 == p2` should be (and is) `False`, because the two points are different. But the two points `p1` and `p3` have the same coordinates. They should be considered equal for all intents and purposes. Vice versa, `p1 != p2` should be (and is) `True`, but `p1 != p3` should be `False` but turns out to be `True`.

The reason for this is that Python cannot know when and why instances of our own class should be equal. So it simply assumes that equality = identity, i.e., an object is only equal to itself. We could fix this by implementing the `__eq__` dunder method. This method would receive an arbitrary object `other` as input and should return `True` if that is equal to the object whose method was invoked. Similar to the `str(o)` function, which invokes `o.__str__()` , the `a == b` operator invokes `a.__eq__(b)`, if `a` implements `__eq__`.

If you implement `__eq__`, Python will make the reasonable assumption that `(a != b)== not (a == b)`, i.e., assume that two objects are unequal if and only if they are not equal [589]. However, this is not necessarily always the case[2]. Therefore, Python also allows us to implement an `__ne__` dunder method which is called by `a != b` as `a.__ne__(b)` if it is implemented[589].

Finally, we compare whether `p1` is the same as the integer number `5`. This, obviously, should return `False`. And it does so. This is because the two objects `p1` and `5` are not identical. As said, the default equality comparison only checks for identity. If implement `__eq__` by ourselves, this method should also return a `False` if it receives `5` as argument (and not crash or raise an exception...). Anything else would be nonsense.

In order to fix all of the problems discussed above, we implement the three dunder methods `__str__`, `__repr__`, and `__eq__` for our `Point` class in file `point_with_dunder.py` shown as Listing 13.5. The concise string representation returned by `__str__` will just be the point coordinates separated by commas and in parentheses. This offers all the information needed at a glance, but it could be mistaken with a tuple as string. Therefore, the canonical string representation produced by `__repr__` will return a string of the shape `"Point(x, y)"`.

Finally, the `__eq__` method will first check if the `other` object is an instance of `Point`. If so, it will return `True` if and only if the `x` and `y` coordinate of the `other` point are the same as of the point `self`. Otherwise, it will return the constant `NotImplemented`:

> A special value which should be returned by the binary special methods [...] to indicate that the operation is not implemented with respect to the other type...
>
> *Note:* When a binary (or in-place) method returns `NotImplemented` the interpreter will try the reflected operation on the other type (or some other fallback, depending on the operator). If all attempts return `NotImplemented`, the interpreter will raise an appropriate exception. Incorrectly returning `NotImplemented` will result in a misleading error message or the `NotImplemented` value being returned to Python code.
>
> — *[81], 2001*

By returning `NotImplemented` for `other` objects that are not instances of class `Point`, we simply defer to the default behavior of the `==` operator. In other words, our `__eq__` method can only compare the current `Point` for equality with another `Point`. If `other` is not an instance of `Point`, then no way to compare for equality with it exists. Now, we could return `False` in this case, which would be fine as well. Returning `NotImplemented` will give us the same result in comparisons with objects of other types (like `5`). However, it keeps an avenue open for other programmers to design new classes which support comparison with our `Point` instances in a consistent way. When we implement the `__eq__` method like this, the proper type hint for the return value is `bool | NotImplementedType`.

Program `point_with_dunder_user.py` given as Listing 13.6 is the same as Listing 13.3, but now uses this new variant of our class `Point`. As you can see in Listing 13.7, its output now matches much better to what one would expect. The function `str` now gives us concise and informative output when applied to an instance of `Point`. The `repr` operator gives us a text that we could copy into

[2] In [589], it is stated that IEEE 754 floating point numbers do not satisfy that `==` and `!=` are each other's complements. However, I could not find for an example where this was true in the standard [282], maybe with the exception of signaling `nans`, which does not matter in Python. Maybe it was true for some Python implementations back then, as [608] indicates.

the console and that would then re-create the point. The equality and inequality operations now also behave reasonable and detect whether two points have the same coordinates. They also work in the presence of non-`Point` inputs.

We now have begun our venture into the realm of so-called *dunder* methods. These methods control much of the behavior of the operators and constructs of the `Python` language. With `__str__`, `__repr__`, `__eq__`, and `__ne__` we already touched several methods that we used quite often, albeit indirectly. What other adventures may be waiting for us in these deep bowels of the `Python` machine?

Listing 13.5: Our `Point` class, extended with the `__str__`, `__repr__`, and `__eq__` dunder methods. (src)

```
"""A class for points, with string and equals dunder methods."""

from math import isfinite
from types import NotImplementedType
from typing import Final


class Point:
    """A class for representing a point in the two-dimensional plane."""

    def __init__(self, x: int | float, y: int | float) -> None:
        """
        The constructor: Create a point and set its coordinates.

        :param x: the x-coordinate of the point
        :param y: the y-coordinate of the point
        """
        if not (isfinite(x) and isfinite(y)):
            raise ValueError(f"x={x} and y={y} must both be finite.")
        #: the x-coordinate of the point
        self.x: Final[int | float] = x
        #: the y-coordinate of the point
        self.y: Final[int | float] = y

    def __repr__(self) -> str:
        """
        Get a representation of this object useful for programmers.

        :return: `"Point(x, y)"`

        >>> repr(Point(2, 4))
        'Point(2, 4)'
        """
        return f"Point({self.x}, {self.y})"

    def __str__(self) -> str:
        """
        Get a concise string representation useful for end users.

        :return: `"(x,y)"`

        >>> str(Point(2, 4))
        '(2,4)'
        """
        return f"({self.x},{self.y})"

    def __eq__(self, other) -> bool | NotImplementedType:
        """
        Check whether this point is equal to another object.

        :param other: the other object
        :return: `True` if and only if `other` is also a `Point` and has
            the same coordinates; `NotImplemented` if it is not a point

        >>> Point(1, 2) == Point(2, 3)
        False
        >>> Point(1, 2) == Point(1, 2)
        True
        """
        return (other.x == self.x) and (other.y == self.y) \
            if isinstance(other, Point) else NotImplemented
```

Listing 13.6: The same program exploring string representations and equality as shown in Listing 13.3, but this time using our new `Point` class from Listing 13.5. (stored in file `point_with_dunder_user.py`; output in Listing 13.7)

```
"""Examples for using our class :class:`Point` with dunder methods."""

from point_with_dunder import Point

p1: Point = Point(3, 5)  # Create a first point.
p2: Point = Point(7, 8)  # Create a second, different point.
p3: Point = Point(3, 5)  # Create a third point, which equals the first.

print(f" {str(p1) = }")  # a short string representation of p1
print(f"     p1  =  {p1!s}")  # (almost) the same as the above
print(f"{repr(p1) = }")  # sa representation for programmers
print(f"     p1  =  {p1!r}")  # (almost) the same as the above

print(f"{(p1 is p2) = }")  # False, p1 and p2 are different objects
print(f"{(p1 is p3) = }")  # False, p1 and p3 are different objects
print(f"{(p1 == p2) = }")  # False, calls our `__eq__` method
print(f"{(p1 == p3) = }")  # True, as it should be, because of `__eq__`
print(f"{(p1 != p2) = }")  # True, returns `not __eq__`
print(f"{(p1 != p3) = }")  # False, as it should be

print(f"{(p1 == 5) = }")  # comparison with the integer 5 yields False
```

↓ `python3 point_with_dunder_user.py` ↓

Listing 13.7: The stdout of the program `point_with_dunder_user.py` given in Listing 13.6.

```
 str(p1) = '(3,5)'
     p1  =  (3,5)
repr(p1) = 'Point(3, 5)'
     p1  =  Point(3, 5)
(p1 is p2) = False
(p1 is p3) = False
(p1 == p2) = False
(p1 == p3) = True
(p1 != p2) = True
(p1 != p3) = False
(p1 == 5) = False
```

13.2 Objects in Sets and as Keys in Dictionaries: `__hash__`, `__eq__`

Our `Point` objects are immutable. They just consist of two numerical coordinates. Maybe there could be an application where someone would like to use them as keys for a dictionary. Or maybe somebody would like to construct a set of points. For this to be possible, three things are needed:

1. Instances of `Point` must be immutable. This is true already. I just wanted to say it again.
2. There must be a dunder method `__eq__` that compares an instance of `Points` for equality with other objects. This, we already have done in Listing 13.5.
3. The dunder method `__hash__` must be implemented that returns the hash value of a `Point` in form of an `int`. This one is still missing.

If we can fulfill these criteria, `Points` can be elements of sets or keys in dictionaries. So we only have to implement one more method, namely `__hash__`. For the two dunder methods `__eq__` and `__hash]__`, it must hold that [31]

$$\texttt{a.__eq__(b)} \Rightarrow \texttt{a.__hash__()} = \texttt{b.__hash__()} \tag{13.1}$$

This is equivalent [31, 82] to:

$$\texttt{a == b} \Rightarrow \texttt{hash(a)} = \texttt{hash(b)} \tag{13.2}$$

But let us step back a bit here. What is a hash value? Why is an integer hash value needed? Why does the equality of two objects require them to have the same hash value?

Dictionaries in Python (and Java) internally use tables, where key-value relationships are stored [226, 369]. Sets are basically the same, but only store the keys. The internal tables could be represented as linear lists. Differently from lists, new elements are not added at the end. Instead, they would be more like lists of a fixed length where new elements are placed at specific indices where they can be found again. These hash tables [134, 322, 513] are very fast. They have an element-wise read, search, and update time complexity of $\mathcal{O}(1)$ [9, 276, 413]. An element in a list `l` can be searched in $\mathcal{O}(\texttt{len(l)})$ instead, i.e., finding one paricular element is much slower.

As said, you can imagine that a hash table uses something like a list as backing store. How can we turn a list `l` with search complexity $\mathcal{O}(\texttt{len(l)})$ into a hash table with search complexity $\mathcal{O}(1)$? For now, let's imagine that we want to store only integers and that our list is much larger than the number of elements that we want to store. We use a list of `int | None` values and it is filled initially with only `None`. If we want to store an integer `i`, we would compute the modulo division `i % len(l)` an just place it at that index. If we want to check whether an integer `j` is in the list, we again compute `j % len(l)` and check whether it is equal to the element at that index (using `__eq__`). Both operations, insertion and search, now work in $\mathcal{O}(1)$. Of course, this is a very coarse simplification [226, 369]. There could be two different integers with the same result for `i % len(l)`, for example. Dictionaries and sets need to handle such collisions. They also need to grow if they begin to fill up, and so on. But that's the basic idea.

Since we also want to be able to use objects that are *not* integers as keys, too, we need a way to "translate" these objects to integers. That is what `__hash__` is supposed to do. It does not have to be a bijective mapping, i.e., it only needs to go in the direction of `int` but not back. It does not have to be injective, i.e., some different objects can have the same hash code. However, it should ideally be as injective as possible, i.e., the fewer objects have the same hash code, the better. The dictionary/set implementation is then responsible for translating these integer hash codes into valid indices into its internal table.

So the hash values are needed to find the objects in the dictionaries and sets. We want to know whether the object `a` is `in` the set `s`? Then the set `s` uses `hash(a)` which invokes `a.__hash__()` to get the hash value of `a`.[3] Then `s` translates the hash value to an index. It checks its internal table whether there is an object `b` at that index with `b == a`. If yes, then, well, `a` is in the set `s`, i.e., `a in s` yields `True`. If not, then not. As said, the reality is more complex, because of potentially occurring collisions, but for our excursion here, this very coarse approximation of how this works shall suffice.

Anyway, if we want to add an object to the set `s`: Then again the index is computed via the hash value and if the object is not already there, it is placed there. Dictionaries work the same, but instead

[3] This works the same way in which `repr(a)` would invoke `a.__repr__()` if it is defined.

store key-value pairs, where the hash values of the keys are computed to find the right places in the internal tables. You can read about the details in these very interesting sources here [226, 369].

It is already clear, however, that calling `__hash__` twice for the same object `a` must return the same value. Since this hash value is used to find the place in the table where the object should be, this must never change. This is also why dictionary (and set) keys must be immutable (see Best Practice 27 and 30).

It is also clear that two objects `a` and `b` that are *equal* must also have the same hash value. If two objects are equal, it should be that `a in s` = `b in s`. Otherwise, it could be that `"123" in s` is `True` for the string literal `"123"`, but `False` for `str(123)in s`. That would make no sense at all.

Overall, the following issues need to be considered when making objects "hashable."

> ... The `__hash__()` method should return an integer. The only required property is that objects which compare equal have the same hash value; it is advised to mix together the hash values of the components of the object that also play a part in comparison of objects by packing them into a `tuple` and hashing the tuple.
>
> — *[31], 2001*

Best Practice 66

For implementing `__eq__` and `__hash__`, the following rules hold [31]:

- Only immutable classes are allowed to implement `__hash__`, i.e., only classes where all attributes have the `Final` type hint and are only assigned on the initialize `__init__`.
- The result of `a.__hash__()` must never change (since `a` must never change either).
- If a class does not define `__eq__`, it cannot implement `__hash__` either.
- Instances of a class that implements `__eq__` but not `__hash__` cannot be used as keys in a dictionary or set.
- Only instances of a class that implements both `__eq__` and `__hash__` can be used as keys in dictionaries or sets.
- The results of `__eq__` and `__hash__` must be computed using the exactly same attributes. In other words, the attributes of an object `a` that determine the results of `a.__eq__(...)` must be exactly the same as those determining the results of `a.__hash__()`.
- It is best to compute `a.__hash__()` by simply putting all of these attributes into a `tuple` and then passing this `tuple` to `hash`.
- Two objects that are equal must have the same hash value, i.e., Equations 13.1 and 13.2 must hold.

With that out of the way, we can now make our `Point` class hashable. In Listing 13.8 outlining file `point_with_hash.py`, we modify the `Point` class from Listing 13.5. We retain the implementation of `__eq__`. All we have to do is to add the method `__hash__`.

The only attributes that play a role in our `__eq__` method are the two coordinates of the point, `self.x` and `self.y`. So the result `__hash__` should simply be `hash((self.x, self.y))`. The double-parentheses are because this basically means `t = (self.x, self.y)` and then computing `hash(t)`.

When we were writing this, we suddenly get startled. We permitted the coordinates of our points to be either `ints` or `floats`. As we know, `5.0 == 5` is `True`. Therefore, `Point(5.0, 3).__eq__(Point(5, 3))` is also `True`. But is `hash((5.0, 3))` really the same as `hash((5, 3))`? Or, in simpler terms, does `hash(5.0)== hash(5)` really hold?

If not, then we could violate the contract of `__hash__` and `__eq__` from Best Practice 66 in a very subtle and unexpected way. `Point(5.0, 3)` would be equal to `Point(5, 3)`, but their hash codes would differ. If we have a set `s` of points and store `Point(5.0, 3)` in this set, then the outcome of the query `Point(5, 3)in s` could either be `True` or `False`. If the layout of the internal table – whose size depends on all previous insertions and deletions – is such that the different hash codes of both objects

are transformed to the same index, then the outcome would be `True`. In the much more likely case that they don't, the outcome is `False`.

Thus, if `hash(5.0)!= hash(5)` was true, then our implementation could yield programs that behave unexpectedly differently, in rare situations, which we might not be able to deterministically reproduce. That's the one of the worst kinds of bugs we can imagine. At the same time, we could create sets that contain equal elements multiple times. Which, in turn, would violate the definition of sets. No wonder that Python is rather strict with the rules imposed on `__hash__` and `__eq__`...

Alas, the developers of Python have already solved this:

> Numeric values that compare equal have the same `hash` value (even if they are of different types, as is the case for `1` and `1.0`).
>
> — *[83], 2001*

Therefore, we can indeed implement `__hash__` with a single line of code in Section 13.2. We use our new variant of the `Point` class in Listing 13.9 with program `point_with_hash_user.py`.

We again first create three points `p1 = Point(3, 5)`, `p2 = Point(7, 8)`, and `p3 = Point(3, 5.0)`. `p1 == p2` is `False`, while `p1 == p3` is `True`, despite the fact that the `y`-coordinate of `p1` is an `int` and the one of `p2` is a `float` (but with the same value).

Then we create the set `points` as `{p1, p2, p3}`. It has the size 2. Since `p1 == p3`, only one of these two objects is stored in the set. However, `p1 in points`, `p2 in points`, and `p3 in points` are all `True`. This is because `p1` and `p3` are equal and also have the same hash value.

If we create a new point `p4` with coordinates equal to those of `p2`, then `p4 in points` will also hold. However, a point `p5` whose coordinates are different from those of `p1` and `p2` will not be an element of `points`, i.e., `p5 in points` would be `False`.

Now we can also use the instances of our class `Point` as keys for a dictionary `point_vals`. The same dictionary operations as discussed way back in Section 5.5 can be used without problems. We associate value `"A"` with key `p1` and `"B"` with key `p2`. We then insert another key-value mapping by storing `"C"` under key `Point(7, 9)`. This new key-value pair also appears in the dictionary as expected. When we store the value `"D"` under key `Point(3.0, 5.0)`, then this overwrites the value `"A"` stored under `p1`, because `p1` is `Point(3, 5)`. Querying `point_vals[p1]` now yields `"D"`.

The subject of making objects hashable is not something that we encounter very often. But it is a topic that shows that working with dunder methods requires special care. We should really understand what we are doing. Otherwise, we could create very hard to understand errors. We could implement `__eq__` and `__ne__` by accident such that `x == y` is no longer the opposite of `x != y`. Or we could implement `__eq__` and `__hash__` such that sets are no longer set... Such bugs would be very hard to detect. So we always have to read the documentation thoroughly.

Listing 13.8: Our `Point` class, extended with the `__eq__` and `__hash__` dunder methods. (src)

```
"""A class for points, with equals and hash dunder methods."""

from math import isfinite
from types import NotImplementedType
from typing import Final


class Point:
    """A class for representing a point in the two-dimensional plane."""

    def __init__(self, x: int | float, y: int | float) -> None:
        """
        The constructor: Create a point and set its coordinates.

        :param x: the x-coordinate of the point
        :param y: the y-coordinate of the point
        """
        if not (isfinite(x) and isfinite(y)):
            raise ValueError(f"x={x} and y={y} must both be finite.")
        #: the x-coordinate of the point
        self.x: Final[int | float] = x
        #: the y-coordinate of the point
        self.y: Final[int | float] = y

    def __repr__(self) -> str:
        """
        Get a representation of this object useful for programmers.

        :return: `"Point(x, y)"`
        """
        return f"Point({self.x}, {self.y})"

    def __eq__(self, other) -> bool | NotImplementedType:
        """
        Check whether this point is equal to another object.

        :param other: the other object
        :return: `True` if and only if `other` is also a `Point` and has
            the same coordinates; `NotImplemented` if it is not a point
        """
        return (other.x == self.x) and (other.y == self.y) \
            if isinstance(other, Point) else NotImplemented

    def __hash__(self) -> int:
        """
        Compute the hash of a :class:`Point` based on its coordinates.

        :return: the hash code

        >>> hash(Point(4, 5))
        -1009709641759730766
        >>> hash(Point(4.0, 5))
        -1009709641759730766
        """
        return hash((self.x, self.y))  # hash over the tuple of values
```

Listing 13.9: Using the new `Point` class from Listing 13.8 in sets and dictionaries. (stored in file point_with_hash_user.py; output in Listing 13.10)

```
"""Examples for using our class :class:`Point` with hash."""

from point_with_hash import Point


p1: Point = Point(3, 5)     # Create a first point.
p2: Point = Point(7, 8)     # Create a second, different point.
p3: Point = Point(3, 5.0)   # A third point, which equals the first.

print(f"{(p1 == p2) = }")   # False, since p1 is really != p2
print(f"{(p1 == p3) = }")   # True, since p1 equals p3

points: set[Point] = {p1, p2, p3}  # This set will contain 2 points.
print(f"{points = }")  # The set of two points, because p1 == p2.
print(f"{p1 in points = }")  # True
print(f"{p2 in points = }")  # True
print(f"{p3 in points = }")  # True
print(f"{Point(7.0, 8.0) in points = }")  # True: point is equal to p2
print(f"{Point(3.1, 5) in points = }")    # False: point is not in set

# A dictionary with points as keys.
point_vals: dict[Point, str] = {p1: "A", p2: "B"}
print(f"{point_vals = }")       #  {Point(3, 5): 'A', Point(7, 8): 'B'}
point_vals[Point(7, 9)] = "C"   # Put a new point/string-item in the dict
print(f"{point_vals = }")       # Now there are three items.
point_vals[Point(3.0, 5.0)] = "D"  # Change value associated with p1.
print(f"{point_vals = }")       # There are still three items.
print(point_vals[p1])           # Ths gives us 'D'.
```

↓ `python3 point_with_hash_user.py` ↓

Listing 13.10: The stdout of the program point_with_hash_user.py given in Listing 13.9.

```
(p1 == p2) = False
(p1 == p3) = True
points = {Point(7, 8), Point(3, 5)}
p1 in points = True
p2 in points = True
p3 in points = True
Point(7.0, 8.0) in points = True
Point(3.1, 5) in points = False
point_vals = {Point(3, 5): 'A', Point(7, 8): 'B'}
point_vals = {Point(3, 5): 'A', Point(7, 8): 'B', Point(7, 9): 'C'}
point_vals = {Point(3, 5): 'D', Point(7, 8): 'B', Point(7, 9): 'C'}
D
```

13.3 Arithmetic Dunder and Ordering

Much of the actual behavior of Python's syntactx is implemented by dunder methods. Indeed, even the arithmetic operators `+`, `-`, `*`, and `/`. This allows us to define new numerical types if we want.

And since we did a lot of math-nerdery in this book already ... of course we want that! We implement the basic arithmetic operations for a class `Fraction` that represents fractions $q \in \mathbb{Q}$, i.e., it holds that $q = \frac{a}{b}$ with $a, b \in \mathbb{Z}$ and $b \neq 0$.[4] In other words, we want to pour primary school mathematics into a new numerical type. To refresh our memory, a is called the numerator and b is called the denominator of the fraction $\frac{a}{b}$.

Let's start implementing our class `Fraction` in file `fraction.py`. The overall code for this class is a bit longer compared to our previous examples. We therefore split it into several parts, namely Listings 13.11 to 13.15 (and later add Listing 13.19). At first we need to decide which attributes such a class would need. We construct the initializer dunder method `__init__` in Listing 13.11.

Since the fraction $\frac{a}{b}$ can be defined by the two integer numbers a and b, it makes sense to also have two `int` attributes `a` and `b`. We want our fractions to be immutable. You cannot change the value of `5`, and you should also not be able to change the value of `1/3`. The attributes will therefore receive the type hint `Final[int]` [546].

Our fractions should be *canonical*. It is totally possible that two fractions $\frac{a}{b} = \frac{c}{d}$ with $a \neq c$ and $b \neq c$. This is the case for, let's say, $\frac{-9}{3}$ and $\frac{12}{-4}$. In such cases we want to ideally store them in objects that have exactly the same attribute values.

The fractions $\frac{1}{2}$ and $\frac{2}{4}$ are the same. They should both be represented as $\frac{1}{2}$. It is clear that $\frac{a}{b} = \frac{c*a}{c*b}$ for all integer numbers $a, b, c \in \mathbb{Z}$ and $b, c > 0$. Before storing a and b, we will divide both numbers by their *greatest common divisor*. Back in Section 8.1, we implemented our own function `gcd` for computing the greatest common divisor based on the Euclidean algorithm. This time, we will use the `gcd` function from the `math` module directly. Anyway, dividing the numerator and denominator by their `gcd` ensures that our fractions are represented in a compact way.

This leaves only the question where the sign should be stored. Obviously, $\frac{-5}{2} = \frac{5}{-2}$ and $\frac{5}{2} = \frac{-5}{-2}$. We decide that the sign of the fraction is always stored in the attribute `a`. In other words, if $\frac{a}{b} < 0$, then `a` will be negative, otherwise it should be positive or 0. It can only be that $\frac{a}{b} < 0$ if exactly one of $a < 0$ or $b < 0$ is true. Therefore, the sign of our fraction is determined by `-1 if ((a < 0)!= (b < 0))else 1`.

In the initializer, we also need to make sure that things like $\frac{7}{0}$ do not happen. In this case, we will `raise` an `ZeroDivisionError`.

As last step, we must add proper doctests to the initializer. We need to check whether the values a and b are properly stored in the attributes `a` and `b`. Then we need to check whether our canonicalization by dividing with the `gcd` correctly maps $\frac{12}{2}$ to $\frac{6}{1}$. And we need to verify that $\frac{2}{-12}$ and $\frac{-2}{12}$ correctly become $\frac{-1}{6}$ while $\frac{-2}{-12}$ becomes $\frac{1}{6}$. The special case of the number zero also needs to be checked: We know that `gcd(0, -9)= -9`, so $\frac{0}{-9}$ should become $\frac{0}{1}$. Still, it is better to verify that. Finally, we need to verify that the `ZeroDivisionError` is indeed raised when we try to instantiate `Fraction` with a zero denominator. Without needing to read the actual code of `__init__`, a user can therefore already learn a lot about how our class `Fraction` represents rational numbers just from the doctests.

Several special fractions will occur very often in computations. Instead of creating them again and again, we can define them as constants. A constant is a variable that must never be changed.

Best Practice 67

Constants are module-level variables which must be assign a value upon definition and which must be annotated with the type hint `Final` [546].

Best Practice 68

The names of constants contain only capital letters with underscores separating words. Examples include `MAX_OVERFLOW` and `TOTAL` [594].

[4] Python already has such a type built-in. Our goal here is to explore dunder methods, so we make our own class instead. In any actual application, you would use the more efficient class `Fraction` from the module `fractions` [207].

Listing 13.11: Part 1 of the `Fraction` class: The initializer `__init__` and global constants. (src)

```
"""A new numerical type for fractions."""

from math import gcd
from types import NotImplementedType
from typing import Final, Union


class Fraction:
    """The class for fractions, i.e., rational numbers."""

    def __init__(self, a: int, b: int = 1) -> None:
        """
        Create a normalized fraction.

        :param a: the numerator
        :param b: the denominator

        >>> f"{Fraction(12, 1).a}, {Fraction(12, 1).b}"
        '12, 1'
        >>> f"{Fraction(12, 2).a}, {Fraction(12, 2).b}"
        '6, 1'
        >>> f"{Fraction(2, 12).a}, {Fraction(2, 12).b}"
        '1, 6'
        >>> f"{Fraction(2, -12).a}, {Fraction(2, -12).b}"
        '-1, 6'
        >>> f"{Fraction(-2, -12).a}, {Fraction(-2, -12).b}"
        '1, 6'
        >>> f"{Fraction(-2, 12).a}, {Fraction(-2, 12).b}"
        '-1, 6'
        >>> f"{Fraction(0, -9).a}, {Fraction(0, -9).b}"
        '0, 1'
        >>> try:
        ...     Fraction(1, 0)
        ... except ZeroDivisionError as z:
        ...     print(z)
        1/0
        """
        if b == 0:  # A denominator of zero is not permitted.
            raise ZeroDivisionError(f"{a}/{b}")
        g: int = gcd(a, b)  # We use the GCD to normalize the fraction.
        sign: int = -1 if ((a < 0) != (b < 0)) else 1  # The sign.
        #: the numerator of the fraction will also include the sign
        self.a: Final[int] = sign * abs(a // g)
        #: the denominator of the fraction will always be positive
        self.b: Final[int] = abs(b // g)


#: the constant zero
ZERO: Final[Fraction] = Fraction(0, 1)
#: the constant one
ONE: Final[Fraction] = Fraction(1, 1)
#: the constant 0.5
ONE_HALF: Final[Fraction] = Fraction(1, 2)
```

Listing 13.12: Part 2 of the `Fraction` class: String representation via `__str__` and `__repr__`. (src)

```
    def __str__(self) -> str:
        """
        Convert this number to a string fractional.

        :return: the string representation

        >>> print(Fraction(-5, 12))
        -5/12
        >>> print(Fraction(3, -1))
        -3
        >>> print(Fraction(12, 23))
        12/23
        """
        return str(self.a) if self.b == 1 else f"{self.a}/{self.b}"

    def __repr__(self) -> str:
        """
        Convert this number to a string.

        :return: the string representation

        >>> Fraction(-5, 12)
        Fraction(-5, 12)
        >>> Fraction(3, -1)
        Fraction(-3, 1)
        >>> Fraction(12, 23)
        Fraction(12, 23)
        """
        return f"Fraction({self.a}, {self.b})"
```

Best Practice 69

Constants are documented by writing a comment starting with `#:` immediately above them [525].

We define three constants, `ZERO`, `ONE`, and `ONE_HALF`, which hold corresponding instances of `Fraction`. These numbers are often used, and providing them as constant can safe both runtime and memory.

Did you notice that we had to write doctests in the docstring of our class in a very inconvenient way? This was because we did not yet define the `__str__` and `__repr__` methods for our class `Fraction`. We do this in Listing 13.12. The method `__str__` is supposed to return a compact representation of the fractions. We implement it such that it returns `self.a` as string if the denominator is one, i.e., if `self.b == 1`. Because then the fraction is actual an integer number. Otherwise, it should return `f"{self.a}/{self.b}"`.

This is easy and clear enough for each user to immediately recognize the value of the fraction. It is also ambiguous, though, because one cannot distinguish `str(Fraction(12, 1))` from `str(12)`. Fractions that represent integer numbers will produce the same strings as these integer numbers.

The `__repr__` method exists to produce unambiguous output. We implement it to return `f"Fraction({self.a}, {self.b})"`.

In the docstrings of both methods, we include doctests. Notice that `__str__` is used automatically if pass an object to `print`. This means that we can compare the expected output of `f.__str__()` for a fraction `f` to the result of `print(f)`. Otherwise, doctests always convert objects to string using `repr`. This means that the line `Fraction(-5, 12)` in the doctest of `__repr__` actually calls `repr(Fraction(-5, 12))`. Anyway, with the string conversion out of the way, we can begin to implement mathematical operators.

In Listing 13.13, we want to enable our `Fraction` class to be used with the `+` and `-` operators. In Python, doing something like `x + y` will invoke `x.__add__(y)`, if the class of `x` defines the `__add__`

Listing 13.13: Part 3 of the `Fraction` class: Addition (via `__add__`) and subtraction (via `__sub__`). (src)

```
    def __add__(self, other) -> Union[NotImplementedType, "Fraction"]:
        """
        Add this fraction to another fraction.

        :param other: the other number
        :return: the result of the addition

        >>> print(Fraction(1, 3) + Fraction(1, 2))
        5/6
        >>> print(Fraction(1, 2) + Fraction(1, 2))
        1
        >>> print(Fraction(21, -12) + Fraction(-33, 42))
        -71/28
        """
        return Fraction((self.a * other.b) + (other.a * self.b),
                        self.b * other.b) if isinstance(other, Fraction)\
            else NotImplemented

    def __sub__(self, other) -> Union[NotImplementedType, "Fraction"]:
        """
        Subtract this fraction from another fraction.

        :param other: the other fraction
        :return: the result of the subtraction

        >>> print(Fraction(1, 3) - Fraction(1, 2))
        -1/6
        >>> print(Fraction(1, 2) - Fraction(3, 6))
        0
        >>> print(Fraction(21, -12) - Fraction(-33, 42))
        -27/28
        """
        return Fraction(
            (self.a * other.b) - (other.a * self.b), self.b * other.b)\
            if isinstance(other, Fraction) else NotImplemented
```

Listing 13.14: Part 4 of the `Fraction` class: Multiplication (via `__mul__`), division (via `__truediv__`), and computing the absolute value (via `__abs__`). (src)

```
    def __mul__(self, other) -> Union[NotImplementedType, "Fraction"]:
        """
        Multiply this fraction with another fraction.

        :param other: the other fraction
        :return: the result of the multiplication

        >>> print(Fraction(6, 19) * Fraction(3, -7))
        -18/133
        """
        return Fraction(self.a * other.a, self.b * other.b) \
            if isinstance(other, Fraction) else NotImplemented

    def __truediv__(self, other) -> Union[NotImplementedType, "Fraction"]:
        """
        Divide this fraction by another fraction.

        :param other: the other fraction
        :return: the result of the division

        >>> print(Fraction(6, 19) / Fraction(3, -7))
        -14/19
        """
        return Fraction(self.a * other.b, self.b * other.a) \
            if isinstance(other, Fraction) else NotImplemented

    def __abs__(self) -> "Fraction":
        """
        Get the absolute value of this fraction.

        :return: the absolute value.

        >>> print(abs(Fraction(-1, 2)))
        1/2
        >>> print(abs(Fraction(3, 5)))
        3/5
        """
        return self if self.a > 0 else Fraction(-self.a, self.b)
```

method. From primary school, we remember that $\frac{a}{b} + \frac{c}{d} = \frac{a*d+c*b}{b*d}$, for $b, d \neq 0$. Therefore, if `other` is also an instance of `Fraction`, then `__add__(other)` computes the result like that and creates a new `Fraction`. Notice that the initializer of that new fraction will automatically normalize the fraction by using `gcd`.

If `other` is not an instance of `Fraction`, we return `NotImplemented`. We already know this from our implementation of `__eq__` for points. This result enables Python to look for other routes to perform addition with our objects. Here, Python would then look whether `other` provides a `__radd__` method that does not return `NotImplemented` ... but we will not implement all possible arithmetic dunder methods here so we skip this one.

Either way, the behavior of this method is again be tested with doctests. These tests check that $\frac{1}{3}+\frac{1}{2}$ actually yields $\frac{5}{6}$. We test whether $\frac{1}{2}+\frac{1}{2}$ really returns $\frac{1}{1}$. Then we also check correct normalization by trying whether $\frac{21}{-12} + \frac{-33}{42} = \frac{882+396}{-504} = \frac{1278}{-504} = \frac{18*1278}{18*-28} = \frac{-71}{28}$.

After confirming that these tests succeed, we continue by implementing the `__sub__` method in exactly the same way. This enables subtraction by using `-`, because `x - y` will invoke `x.__sub__(y)`, if the class of `x` defines the `__sub__` method. Clearly, $\frac{a}{b} - \frac{c}{d} = \frac{a*d-c*b}{b*d}$ for $b, d \neq 0$. As doctests, the same three cases as used for `__add__` will do.

In Listing 13.14, we now focus on multiplication and division. The `*` operation will utilize the method `__mul__`, if implemented. The `/` operation uses the method `__truediv__`, if implemented. Multiplying the fractions $\frac{a}{b}$ and $\frac{c}{d}$ yields $\frac{a*c}{b*d}$ for $b, d \neq 0$. Dividing $\frac{a}{b}$ by $\frac{c}{d}$ yields $\frac{a*d}{b*c}$ for $b, c, d \neq 0$. The dunder methods can be implemented according to the same schematic as before. We test multiplication by confirming that $\frac{6}{19} * \frac{3}{-7} = \frac{6*3}{19*-7} = \frac{18}{-133} = \frac{-18}{133}$. The division is tested by computing whether $\frac{6}{19} * \frac{3}{-7} = \frac{6*-7}{19*3} = \frac{-42}{57} = \frac{3*-14}{3*19}$ indeed gives us $\frac{-14}{19}$.

Now we also implement support for the `abs` function. `abs` returns the absolute value of a number. Therefore, `abs(5)` = `abs(-5)` = `5`. `abs(x)` will invoke `x.__abs__()`, if present. We can implement this method as follows: If our fraction is positive, then it can be returned as-is. Otherwise, we return a new, positive variant of our fraction.

Finally, in Listing 13.15 we implement the six rich comparison dunder methods given in [589]:

- `__eq__` implements the functionality of `==`, as we already discussed before.
- `__ne__` implements the functionality of `!=`.
- `__lt__` implements the functionality of `<`.
- `__le__` implements the functionality of `<=`.
- `__gt__` implements the functionality of `>`.
- `__ge__` implements the functionality of `>=`.

Implementing equality and inequality is rather easy, since our fractions are all normalized. For two fractions `x` and `y`, it holds that `x == y` if and only if `x.a == y.a` and `x.b == y.b`. `__eq__` is thus quickly implemented. `__ne__` is its complement for the `!=` operator. `x != y` is `True` if either `x.a != y.a` or `x.b != y.b`.

The other four comparison methods can be implemented by remembering how we used the common denominator for addition and subtraction. We did addition like this: $\frac{a}{b} + \frac{c}{d} = \frac{a*d}{b*d} + \frac{c*b}{b*d} = \frac{a*d+c*b}{b*d}$. Looking at this again, we realize that $\frac{a}{b} < \frac{c}{d}$ is the same as $\frac{a*d}{b*d} < \frac{c*b}{b*d}$. This, in turn, must be the same as $a * d < c * b$. Thus, $\frac{a}{b} \leq \frac{c}{d}$ is the same as $a * d \leq c * b$. The greater and greater-or-equal operations can be defined the other way around.

All six comparison operations are defined accordingly in Listing 13.15. This time, I omitted doctests for the sake of space. Matter of fact, I have shortened the code and tests in all of the above code snippets. For example, we do not check whether the parameters of the initializer `__init__` are actually integers (and raise a `TypeError` otherwise).

Such checks should then be covered by additional unit tests. You should never omit such checks and tests. In your program code, you *do* have space. You can also pack them into additional modules. Your code does not need to fit on book pages...

Anyway, in Listing 13.16 we present the output of `pytest` running the doctests of all the methods we implemented. All of them succeed. This means that we can be fairly confident that using our `Fraction` class in real computations would provide us correct results.

There are quite a few more dunder methods for implementing arithmetic operations. And this truly is a fun thing to do. We can have our classes for fractions and complex numbers. Well, `Python` already has those. But we could have complex numbers based on fractions. The sky is the limit!

Listing 13.15: Part 5 of the `Fraction` class: All order-related dunder methods. (src)

```
    def __eq__(self, other) -> bool | NotImplementedType:
        """
        Check whether this fraction equals another fraction.

        :param other: the other fraction
        :returns: `True` if `self` equals `other`, `False` otherwise
        """
        return (self.a == other.a) and (self.b == other.b) \
            if isinstance(other, Fraction) else NotImplemented

    def __ne__(self, other) -> bool | NotImplementedType:
        """
        Check whether this fraction does not equal another fraction.

        :param other: the other fraction
        :returns: `False` if `self` equals `other`, `True` otherwise
        """
        return (self.a != other.a) or (self.b != other.b) \
            if isinstance(other, Fraction) else NotImplemented

    def __lt__(self, other) -> bool | NotImplementedType:
        """
        Check whether this fraction is less than another fraction.

        :param other: the other fraction
        :returns: `True` if `self` less than `other`, `False` otherwise
        """
        return ((self.a * other.b) < (other.a * self.b)) \
            if isinstance(other, Fraction) else NotImplemented

    def __le__(self, other) -> bool | NotImplementedType:
        """
        Check whether this fraction is less than or equal to another.

        :param other: the other fraction
        :returns: `True` if `self <= other`, `False` otherwise
        """
        return ((self.a * other.b) <= (other.a * self.b)) \
            if isinstance(other, Fraction) else NotImplemented

    def __gt__(self, other) -> bool | NotImplementedType:
        """
        Check whether this fraction is greater than another fraction.

        :param other: the other fraction
        :returns: `True` if `self > other`, `False` otherwise
        """
        return ((self.a * other.b) > (other.a * self.b)) \
            if isinstance(other, Fraction) else NotImplemented

    def __ge__(self, other) -> bool | NotImplementedType:
        """
        Check whether this fraction is greater than or equal to another.

        :param other: the other fraction
        :returns: `True` if `self >= other`, `False` otherwise
        """
        return ((self.a * other.b) >= (other.a * self.b)) \
            if isinstance(other, Fraction) else NotImplemented
```

Listing 13.16: The output of pytest executing the doctests for our `Fraction` class.

```
$ pytest --timeout=10 --no-header --tb=short --doctest-modules fraction.py
========================== test session starts ==========================
collected 9 items
fraction.py .........                                             [100%]
========================== 9 passed in 0.01s ===========================
# pytest 9.2.0 with pytest-timeout 2.4.0 succeeded with exit code 0.
```

13.4 Interlude: Debugging

We now want to use mathematics based on our class `Fraction` for some "real" computation. Remember back in Section 7.6, we implemented the algorithm of Heron to compute the square root using a `while` loop. In Section 8.2, we then poured this code into a function. If we revisit this function `sqrt` in Listing 8.5, we notice that it computes the square root using only comparison, addition, multiplication, and division. We do have these operations available for `Fractions`! This would mean that we could now compute the square root of a number to an arbitrary precision. Well, we will need to introduce some sort of stopping criterion, but given that... ...We could compute $\sqrt{2}$ accurate to 700 digits!

Except, so far, we do not have anything that would print these digits. Our `__str__` method returns fractions as text of the form `"a/b"`. Had we computed a fraction that approximates $\sqrt{2}$ to some precision, this could maybe be printed as $\frac{6369051672525773}{4503599627370496}$. Instead, we want it to print as 1.4142135623730951. So we first need to implement a method `decimal_str`, which translates a `Fraction` to such a decimal string. Since some fractions, like $\frac{1}{3}$ and $\frac{1}{7}$ have never-ending decimal representations, this function needs a parameter `max_frac` specifying the maximum number of fractional digits to generate. We will set it to 100 by default.

In file `fraction_decimal_str_err.py`, we implement a new variant of our class `Fraction`. In Listing 13.17, we show an excerpt of this file, presenting the part of our class `Fraction` that contains the code converting the fraction to a decimal string. Like all of our code in `Fraction`, it takes the straightforward and probably inefficient route. The idea is simply to first cut-off the integer part of the fraction and then to produce the fractional digits one-by-one.

We begin our method `decimal_str` by first copying the numerator into a variable `a`. If `a == 0`, then the whole fraction is 0 and we can directly return `"0"`. Otherwise, we check if the fraction is negative. The Boolean variable `negative` is set to `True` if `a < 0` and to `False` otherwise. We then make sure that `a` is positive by setting it to `abs(a)`. Then we also copy the denominator into a variable `b`.

We will not change the local variables `negative` and `b` in our method, so we mark them both with the type hint `Final`.

Best Practice 70

Every time you declare a variable that you do not intend to change, mark it with the type hint `Final` [546]. On the one hand, this conveys the intention "this does not change" to anybody who reads the code. On the other hand, if you do accidentally change it later, tools like `Mypy` can notify you about this error in your code.

In a `while` loop, we now fill a list `digits` with the digits representing the fraction. The loop will continue until either `a == 0` or our list `digits` contains `max_frac` decimals. Let us assume that our fraction is $-\frac{179}{16}$. Then `negative == True`, `a = 179`, and `b = 16`. In the loop body, we append the result of the integer division of `a` by `b`, i.e., `a // b`, to the list `digits`. In the first iteration, this gives us `179 // 16`. So the first "digit" we append to `digits` is `11`. This is the integer part of our fraction and this is the only time that a digit larger than 9 can appear. We now update `a` to `10 * (a %b)`. `%` is the modulo division. Thus, `a % b` gives us the remainder of the division of 179 by 16, namely 3. Thus, we get `a = 30`.

In the second iteration, `a // b`, i.e., `30 // 16`, gives us the next digit `1`. Now, `10 * (a %b)` becomes 140 as the new value of `a`. This then gives `140 // 16`, namely 8, as the third digit and `a` is updated to `10 * (a %b)`, which is 120. At the beginning of the fourth iteration, `a = 120` (while `b = 16` remains unchanged). The fourth value appended to `digits` therefore is `120 // 16 == 7`. The variable `a` is updated to the result of `10 * (a %b)`, which is 80. As last digit, we therefore add `80 // 16`, which is `5`. This is the last digit, because `80 % 16` is 0. Therefore, `a == 0` holds after the fifth iteration. This makes the first part of the loop condition (`a != 0`) become `False` and the loop terminates.

At this point, `digits == [11, 1, 8, 7, 5]`. This is also right, because $\frac{179}{16} = 11.1875$.

Notice that there are two conditions which can make the loop stop: It stops if we can represent the fraction completely and exhaustively as decimal string with no more than the specified number `max_frac` of digits. This was the case in our example. The loop also stop if we reach the maximum number of fractional digits, i.e., if `len(digits)` exceeds `max_frac`.

Listing 13.17: Part 6 of the `Fraction` class: Adding a `decimal_str` conversion method. (src)

```
    def decimal_str(self, max_frac: int = 100) -> str:
        """
        Convert the fraction to decimal string.

        :param max_frac: the maximum number of fractional digits
        :return: the string

        >>> Fraction(124, 2).decimal_str()
        '62'
        >>> Fraction(1, 2).decimal_str()
        '0.5'
        >>> Fraction(1, 3).decimal_str(10)
        '0.3333333333'
        >>> Fraction(-101001, 100000000).decimal_str()
        '-0.00101001'
        >>> Fraction(1235, 1000).decimal_str(2)
        '1.24'
        >>> Fraction(99995, 100000).decimal_str(5)
        '0.99995'
        >>> Fraction(91995, 100000).decimal_str(3)
        '0.92'
        >>> Fraction(99995, 100000).decimal_str(4)
        '1'
        """
        a: int = self.a  # Get the numerator.
        if a == 0:  # If the fraction is 0, we return 0.
            return "0"
        negative: Final[bool] = a < 0  # Get the sign of the fraction.
        a = abs(a)  # Make sure that `a` is now positive.
        b: Final[int] = self.b  # Get the denominator.

        digits: Final[list] = []  # A list for collecting digits.
        while (a != 0) and (len(digits) <= max_frac):  # Create digits.
            digits.append(a // b)  # Add the current digit.
            a = 10 * (a % b)  # Ten times the remainder -> next digit.

        if (a // b) >= 5:  # Do we need to round up?
            digits[-1] += 1  # Round up by incrementing last digit.

        if len(digits) <= 1:  # Do we only have an integer part?
            return str((-1 if negative else 1) * digits[0])

        digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
        if negative:  # Do we need to restore the sign?
            digits.insert(0, "-")  # Insert the sign at the beginning.
        return "".join(map(str, digits))  # Join all digits to a string.
```

But this can also cause problems. For example, what happens if I want to represent $\frac{10006}{10000} = 1.0006$ with only *3* fractional digits. After the loop, the list `digits` would be `[1, 0, 0, 0]`. At this stage, we would have `a = 60000` and `b = 10000`. The 6 at the end of the numerator is a bit annoying: If we represent this fraction with three digits, it should be 1.001, not the 1.000 corresponding to our `digits` list.

Well, all we need to do to fix this is to check whether the next digit that we would not append to `digits` would be greater or equal to 5. If so, we increment the last digit in `digits` by 1. (This idea will later turn out to be an error...) Anyway, to introduce this rounding behavior, we insert an `if (a // b)>= 5` which does `digits[-1] += 1` if its condition is met.

We now have a representation of the fraction as a list of digits. All we need to do is to convert these to a string and return them to the user.

Listing 13.18: The output of pytest executing the doctests for our `Fraction` class with the `decimal_str` method from Listing 13.17: It fails!

```
$ pytest --timeout=10 --no-header --tb=short --doctest-modules
    ↪ fraction_decimal_str_err.py
========================== test session starts ==========================
collected 1 item
fraction_decimal_str_err.py F                                      [100%]
================================ FAILURES ================================
________ [doctest] fraction_decimal_str_err.Fraction.decimal_str ________
037 '0.5'
038 >>> Fraction(1, 3).decimal_str(10)
039 '0.3333333333'
040 >>> Fraction(-101001, 100000000).decimal_str()
041 '-0.00101001'
042 >>> Fraction(1235, 1000).decimal_str(2)
043 '1.24'
044 >>> Fraction(99995, 100000).decimal_str(5)
045 '0.99995'
046 >>> Fraction(91995, 100000).decimal_str(3)
Expected:
    '0.92'
Got:
    '0.9110'
/home/runner/work/programmingWithPython/programmingWithPython/__git__/
    ↪ realms/git/gh_thomasWeise_programmingWithPythonCode/dunder/
    ↪ fraction_decimal_str_err.py:46: DocTestFailure
======================== short test summary info ========================
FAILED fraction_decimal_str_err.py::fraction_decimal_str_err.Fraction.
    ↪ decimal_str
========================== 1 failed in 0.01s ==========================
# pytest 9.2.0 with pytest-timeout 2.4.0 failed with exit code 1.
```

First, we check if we need to insert a decimal dot (".") . If we only have a single digit, then our fraction is an integer number and we can return it as such. Thus, `if len(digits)<= 1`, we convert the single digit to a string (after re-inserting the sign).

Otherwise, we need to have a decimal dot "." after the first number in `digits`. We can use `digits.insert(1, ".")` to place it there. In our original example of $\frac{-179}{16}$, we first had `digits == [11, 1, 8, 7, 5]`. After this step, we get `digits == [11, ".", 1, 8, 7, 5]`.

If the fraction was `negative`, we place a minus sign before the list contents, via `digits.insert(0, "-")`. In our example, this means that we get `digits == ["-", 11, ".", 1, 8, 7, 5]`.

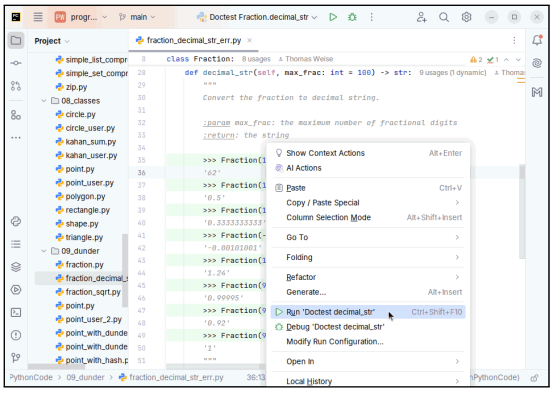
All what remains is translate all the integers to strings and to concatenate the result. A single line of code takes care of that: `"".join(map(str, digits))`. `map` returns an `Iterator` that returns the results of the `str` function applied to all the elements in `digits` one by one. `str` applied to a string just returns the string itself. Applied to an integer, it converts it to a string.

The method `join` of a string concatenates all elements of the `Iterable` it receives as parameter and places the string itself as separator. For example, `"X".join(["a", "b", "c"])` would yield `"aXbXc"`. We use the empty string for joining, so for our example, we finally return `"-11.1875"`.

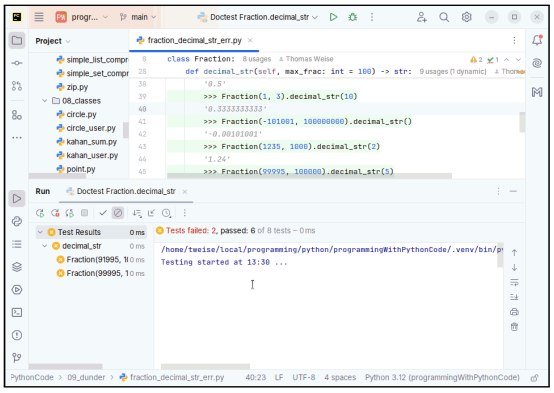
After finishing the implementation of `decimal_str`, what remains to do is to test it. This can be done by placing some doctests into the docstring of the method. As test cases, we choose several normal situations as well as several corner cases. We first test that integer numbers are correctly represented. It is clear that `Fraction(124, 2).decimal_str()` should yield `"62"`. Then we check a simple fraction that can exactly be translated: `Fraction(1, 2).decimal_str()` should result in `"0.5"`.

As fraction that cannot be exactly written down as finite decimal string, we choose $\frac{1}{3}$. $\frac{1}{3}$ to ten digits after the dot should yield `"0.3333333333"`. As an example of negative fractions and also as an example of fractions with multiple leading zeros, we use $\frac{-101001}{100000000}$. This should give us `"-0.00101001"`.

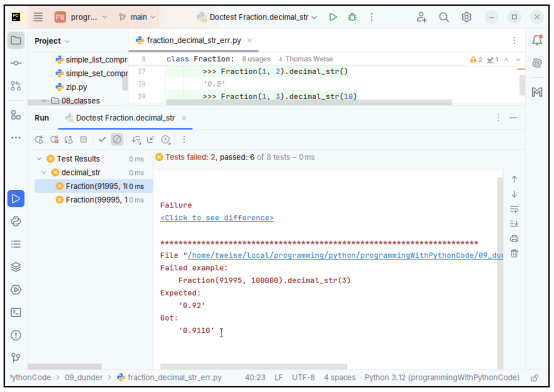
As test for rounding off the last digit, we expect that `Fraction(1235, 1000).decimal_str(2)` should yield `"1.24"`. This number would have three fractional digits, but we only want two. Since the third



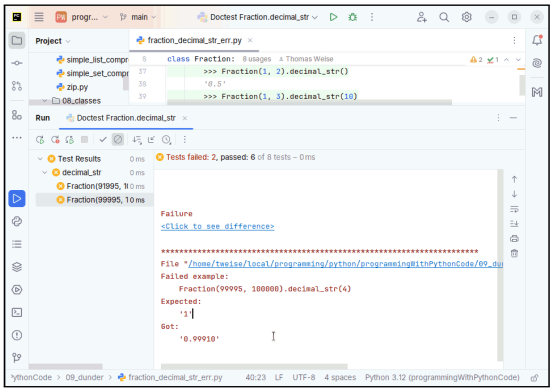
(13.1.1) We open a contact menu by right-clicking into out function. We left-click on Run 'Doctest decimal_str'.



(13.1.2) The doctests are run, and in the window at the bottom-left, we see the failing tests.



(13.1.3) Left-clicking on the first failed test in the small window shows us the test output in bottom-centered window.



(13.1.4) Left-clicking on the second failed test in the small window shows us the test output in bottom-centered window.

Figure 13.1: Running doctests in PyCharm.

digit would be a 5, the rounding should should occur. Instead of `"1.235"` or `"1.23"`, we would expect to see `"1.24"`. `Fraction(99995, 100000).decimal_str(5)`, i.e., 0.99995 rounded to five decimal digits, should yield `"0.99995"`.

Computing `Fraction(91995, 100000).decimal_str(3)` means rounding 0.91995 to three decimals. The last digit, a 5, would be cut off. This means that we need to round-up, which would make the second-to-last digit (9) to also be rounded up, causing the next 9 to toggle as well. The 1 would then be rounded up to a 2 and we should get `"0.92"`. A similar thing should happen when we evaluate `Fraction(99995, 100000).decimal_str(4)`: Rounding 0.9995 to four digits will round-up the 5, which will cause all the 9s to be rounded up, too. We finally should get result `"1"`.

In Listing 13.18, we find the results of the doctests ran with pytest. Interestingly, they fail! The output tells us that `Fraction(91995, 100000).decimal_str(3)` does not yield the expected `"0.92"`. Instead, we get `"0.9110"`. Where does the trailing zero come from? And why do we have two 1s? Even if we did not round correctly, at least we should get something like 0.919, but certainly not 0.911?

We want to investigate this very strange error. First, let us repeat the doctests by also executing them inside PyCharm in Figure 13.1. We open our source file `fraction_decimal_str_err.py` and scroll to our method `decimal_str`. With a right mouse click, a context menu is opened. Here, we then left-click on Run 'Doctest decimal_str' (Figure 13.1.1).

This executes *all* the doctests. In the small window at the bottom-left, we can see the *failing* tests (Figure 13.1.2). We can click on these failed tests to get more information. A left-click on the first failed test in this window in the bottom-left will then display the output of that text in the bottom-centered window (Figure 13.1.3). This is the same information we already saw in Listing 13.18. What we did not see in that output is that actually *two* doctests failed. A left-click on the second failed test in Figure 13.1.3 tells us that `Fraction(99995, 100000).decimal_str(4)` did not yield the

expected `"1"`. Instead, it produced `"0.99910"`. Why is there a `"0"` at the end of our number? Where did it come from? Zeros at the end should not be possible with our code. Also, there are four 9s in our number, not three. What went wrong here?

We are clueless why these tests fail. The question arises: What can we do?

If we want to find where things go wrong, it would be very useful if we could somehow execute our program step-by-step. When I explained how `decimal_str` works, I used $\frac{-179}{16}$ as an example and explained what the method would do. Would it not be nice if we could actually step-by-step execute the program for the test? Then we could see what it *actually* does? Luckily, we can do that! With a tool called debugger which ships with `Python` and `PyCharm`.

Useful Tool 10

A debugger is a tool that ships with many programming languages and IDEs. It allows you to execute a program step-by-step while observing the current values of variables. This way, you can find errors in the code more easily [5, 483, 636]. A comprehensive example on how to use the debugger in `PyCharm` is given in Section 13.4.

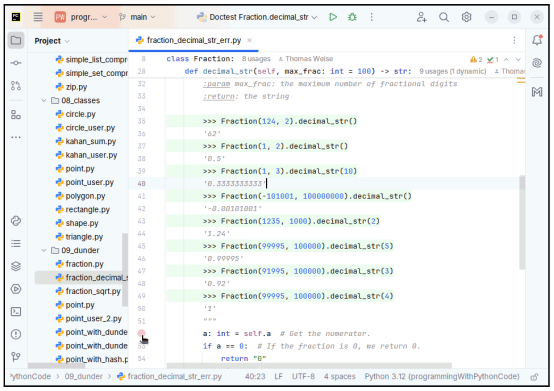
In `PyCharm`, we can apply the debugger to a complete program, but also to doctests. This is what we will do in Figure 13.2. First, we again open our file `fraction_decimal_str_err.py` in `PyCharm` and locate our function `decimal_str` (Figure 13.2.1).

On the left side of our code window, there is a column with the line numbers. We can left-click on a line to place a breakpoint there. A breakpoint is a mark in the IDE at which we later want the program execution to pause. We want our program to pause right at the beginning of `decimal_str`. Therefore, we place the breakpoint there (Figure 13.2.2). The breakpoint is shown as a red ball over the line number.

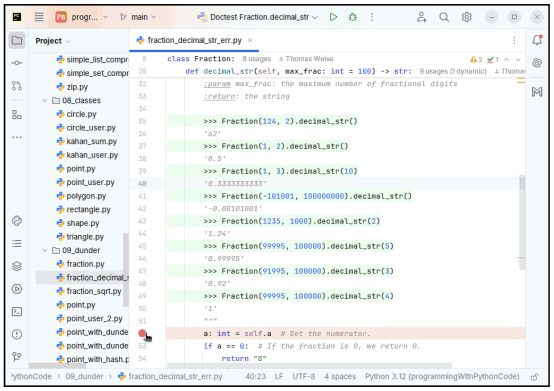
In order to begin the debugging process, we again open the context menu by right-clicking into the doctest. This time, instead of *running* the doctest, we click Debug 'Doctest decimal_str' (Figure 13.2.3). The doctests will now be executed. Instead of running them completely, the debugger kicks in: The execution is paused at exactly our breakpoint. This line of code is *not yet executed*, but marked in blue (Figure 13.2.4).

Before we continue, we look at the bottom of our `PyCharm` window. There is a row with a Debug register. We can activate it with the right mouse button on the top of this row and then drag it upwards. Now we get a division of our window that contains the debug information. Most importantly, in the register Threads & Variables, we can see the values of all local variables at the current point in execution (Figure 13.2.5).

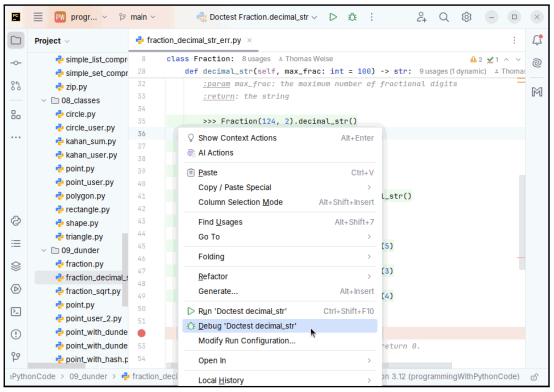
We see that `max_frac` has the (default) value `100`. When clicking on the variable `self`, we see that the numerator `a` of the current fraction has value `62`, while the denominator `b` is `1`. This is exactly what we expect: Our first test case was `Fraction(124, 2).decimal_str()`, so the normalized fraction is indeed $\frac{62}{1}$ (Figure 13.2.6).



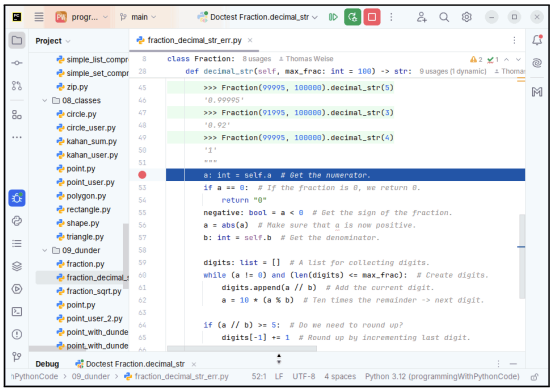
(13.2.1) We open our source code file in PyCharm and locate our function `decimal_str`.



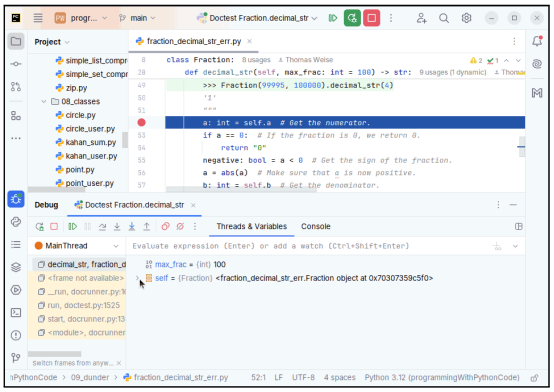
(13.2.2) We place a breakpoint at the first line of our function by right-clicking on the line number. This is denoted by the red ball over the line number. Later, the program execution will pause at this location.



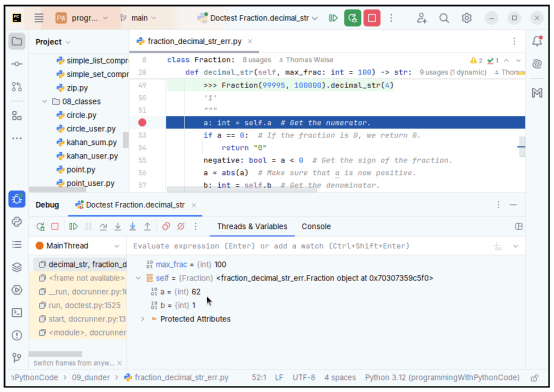
(13.2.3) We open the context menu by right-clicking into the doctest and click Debug 'Doctest decimal_str'.



(13.2.4) The execution of the doctests begins, but is immediately paused the first time the breakpoint is reached. This line is now marked with blue color, but not yet executed.



(13.2.5) We drag the Debug register up from the bottom of our PyCharm window. We find the new register Threads & Variables, where we can see the value of all local variables at the current point in execution.



(13.2.6) Clicking on the variable `self` reveals that we are in the first test case, where `Fraction(124, 2)` is tested, which was normalized to `Fraction(62, 1)` by the initializer `__init__`.

Figure 13.2: Using the debugger in PyCharm.

We know that this test case will be completed successfully. We thus are not interested in it. We click on the ▯▷ symbol in Debug register, which will let the program continue its execution (Figure 13.2.7). Alternatively, we can hit F9, which has the same effect.

The execution of the doctests is resumed. It again pauses at our breakpoint (Figure 13.2.8). This time, we can see that we have arrived at the beginning of the second doctest case with `Fraction(1, 2)`. This test case, too, is not interesting. So we again either click ▯▷ or press F9 to resume execution.

This takes us to the beginning of the third doctest case, where the fraction is $\frac{1}{3}$ and `max_frac` is `10` (Figure 13.2.9). As we already know, this test case is again one that will succeed. It will not give us any useful information. We can skip it as well by pressing F9.

The next time we reach the breakpoint is for the fourth doctest case, $\frac{-101001}{100000000}$ (Figure 13.2.10). We skip that one as well.

When the debugger arrives at the fifth test case, `Fraction(1235, 1000)`, we find that this fraction has been normalized correctly to $\frac{247}{200}$. Nonetheless, we can skip this test case via F9, too, because we know that it will succeed (Figure 13.2.11).

This takes us to the last successful doctest case, `Fraction(99995, 100000)`, which corresponds to $\frac{19999}{20000}$ in Figure 13.2.12. After skipping it by pressing ▯▷, we will finally arrive at the cases that did fail and which we hence want to investigate step-by-step.

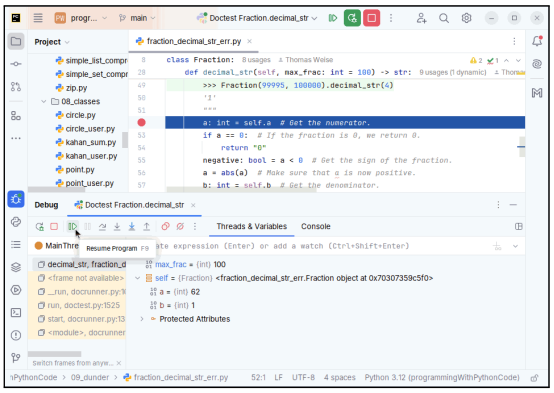
Figure 13.2.13 shows that we now arrived at the beginning of the failing doctest case `Fraction(91995, 100000).decimal_str(3)`. The fraction $\frac{91995}{100000}$ got normalized to $\frac{18399}{20000}$ in the initializer `__init__`. The parameter `max_frac` of `decimal_str` has the value `3`, as we can see in the Threads & Variables window. We now want to execute the `decimal_str` method step-by-step. Right now, the debugger has paused the execution right at the very first line of this method. This line has not yet been executed.

In Figure 13.2.14, we execute this line of code, either by pressing the ⇗ button or by hitting F8. We can see in Figure 13.2.15 that now a new variable has appeared in the Threads & Variables window. Since we executed `a = self.a`, the local variable `a` now exists and has value `18399`. Now, the next line of code that can be executed is marked with blue color.

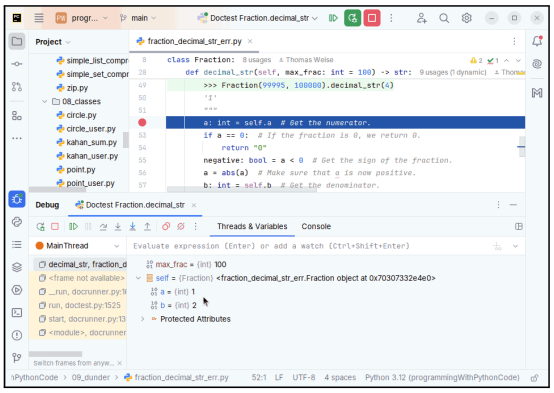
By pressing F8, the `if a == 0:` is executed. Since `a == 0` is not `True`, the body of the `if` is not executed. The program jumps right over it. The next line after the `if` is marked Figure 13.2.16. We execute it by pressing F8.

The local variable `negative` is created. Since `a < 0` is `False`, `negative` is `False`, too. The next line of code is marked and we press ⇗ to execute it with F8 (Figure 13.2.17).

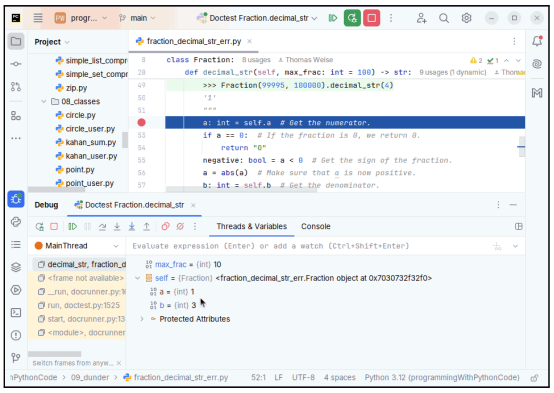
`a = abs(a)` has no effect, since `a` is already positive. We press F8 to continue (Figure 13.2.18).



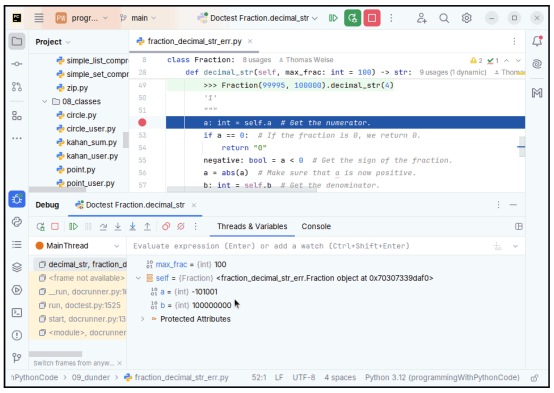
(13.2.7) The first doctest case $\frac{124}{2}$ is uninteresting, so we continue the program execution by clicking ▷‖ or hitting F9.



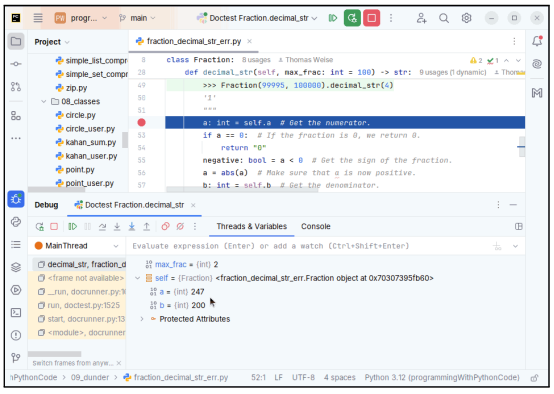
(13.2.8) The second doctest case $\frac{1}{2}$ is also uninteresting. We continue the program execution by clicking ▷‖.



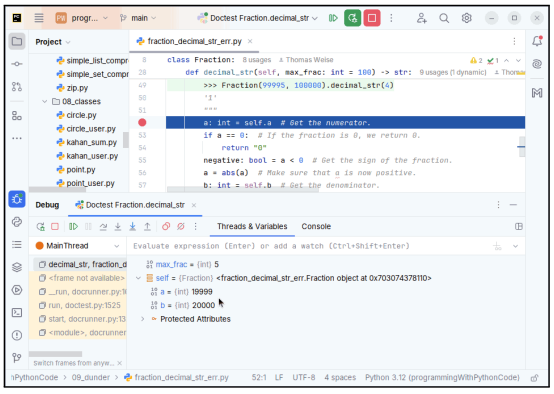
(13.2.9) The third doctest case, where the fraction is $\frac{1}{3}$ and `max_frac` is `10`. We again skip over it by pressing F9.



(13.2.10) The fourth doctest case $\frac{-101001}{100000000}$ can be skipped as well.

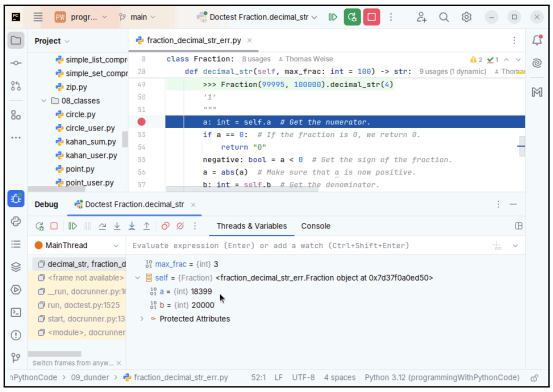


(13.2.11) The test case $\frac{1235}{1000}$, normalized to $\frac{247}{200}$, also was successful and can be skipped by pressing ▷‖.

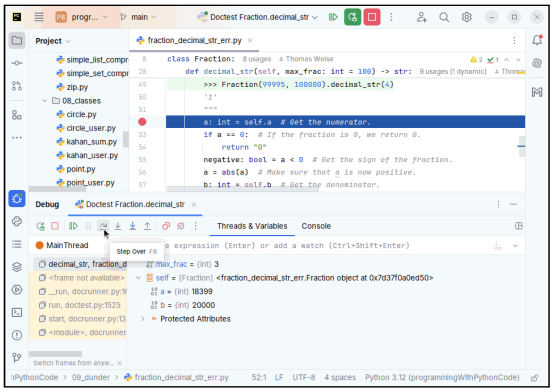


(13.2.12) The last one of the uninteresting doctest cases: `Fraction(99995, 100000)`. We again press F9.

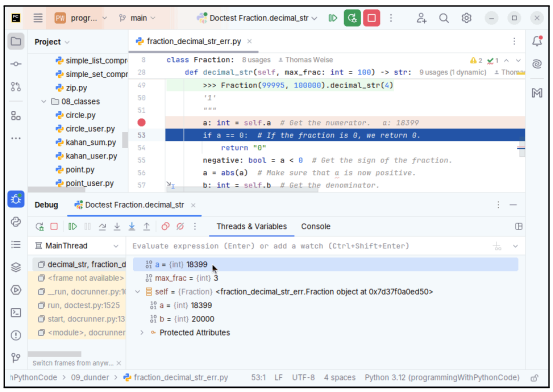
Figure 13.2: Using the debugger in PyCharm (continued).



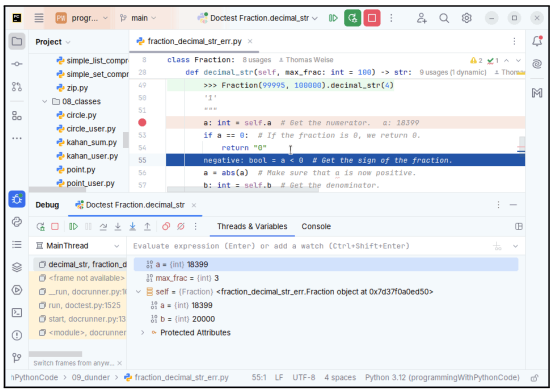
(13.2.13) We arrive at the beginning of the failing doctest case `Fraction(91995, 100000).decimal_str(3)`. The `max_frac` parameter has value `3`, `self.a` is `18399` and `self.b` is `20000`, because the fraction was normalized in `__init__`.



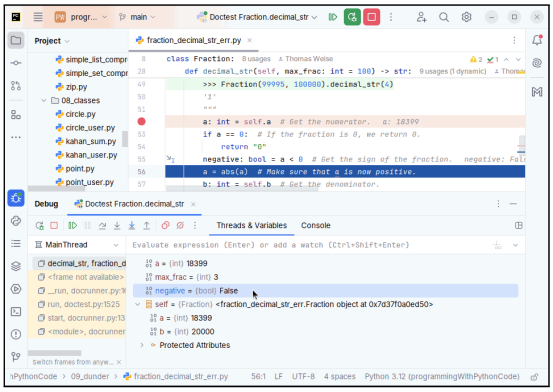
(13.2.14) We now execute the first line of the `decimal_str` function where the debugger has paused. This is done by either pressing the ⤼ button or by hitting F8.



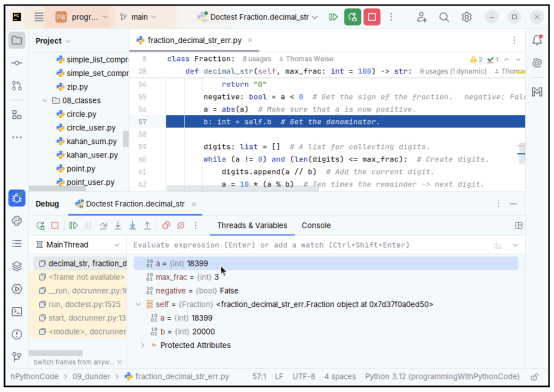
(13.2.15) The execution of the assignment `a = self.a` creates a new local variable `a` with value `18399`. We press F8 to continue the execution.



(13.2.16) The condition for the `if` is not met, so the execution jumps over its body and the next line after the `if` is marked. We execute it by pressing F8.



(13.2.17) The new local variable `negative` with value `False` appears. The next line of code is marked and we execute it by pressing ⤼.



(13.2.18) `a = abs(a)` has no effect, since `a` is already positive. We press F8 to continue.

Figure 13.2: Using the debugger in PyCharm (continued).

This executes `b = self.b`. Thus, the new local variable `b` with value `20000` is created in Figure 13.2.19. We are now at the last line of "trivial setup" of our `decimal_str` method, the creation of the list `digits`. We continue debugging by pressing F8.

The new variable `digits` has indeed appeared in Figure 13.2.20. It is an empty list `[]`. We arrived at the beginning the `while` loop. We press F8, which will cause the condition of the loop being checked.

In Figure 13.2.21, we find that now the first line of the loop's body is marked. This means that `a != 0` and `len(digits)<= max_frac` are both `True`. And they should be, since `a` is `18399`, `len(digits)` is 0, and `max_frac` is 3. We press the ⤳ button to execute the first line of the loop body.

`digits.append(a // b)` will append the value `18399 // 20000` to the list `digits`. As this is the result of an integer division where the denominator is larger than the numerator, `digits` is now `[0]` (Figure 13.2.22). We press F8 to continue.

Now, `a = 10 * (a %b)` is executed. Since `18399 %20000` is still `18399`, `a` becomes `183990` in Figure 13.2.23. The head of the loop is now marked again. We press the ⤳ button to let execute it. In Figure 13.2.24, the loop condition is still met, so the first line in the loop body is marked again.

In Figures 13.2.25 to 13.2.27 and 13.2.29 to 13.2.32 we work our way through the loop in the same way, by pressing F8 repeatedly. First, `9` gets appended to `digits`, then `a` gets updated to `39900`. In the following iteration, `1` gets appended to `digits`, then `a` gets updated to `199000`. In the loop iteration after that, `9` gets appended to `digits` and `a` gets updated to `190000`.

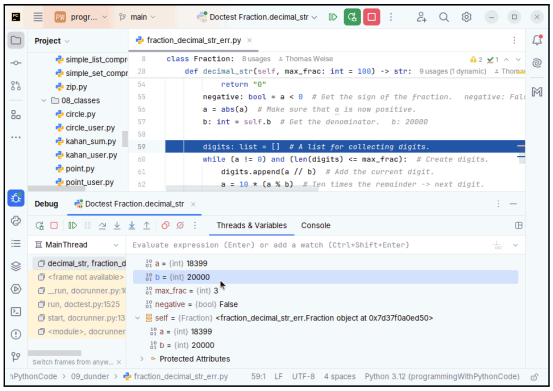
So far, everything looks right. At this stage, `digits` has become `[0, 9, 1, 9]`. Since `max_frac` is `3`, `len(digits)<= max_frac` is no longer `True`. In Figure 13.2.32, we have arrived back at the head of the loop. When we hit F8, the loop condition is evaluated again. This time it evaluates to `False`. The loop terminates and the mark is placed on the next line of code after the loop in Figure 13.2.33.

If we look at what was computed so far, we find that everything is exactly as it should be. We want to translate the fraction $\frac{91995}{100000}$ to a decimal string with three fractional digits. So far, we got the digits 0, 9, 1, and 9 in Figure 13.2.34.

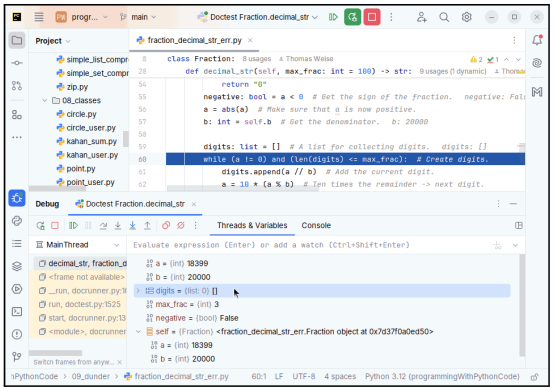
The next line of code, `if (a // b)>= 5`, is supposed to check whether we should round up the last digit. Since `a` is `190000` and `b` is still `20000`, `a // b` is `9`. So the condition should be met. We press the ⤳ button to find it out.

A look at Figure 13.2.35 reveals the bug, i.e., the error in our code: In order to round up, we incremented the last number in our list `digits`. `digits` was `[0, 9, 1, 9]`. So it is `[0, 9, 1, 10]`.

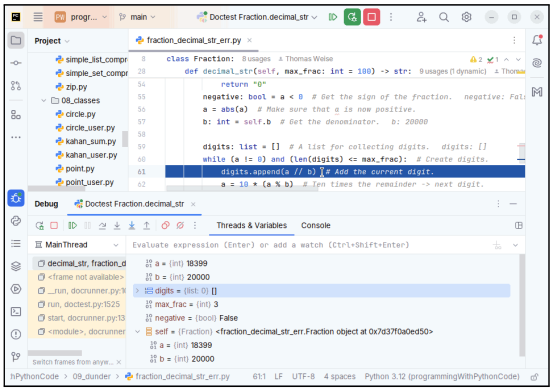
The strange trailing zeros in our output were not separate digits. They were the zeros of a ten. We did not consider that, when rounding up, we do not just have simple cases like 1.25 which we can round to 1.3 by only incrementing one digit. We can have cases like 0.9999, which rounds up to 1, even if we want three fractional digits of precision. Our code does not do this. We can stop the debugging here and go back to our code.



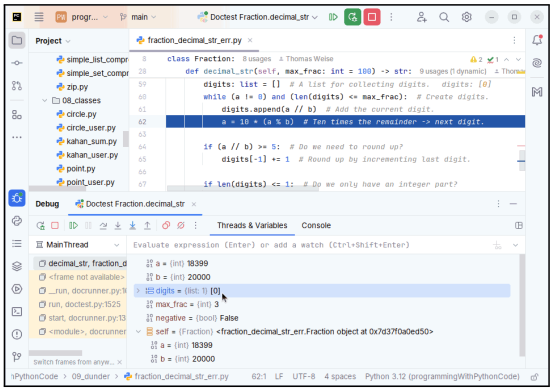
(13.2.19) After executing `b = self.b`, the new local variable `b` with value `20000` comes into existence. We continue debugging by pressing F8.



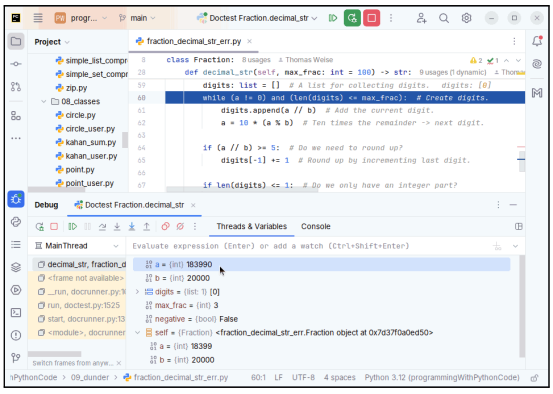
(13.2.20) The empty list `digits` has been created. By pressing F8, the executing `while` loop will begin by checking its condition.



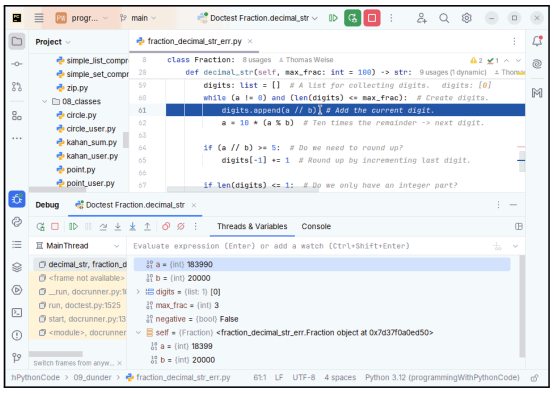
(13.2.21) The condition of the `while` loop is met. The first line of the loop's body is marked. We press ⭳ to execute it.



(13.2.22) `digits` now contains the result of `a // b`, i.e., is `[0]`. We press F8.

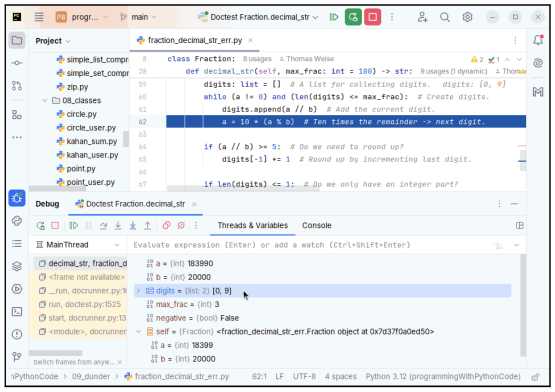


(13.2.23) `a` now is `183990`. We press ⭳ to continue.

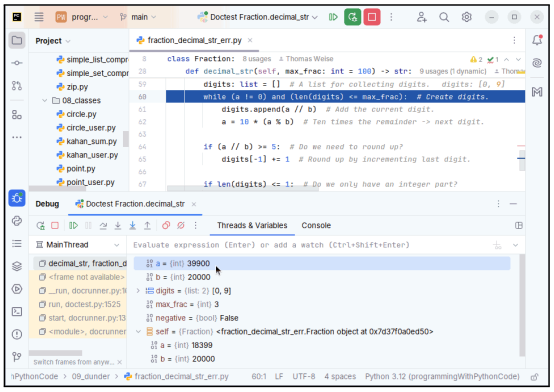


(13.2.24) We hit the F8 key to continue. The loop condition is still met, so the first line of the loop body is marked again.

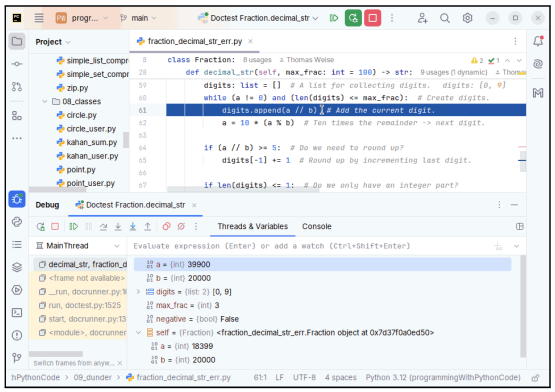
Figure 13.2: Using the debugger in PyCharm (continued).



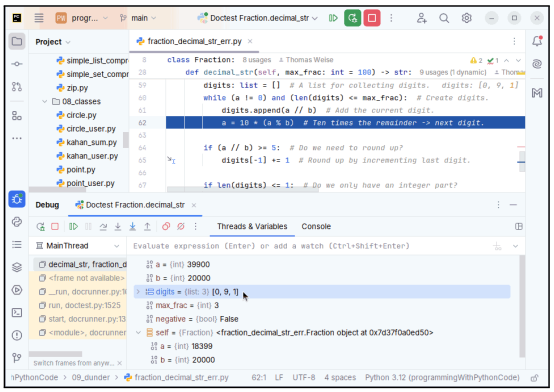
(13.2.25) `9` gets appended to `digits`.



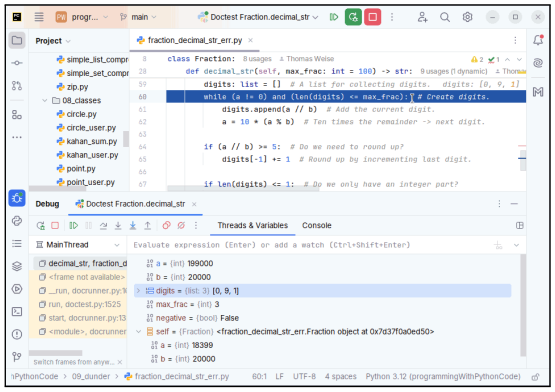
(13.2.26) `a` gets updated to `39900`.



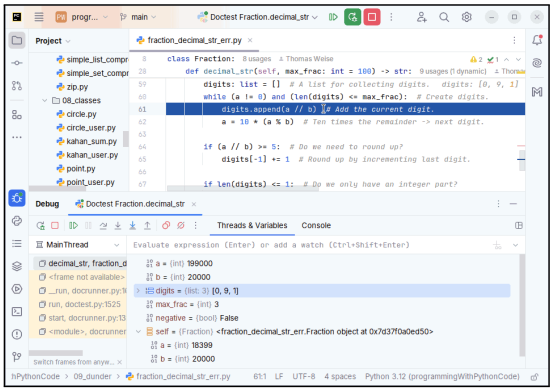
(13.2.27) The loop condition is still met.



(13.2.28) `1` gets appended to `digits`.

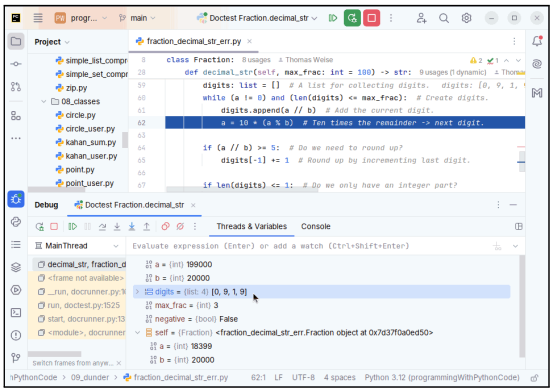


(13.2.29) `a` gets updated to `199000`.

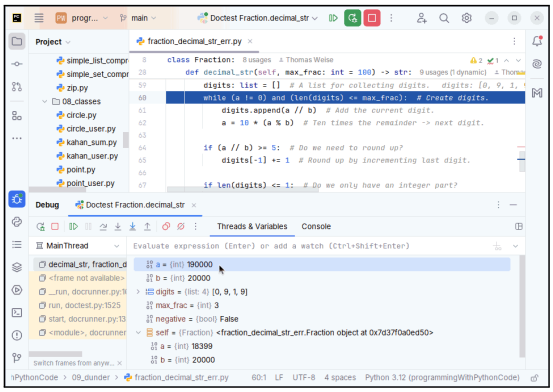


(13.2.30) The loop condition is still met.

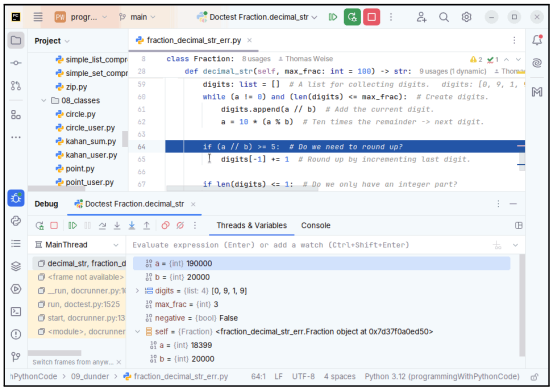
Figure 13.2: Using the debugger in PyCharm (continued).



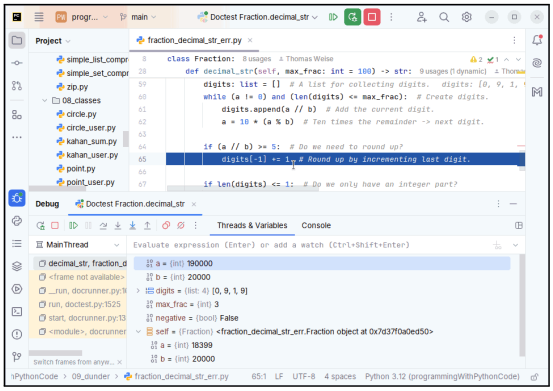
(13.2.31) `9` gets appended to `digits`.



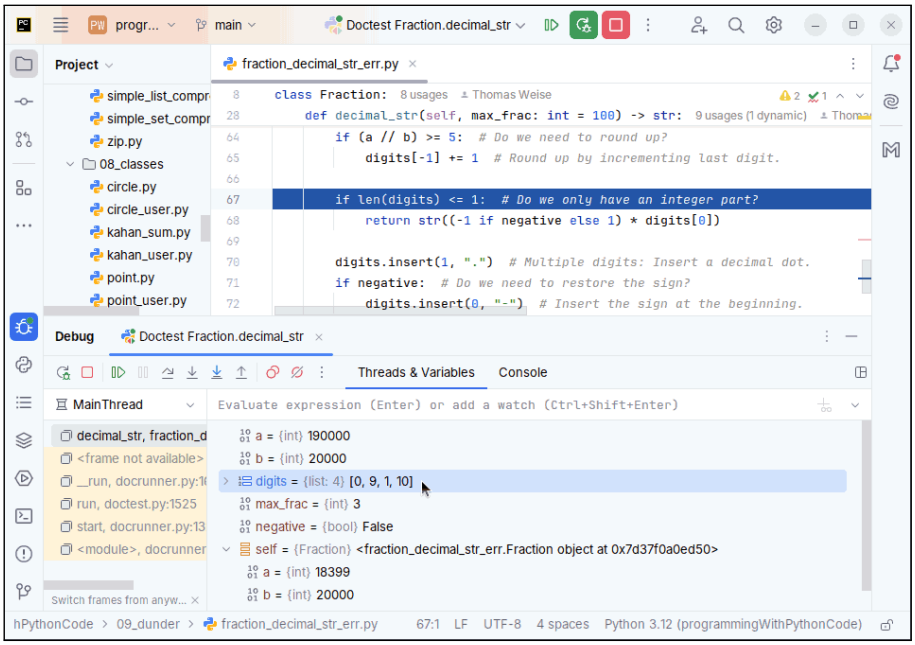
(13.2.32) `a` gets updated to `190000`.



(13.2.33) Since the loop condition evaluates to `False`, the cursor is now at the next line after the loop body. This line checks whether we should round up the next digit. We hit `F8`.



(13.2.34) The condition of the `if` is met, we can now execute its body. We press the ⤼ button.



(13.2.35) Our code for rounding up actually turned the last digit into a `10`! Because we did not consider that rounding up could cause the next 9 to become a 10, which should be represented as a 0 and lead to the next digit to be rounded up as well.

Figure 13.2: Using the debugger in PyCharm (continued).

In Listing 13.19, we revisit the `decimal_str` method of our class `Fraction`. Our rounding-up code becomes more complex: First, we need to loop over all fractional digits, from the end of the list `digits` forward: `for i in range(len(digits)- 1, 0, -1)` does this. If `len(digits)== 5`, then `range(len(digits)- 1, 0, -1)` iterates over the numbers 4, 3, 2, and 1. We increment the digit at index `i` by 1. If it does not become 10, then we can stop the loop via `break`. If it did become 10, then we set it to zero and continue the loop. This will increment the next digit, and so on. If we arrive at index 1 and still need to continue, the regular loop execution ends anyway.

Then, the `else` statement is hit. Recall from Section 7.7 that the `else` statement at the bottom of a loop is *only* executed if the loop finished regularly, i.e., if no `break` statement was executed. Therefore, if and only if the digit at index 1 also became 10 and was then set to 0, we increment the digit at index 0. This digit represents the integer part of our fraction. Here, it is totally OK to round a 9 to a 10. For example, 9.999 can be rounded to 10, and 1239.9 can be rounded to 1240.

This new code for rounding numbers may introduce zeros at the end of our string. We just gobble them up with an additional `while` loop directly after the rounding. And with this, we are done. We have working code converting fractional numbers to decimal strings. All the doctests now pass.

But let's get back to what we actually wanted to do: We wanted to compute the square roots of numbers at insane precision. For this, we can now revisit Listing 8.5. In Listing 13.20 with file `fraction_sqrt.py`, we just copy the code of the `sqrt` function from this file. Then we apply some necessary changes. We will basically replace all `floats` in this code with `Fractions`.

The function's signature changes from `def sqrt(number: float)-> float` to `def sqrt(number: Fraction)-> Fraction`. Well, we know that the square roots of some numbers like $\sqrt{2}$ are irrational. This means that we cannot represent the exactly with an instance of `Fraction`.

In the original code, this is not a problem. Due to the limited range of `floats`, the function will eventually stop. This happens when all of the 15 to 16 digits of precision are exhausted.

However, using `Fraction`, the same code could loop forever. We have a precision limited only by the memory of the computer. So the function could forever try to find better and better approximations.

Thus we need to limit the number iterations with an additional parameter. Thus, we add the parameter `max_steps: int = 10`. You will later see that this default value of only permitting ten steps is already quite sufficient to get very nice approximations.

In Listing 13.20, we replace the numbers `0.0`, `0.5`, and `1.0` with our constants `ZERO`, `ONE_HALF`, and `ONE`. We need to do this because our mathematical dunder methods only work with other instances of `Fraction`. We could have easily implemented them to also support working with instances of `int` or `float`. We did not do that because otherwise, our example code would have been much longer. Either way, all the numerical constants in Listing 13.20 are now instances of `Fraction`.

Our new `sqrt` function begins with an initial check whether the input number is negative. If so, raises an `ArithmeticError`. After that, it has almost the same loop and body as our original variant in Listing 8.5. The only difference is that we decrement `max_steps` by 1 in each iteration. We `break` out the loop once it reaches `0`.

We, of course, add doctests to our function. For example, we test the expression `sqrt(Fraction(2, 1)).decimal_str(750)`. This computes $\sqrt{\frac{2}{1}} = \sqrt{2}$ by performing ten steps of Heron's algorithm. Then it translates the result to a decimal string with *750 fractional digits*. We take the correct value from [408, 518]. Unless our function provides this number of digits correctly *after only ten steps*, the doctest will fail!

The second doctest is simply to check whether $\sqrt{4}$ renders as 2 when converted to a decimal string. Of course, we just approximate the square root through several steps. Thus, the value of the fraction might not actually be 2. However, when rendered to 100 decimals of precision, it should return `"2"`.

Finally, we want to compute the golden ratio ϕ [96, 187, 517], which equals $\frac{1+\sqrt{5}}{2}$. We can write this as `ONE_HALF * (ONE + sqrt(Fraction(5, 1)))`. In the doctest, we want to see whether our results are correct to 420 fractional digits, which we take from [192].

The results of the doctests executed by `pytest` can be seen in Listing 13.21. They all pass. All we did was implementing primary school math into a class in Python, together with an algorithm which is almost 2000 years old. And with *ten steps* of that algorithm, we got approximations of $\sqrt{2}$ as well as ϕ that were accurate to *several hundreds of digits*. Isn't that cool?

In this section, we have learned how to use the debugger. The debugger is one of the most important tools in the tool belt of a programmer. The inner workings of code always become clear when going through it step-by-step. We can set breakpoints and let a process run until it hits them. Then we can observe the values of variables. We can execute every line of code separately. We can see which variables change. We can follow the control flow as it moves through the branches of alternatives and as it iterates through loops. If our code has an error, i.e., a bug, then using the debugger may be the first step to finding it. I guess that's why it's called debugger.

Listing 13.19: The repaired Part 6 of the `Fraction` class: A correct `decimal_str` method. (src)

```
    def decimal_str(self, max_frac: int = 100) -> str:
        """
        Convert the fraction to decimal string.

        :param max_frac: the maximum number of fractional digits
        :return: the string

        >>> Fraction(124, 2).decimal_str()
        '62'
        >>> Fraction(1, 2).decimal_str()
        '0.5'
        >>> Fraction(1, 3).decimal_str(10)
        '0.3333333333'
        >>> Fraction(-101001, 100000000).decimal_str()
        '-0.00101001'
        >>> Fraction(1235, 1000).decimal_str(2)
        '1.24'
        >>> Fraction(99995, 100000).decimal_str(5)
        '0.99995'
        >>> Fraction(91995, 100000).decimal_str(3)
        '0.92'
        >>> Fraction(99995, 100000).decimal_str(4)
        '1'
        """
        a: int = self.a  # Get the numerator.
        if a == 0:  # If the fraction is 0, we return 0.
            return "0"
        negative: Final[bool] = a < 0  # Get the sign of the fraction.
        a = abs(a)  # Make sure that `a` is now positive.
        b: Final[int] = self.b  # Get the denominator.

        digits: Final[list] = []  # A list for collecting digits.
        while (a != 0) and (len(digits) <= max_frac):  # Create digits.
            digits.append(a // b)  # Add the current digit.
            a = 10 * (a % b)  # Ten times the remainder -> next digit.

        if (a // b) >= 5:  # Do we need to round up?
            # This may lead to other digits topple over, e.g., 0.999...
            for i in range(len(digits) - 1, 0, -1):  # except first!
                digits[i] += 1  # Increment the digit at position i.
                if digits[i] != 10:  # Was there no overflow?
                    break  # Digits in 1..9, no overflow, we can stop.
                digits[i] = 0  # We got a `10`, so we set it to 0.
            else:  # This is only reached if no `break` was done.
                digits[0] += 1  # Increment the integer part.

        while digits[-1] == 0:  # Remove all trailing zeros.
            del digits[-1]  # Delete the trailing zero.

        if len(digits) <= 1:  # Do we only have an integer part?
            return str((-1 if negative else 1) * digits[0])

        digits.insert(1, ".")  # Multiple digits: Insert a decimal dot.
        if negative:  # Do we need to restore the sign?
            digits.insert(0, "-")  # Insert the sign at the beginning.
        return "".join(map(str, digits))  # Join all digits to a string.
```

Listing 13.20: Using the `Fraction` class to compute square roots. (src)

```
"""A square root algorithm based on fractions."""

from fraction import ONE, ONE_HALF, ZERO, Fraction


def sqrt(number: Fraction, max_steps: int = 10) -> Fraction:
    """
    Compute the square root of a given :class:`Fraction`.

    :param number: The rational number to compute the square root of.
    :param max_steps: the maximum number of steps, defaults to `10`
    :return: A value `v` such that `v * v` is approximately `number`.

    >>> sqrt(Fraction(2, 1)).decimal_str(750)
    '1.414213562373095048801688724209698078569671875376948073176679737\
9\
907324784621070388503875343276415727350138462309122970249248360558507\
37\
212644121497099935831413222665927505592755799950501152782060571470109\
55\
997160597027453459686201472851741864088919860955232923048430871432145\
08\
397626036279952514079896872533965463318088296406206152583523950547457\
50\
287759961729835575220337531857011354374603408498847160386899970699004\
81\
503054402779031645424782306849293691862158057846311159666871301301561\
85\
689872372352885092648612494977154218334204285686060146824720771435854\
87\
415565706967765372022648544701585880162075847492265722600208558446652\
14\
583988939443709265918003113882464681570826301005948587040031864803421\
94\
897278290641045072636881313739855256117322040 25'
    >>> sqrt(Fraction(4, 1)).decimal_str()
    '2'
    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
    '1.618033988749894848204586834365638117720309179805762862135448622\
7\
052604628189024497072072041893911374847540880753868917521266338622235\
36\
931793180060766726354433389086595939582905638322661319928290267880675\
20\
876689250171169620703222104321626954862629631361443814975870122034080\
58\
879544547492461856953648644492410443207713449470495658467885098743394\
42\
212544877066478091588460749988712400765217057517978834166256249407589\
07'
    """
    if number < ZERO:  # No negative numbers are permitted.
        raise ArithmeticError(f"Cannot computed sqrt({number}).")
    guess: Fraction = ONE       # This will hold the current guess.
    old_guess: Fraction = ZERO  # 0.0 is just a dummy value != guess.
    while old_guess != guess:  # Repeat until nothing changes anymore.
        old_guess = guess  # The current guess becomes the old guess.
        guess = ONE_HALF * (guess + number / guess)  # The new guess.
        max_steps -= 1      # Reduce the number of remaining steps.
        if max_steps <= 0:  # If we have exhausted the maximum steps...
            break           # ...then we stop (and return the guess).
    return guess  # Return the final guess.
```

Listing 13.21: The (successful) output of `pytest` executing the doctests for our `sqrt` function from Listing 13.20 working on instances of the `Fraction` class.

```
$ pytest --timeout=10 --no-header --tb=short --doctest-modules
    ↪ fraction_sqrt.py
========================== test session starts ==========================
collected 1 item
fraction_sqrt.py .                                                  [100%]
=========================== 1 passed in 0.01s ===========================
# pytest 9.2.0 with pytest-timeout 2.4.0 succeeded with exit code 0.
```

13.5 Implementing Context Managers for the `with` Statement Revisited

In Section 9.4.5, we introduced the `with` block, which is often used to ensure that resources are properly closed after using them. We used text files as an example: A text file is opened in the head of `with` for writing using the `open` function. This function returns a stream to which we then can write strings of text in the body of the `with` block. Then, after this body ends, the stream is closed automatically. This happens even if an exception was raised inside the body of the `with`.

Back then, we stated that resources like files are implemented in Python as context managers. A context manager has a special method that is called at the beginning of the `with` block. The method can be used to "open" the resource, for example. The context manager also has a special method that is called at the end the `with` block. This special method can be used to "close" the resource.

At this stage, it will not surprise you that these "special methods" are actually dunder methods. These methods are `__enter__` and `__exit__`. Since `with` blocks are nice syntactical sugar of the Python language, we will here play around with them a little bit.

As an example, let us create a simple API that allows us to write output in a subset of the Extensible Markup Language (XML) format [75, 125, 323]. XML is a format for data interchange which was predominant in distributed systems the 2000s. After that, it began to fade out. It got replaced [10] by JavaScript Object Notation (JSON) [74, 556] and YAML Ain't Markup Language™ (YAML) [124, 170, 323] in many applications. It is still very relevant today, for example, as foundation of several document formats such as those used in LibreOffice [215, 356] and Microsoft Word [166, 396], or as the basis for the SVG format [141]. If you are a bit familiar with web design, then you will find that XML looks a bit like the Hyper Text Markup Language (HTML) [263].

As shown below, an XML documents begins with the XML version declaration `<?xml version="1.0"?>`. Then, it includes XML elements that can be arbitrarily nested. Each elements has a name and begins with an opening and closing string, looking somewhat like `<name>...</name>`. Between the opening and closing string, text and other elements can be included. An element can have attributes stored in the opening string, which looks like `<name key='value'... >...`.

```
<?xml version="1.0"?>
<class title='Programming with Python' year='2024'>
    <description>This is a class on Python.</description>
    <teacher name='Thomas'>I like Python.</teacher>
    <students>
        <student name='Bubba'></student>
        <student name='Bibba'>I like the &#38;Programming with Python&#62;
            ↪ book.</student>
    </students>
</class>
```

Here we have a nicely formatted XML text about this course. The `<class>` element has two attributes, `title` and `year`. It contains other elements, such as `<description>` with a brief description of the class. Then follows the element `<teacher>` with an attribute storing my given name as well as brief text. Finally, there is an element `<students>`, which, in turn, holds two elements of type `<student>`. Each of them hold the student's given name as attribute `name`. The second `<student>` element includes some additional text. In this next, you notice that two characters have been *escaped*: since `<` and `>` are used to mark the beginning and ending of the start and end element strings, they should not occur inside the normal text, as this may confuse the XML parsers. Therefore, they are escaped as entities `&` and `>`.

Would it not be nice to have a simple API that allows us to produce valid XML and that takes care of the escaping of special characters? While countless such tools already exist ... let us make our own. For this, we realize: First, every XML element that is opened must also be closed. Second, XML elements can be nested arbitrarily. This kind of looks like an application of the `with` statement.

We now make our own context manager. When the `with` statement begins, our context manager will write the element start text. Inside the body of the `with` statement, we will allow to write the element text or to open sub-elements. At the end of the `with` statement, the XML element closing text should be written. Furthermore, we want to be able to direct the output of our API to any destination

where strings can be written to, say, `print`, to `lists`, or to files. As you will see in Listings 13.22 and 13.23, we can implement all of that in a fairly small module.

In Listing 13.22, we begin by defining an internal constant `_ESC`, which we use for XML escaping special characters in strings with XML entities. For this purpose, we use the functions `maketrans` and `translate` of the class `str`. The former accepts a dictionary where the keys are single characters and the corresponding values are strings with which these characters should be replaced. It creates some Python-internal data structure which then can be passed to `translate` to perform the replacement. For example, we could do `x = str.maketrans({"A": "XYZ"})`. Then, `"ABCBA".translate(x)` would yield `"XYZBCBXYZ"`. This is very useful to painlessly implement rudimentary support to escape special characters based on the XML standard [75].

Then we define the `class Element`. The initializer `__init__` of this class has four important parameters. `dest` is a `Callable`, i.e., function, that can accept a single string as parameter. This will be where all the output of our API is sent to. `name` is a string with the name of the element. `attrs` contains the attributes of the element. It is either `None` if there are no attributes (which is the default). Otherwise, it is a `dict` mapping string keys to arbitrary values. Finally, `is_root` is a Boolean value that is `True` if this element is the single root element of the XML document that we want to write, and `False` if it is some nested element. This is needed, because before writing the root element, the XML version declaration `<?xml version="1.0"?>` must be written. This must happen only once and only at the beginning of an XML document.

In the initializer, we store the `dest` argument in an attribute that is both private (signified by the two leading underscores) and immutable (signified by the type hint `Final`). We then construct the element start string by filling a list `head` and store its concatenated elements in the attribute `__head`. Only if our element is a root element, the XML version declaration is included in the list `head`. The actual element start string begins by the less-than symbol and the element name (`<name`). Then, if `attrs` is neither `None` nor empty, we want to add the attributes. Interestingly, this condition can be expressed by a simple `if attrs:`. If there are attributes, then we iterate over the key-value pairs in `attrs` by writing `for key, value in attrs.items():`. We use an f-string composed of the `key` and a string for the value, which we construct in a clever way: Since we permit arbitrary value types, we convert the `value` to a string first, using the `str` function. Then, we escape special characters, using the aforementioned `translate` routine with our map `_ESC`. It should be noted that only character here that needs to actually be escaped is the "`'`" character, which therefore also was included in the construction of `_ESC`. This is because we also use this character to delimit our strings: The `!r` format specifier in the f-string adds these single quotation marks. Finally, the element start string ends with `">"`. With `"".join(head)`, all strings in the list `head` are merged into one.

Constructing the element end string is much easier, we simply write `ameself.__foot: Final[str] = f"</n>"` Notice that, so far, we have just cached strings and did not yet write anything to the output `dest`.

This should come when a `with` body begins. At this moment, the Python interpreter will call the method `__enter__` of the context manager. All we need to do in this method is to invoke `self.__dest(self.__head)`. If `__dest` was, for example, the `print` function, this would write the element start to the `stdout` stream. The method `__enter__` also must return an object which can be assigned to a variable for use inside the `with` block using the `as` statement. We here want to return our `Element` object itself, so we write `return self`. The proper type hint for the return type in such situations is `Self`.

Best Practice 71

Methods that return `self` should be annotated with the type hint `Self` [529]. Static code analysis tools then see that the method always returns an object of the same class as the object itself.

Either way, `__enter__` will be called at the beginning of a `with` statement. At the end of an `with` statement, in other words, when we are done using our object, the `__exit__` method will be called. Then, we should write the element end text stored in the attribute `__foot`. Therefore, we write `self.__dest(self.__foot)`.

The `__exit__` method looks a bit different from `__enter__`. First, it also gets three parameters: the exception type `exc_type`, the exception value `exc_value`, and the stack `traceback`. These parameters

are filled with the information about any `Exception` raised within the `with` body. Otherwise, they will be `None` [639]. We here just define a single parameter `*exc`. This syntax tells `Python` to capture all positional arguments into a single tuple. Since we do not care about exceptions here, we just write this to safe space, honestly, because I do not want to write three lines of docstring where one suffices. All we want is to write the element end string. We then always return `False`, which basically tells the `Python` interpreter: *"If any exception occurred inside the `with` statement, please raise this exception again after we are finished with `__exit__`."* The proper type hint for this is literal is `Literal[False]`.

Best Practice 72

If a parameter of a function can take on only some special constant values `X`, say certain integer or string literals, the proper type hint is `Literal[X]` [346, 562]. The same holds if your function always returns the same, simple constant(s) `X`.

With this, we have basically completed implementing our context manager. What is still missing is some functionality to write text into an element and to convenient branching off new sub-elements.

The former is very easy: We define a method `text` taking a string `txt` as input parameter. All we have to do in the body of this method is `self.__dest(txt.translate(_ESC))`. This escapes any dangerous XML special characters in the string and passes the result on to the output destination.

The method `element` is used to branch off new sub-elements. It basically would take the same parameters as the initializer `__init__`. However, the new element must use the same output destination and it cannot be a root element (because it is a sub-element). Therefore, we only need to pass the parameters `name` and `attrs` to a new instance of `Element`, whereas we hand over `self.__dest` as destination and `False` as `is_root`. With this, our simple API for a subset of the XML standard [75] is already completed.

We now use this API in Listings 13.24 and 13.26 to basically reproduce the small XML snippet that I showed you before. In the former example, we use `print` as destination function. This means that any string that our API passes to its internal `__dest` attribute will immediately be written as a single line to the `stdout`.

We begin by creating the `<class>` element with the `title` and `year` attributes. We do this by creating a new instance of `Element` in a `with` block and by passing on the attributes as `dict`. This object is stored as variable `cls` via the "`as`" statement. We the create the `description` element by writing `with cls.element("description")as desc`. The class description can then be written via `desc.text(...)`. Notice that our API writes the element start and end strings to the output without requiring us to do anything. It also escapes all special characters for us. The other elements are created in the same convenient fashion. Our `Python` code basically mirrors the XML structure. The output of the program in Listing 13.25 looks very much like our example, with a slightly different indentation and line breaking. But this is permitted and acceptable under the XML standard [75].

Instead of writing to the `stdout` stream, we can also recall our very first example of the `with` statement back in Section 9.4.5: writing to and reading from a file. Listing 13.26 is almost exactly the same as Listing 13.24. The difference is that we now `open` a text output stream `stream_out` to a file `example.xml` and the stream's `write` method as destination to our API. As a result, all the text is now written to a file instead.

We later `open` the file again and read all of its lines one-by-one via the `for line in stream_in` loop. Files are actually `Iterators` of line strings! All the file contents get written to the `stdout` stream. Since `write` does not append newline characters, this format is much more compact, as can be seen in Listing 13.27. But it is perfectly valid XML.

We now have learned how the functionality of the `with` statement is implemented internally. And we used it to hammer together a very compact and yet functional API for a subset of the XML standard [75]. Obviously, we do not implement the complete standard, which is much more complicated. And you should never use our class if you really wanted to produce XML in a productive code. We also omitted type and sanity checks – for example, we should forbid element names that are empty or contain special characters like `<`. A real implementation would be more conservative (and too long to serve as a good example in this book). Still, on the one hand, our XML is valid. On the other hand, you *could* use extend and improve this class to have all the functionality that you need, if you wanted to. So this is

actually another example that, at this stage and with what you have learned, you can already do real things.

Listing 13.22: Part 1 of our very simply context manager-based XML output API. (src)

```
"""An API for XML output via context managers and `with`."""

from typing import Any, Callable, Final, Literal, Self

#: An internal mapping for escaping reserved XML characters.
_ESC: Final = str.maketrans({"<": "&#38;", ">": "&#62;", "'": "&#39;"})


class Element:
    """An XML element. XML elements can be nested arbitrarily deeply."""

    def __init__(self, dest: Callable[[str], Any],
                 name: str, attrs: dict[str, Any] | None = None,
                 is_root: bool = True) -> None:
        """
        Create the XML element.

        :param dest: the function to receive the text output
        :param name: the name of the element
        :param attrs: the attributes, if any, otherwise `None`
        :param is_root: is this the root element?
        """
        #: the destination, i.e., a function receiving all the output
        self.__dest: Final[Callable[[str], Any]] = dest  # protected var

        head: list[str] = ['<?xml version="1.0"?>\n'] if is_root else []
        head.append(f"<{name}")
        if attrs:  # If attrs is neither None nor empty...
            for key, value in attrs.items():  # ...append as key='value'
                head.append(f" {key}={str(value).translate(_ESC)!r}")
        head.append(">")  # Close the element start.

        #: the header: XML declaration (if root) plus the element start
        self.__head: Final[str] = "".join(head)  # Merge string list.
        #: the element closing text, i.e., something like `</myElement>`
        self.__foot: Final[str] = f"</{name}>"

    def __enter__(self) -> Self:
        """
        Enter the XML element context and write the element start text.

        :returns: this element itself
        """
        self.__dest(self.__head)  # write the header
        return self  # Return this object itself.

    def __exit__(self, *exc) -> Literal[False]:
        """
        We are done with this context: Close the XML element.

        :param exc: the exception information, which we ignore
        :returns: always `False`
        """
        self.__dest(self.__foot)  # write the element closing
        return False  # re-raise exception that occurred in with, if any
```

Listing 13.23: Part 2 of our very simply context manager-based XML output API. (src)

```
    def element(self, name: str,
                attrs: dict[str, Any] | None = None) -> "Element":
        """
        Create a new XML Element inside this element.

        :param name: the name of the element
        :param attrs: the attributes, if any, otherwise `None`
        :return: the new element
        """
        return Element(self.__dest, name, attrs, False)

    def text(self, txt: str) -> None:
        """
        Write some textual content inside this XML element.

        :param txt: the text to be written.
        """
        self.__dest(txt.translate(_ESC))  # Write the text.
```

Listing 13.24: An example of using our simple context manager-based XML output API from Listings 13.22 and 13.23, where the output is printed to the stdout stream. (stored in file xml_user_print.py; output in Listing 13.25)

```
"""Use our simple XML output API to write XML data about this course."""

from xml_context import Element  # import our XML output API

with Element(print, "class", {  # attributes
        "title": "Programming with Python", "year": 2024}) as cls:
    with cls.element("description") as desc:  # first inner element
        desc.text("This is a class on Python.")  # text of inner element
    with cls.element("teacher", {"name": "Thomas"}) as teach:
        teach.text("I like Python.")  # Write text inside the element.
    with cls.element("students") as studis:
        with studis.element("student", {"name": "Bubba"}):
            pass  # This element does not have any text inside.
        with studis.element("student", {"name": "Bibba"}) as studi:
            studi.text("I like the <Programming with Python> book.")
```

↓ `python3 xml_user_print.py` ↓

Listing 13.25: The stdout of the program xml_user_print.py given in Listing 13.24.

```
<?xml version="1.0"?>
<class title='Programming with Python' year='2024'>
<description>
This is a class on Python.
</description>
<teacher name='Thomas'>
I like Python.
</teacher>
<students>
<student name='Bubba'>
</student>
<student name='Bibba'>
I like the &#38;Programming with Python&#62; book.
</student>
</students>
</class>
```

Listing 13.26: An example of using our simple context manager-based XML output API from Listings 13.22 and 13.23, where the output is written to a text file. (stored in file `xml_user_file.py`; output in Listing 13.27)

```
"""Use our simple XML output API to write XML data to a file."""

from os import remove  # The function for deleting the file at the end.

from xml_context import Element  # import our XML output API

# This time, we pass the `write` method of an output stream to the API.
with open("example.xml", mode="w", encoding="UTF-8") as stream_out:
    with Element(stream_out.write, "class", {  # attributes
            "title": "Programming with Python", "year": 2024}) as cls:
        with cls.element("description") as desc:  # first inner element
            desc.text("This is a class on Python.")
        with cls.element("teacher", {"name": "Thomas"}) as teach:
            teach.text("I like Python.")  # Write text inside the element.
        with cls.element("students") as studis:
            with studis.element("student", {"name": "Bubba"}):
                pass  # This element does not have any text inside.
            with studis.element("student", {"name": "Bibba"}) as studi:
                studi.text("I like the <Programming with Python> book.")

# Now we open the file again and read and print its contents.
with open("example.xml", encoding="UTF-8") as stream_in:
    for line in stream_in:  # Iterate over the lines in the file.
        print(line.rstrip())  # Print the line (without trailing newline).

remove("example.xml")  # Finally, we delete the file.
```

↓ `python3 xml_user_file.py` ↓

Listing 13.27: The stdout of the program `xml_user_file.py` given in Listing 13.26.

```
<?xml version="1.0"?>
<class title='Programming with Python' year='2024'><description>This is a
    ↪ class on Python.</description><teacher name='Thomas'>I like Python.</
    ↪ teacher><students><student name='Bubba'></student><student name='
    ↪ Bibba'>I like the &#38;Programming with Python&#62; book.</student></
    ↪ students></class>
```

13.6 Overview over Dunder Methods

Besides the examples mentioned above, there are many more duner methods in Python [273]. We can only provide an abridged overview in Figures 13.3 to 13.5.

From Figure 13.3 we learn that there are three string representation dunder functions. We already discussed `__str__` and `__repr__` in Section 13.1. The former provides a brief and human-readable string representation of an object, suitable for end users. The latter, `__repr__`, is instead intended for fellow programmers and debugging purposes. The third function, `__format__`, allows us to define special formats that can be used, for example, in f-strings.

We can allow our objects to be converted to simple types by specifying dunder methods. Technical, `__str__` and `__repr__` are type conversions to `str`. We can also provide `__int__`, `__float__`, and `__bool__` functions that should return, well, instances of `int`, `float`, or `bool`, if our objects can be represented as such. The `__bool__` function is special here: It is used in conditional expressions, such as `while` loops or in `if` statements. For the common collection classes, it is implemented such that `__bool__` of an empty collection yields `False` and a non-empty one returns `True`. `NoneType` implements it to return `False`, so `None.__bool__()`, which is the same as `bool(None)`, yields `False`.

- `__`
  - String Representation, see Section 13.1
    - `__str__` ..... returns a concise string representation for users; `str(a)` ≅ `a.__str__()`
    - `__repr__` .. string with all information on the object for programmers; `repr(a)` ≅ `a.__repr__()`
    - `__format__` ..... used in f-strings: `f"{a:s}"` ≅ `a.__format__(s)`
  - Type Conversion
    - `__bool__` ..... convert to `bool`: `bool(a)` ≅ `a.__bool__()`; used in conditions, e.g, in `if`
    - `__int__` ..... convert to `int`: `int(a)` ≅ `a.__int__()` [646]
    - `__float__` ..... convert to `float`: `float(a)` ≅ `a.__float__()` [646]
    - `__complex__` ..... convert to `complex`: `complex(a)` ≅ `a.__complex__()`
    - `__bytes__` ..... convert to `bytes`: `bytes(a)` ≅ `a.__bytes__()`
  - Hashing, see Section 13.2
    - `__hash__` ..... compute an integer value representing this object; `hash(a)` ≅ `a.__hash__()`
  - Ordering and Equality, see Section 13.3
    - `__eq__` ..... equality: `a == b` ≅ `a.__eq__(b)`
    - `__ne__` ..... inequality: `a != b` ≅ `a.__ne__(b)`
    - `__lt__` ..... less than: `a < b` ≅ `a.__lt__(b)`
    - `__le__` ..... less than or equal: `a <= b` ≅ `a.__le__(b)`
    - `__gt__` ..... greater than: `a > b` ≅ `a.__gt__(b)`
    - `__ge__` ..... greater than or equal: `a >= b` ≅ `a.__ge__(b)`
  - Context Managers, see Section 13.5
    - `__enter__` ..... enter a `with a as x` statement, returns the value `x` given to `as`
    - `__exit__` leave a `with` statement, receive exception information, returns `False` to re-raise caught exceptions, if any
  - Collections
    - `__len__` ..... get the number of elements in container; `len(a)` ≅ `a.__len__()`
    - `__contains__` ..... check whether element is present; `x in a` ≅ `a.__contains__(x)`
    - `__getitem__` ..... get element at index/key; `a[x]` ≅ `a.__getitem__(x)`
    - `__setitem__` ..... set element at index/key; `a[x]=y` ≅ `a.__setitem__(x, y)`
    - `__delitem__` ..... delete element at index/key; `del a[x]` ≅ `a.__detitem__(x)`
  - Iteration, see Section 10.1
    - `__iter__` ..... get iterator over elements; `iter(a)` ≅ `a.__iter__()`
    - `__next__` ..... get next element from iterator; `next(i)` ≅ `i.__next__()`
    - `__reversed__` ..... iterate backwards; `reverse(a)` ≅ `a.__reversed__()`
  - Classes
    - `__new__` . create a new instance of a class: `cls(a, b=c)` ≅ `cls.__new__(cls, a, b=c)`, is then followed by `cls.__init__(a, b=c)`
    - `__init__` ..... initialize an object: `cls(a, b=c)` ≅ `cls.__init__(a, b=c)`
    - `__call__` ..... call an object like a function: `a(b, c=d)` ≅ `a.__call__(b, c=d)`

Figure 13.3: An overview of the dunder methods in Python (Part 1).

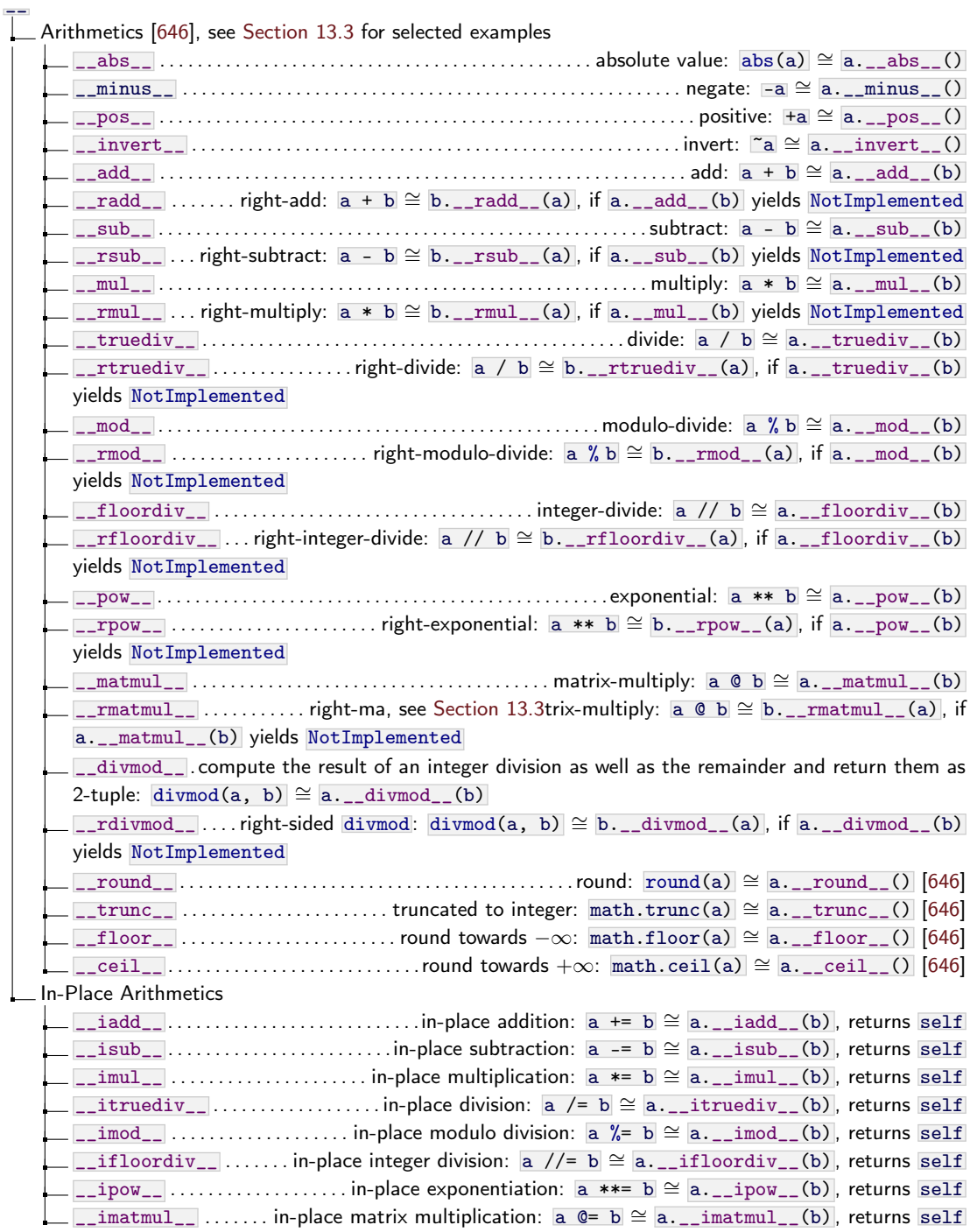


Figure 13.4: An overview of the dunder methods in Python (Part 2).

Numeric types usually implement it to return `False` if the number is zero and `True` otherwise. Besides Boolean, strings, and the basic numeric types, conversion functions can also be implemented for the numeric type `complex` for complex numbers as well as for arrays of `bytes`, both of which we will not discuss here.

We already discussed the comparison / total ordering dunder methods `__eq__`, `__ne__`, `__lt__`, `__le__`, `__gt__`, and `__ge__` in-depth in Section 13.3. Similarly, we provided a comprehensive example on context managers and the corresponding special methods in Section 13.5.

Much of the syntax of Python is governed via dunder methods. This also holds for convenient shorthands, like the `[...]` bracket access to elements or the `in` and `for` keywords. Back in Chapter 5, we learned about the various basic collection classes that Python provides, ranging from dictionaries

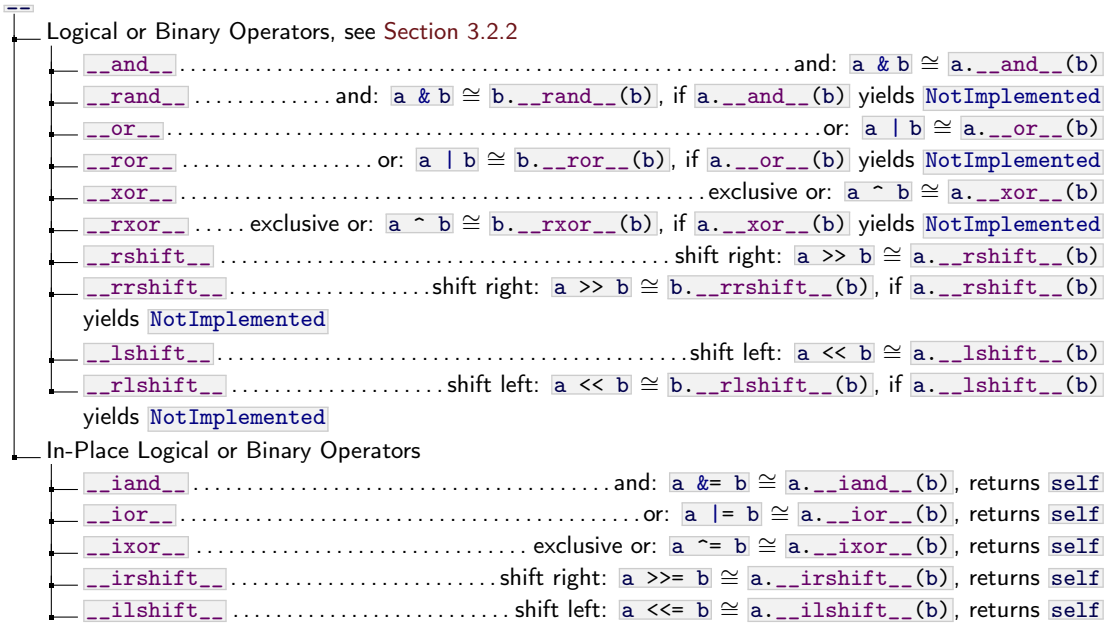


Figure 13.5: An overview of the dunder methods in Python (Part 3).

over sets to lists. Maybe we want to implement our own collection? It maybe could be a tree data structure, for example. In this case, we would first like to support the `len` function, which returns the number of elements stored in the collection, by implementing `__len__`. We maybe also want to support the `in` keyword, allowing us to check whether an element `e` is in a collection `c` by writing `e in c`. This can be done by implementing `__contains__`.

The convenient way to access elements of lists and dictionaries via the `[...]` bracket notation can be implemented as well. For reading, the `__getitem__` method would receive as parameter the index or key and should return the corresponding element. For writing, the `__setitem__` method receives both the index/key and the element to be stored at that index/key as parameter. Additionally, `__delitem__` can be implemented to allow removing the element corresponding to a given key or index.

An important feature of Python is the ability to iterate over sequences, which we discussed in Chapter 10. The `iter` function applied to a collection `c` returns an `Iterator`. It does so by calling `c.__iter__()`. By implementing this function, we turn our collections into instances of `Iterable` and can use our collection in `for` loops. The `Iterator` object returned by `__iter__` then must implement the `__next__` method, which is invoked by `next`. Finally, the `__reversed__` dunder method should return an `Iterator` iterating over the collection `c` *backwards*, which would be returned by `reversed(c)`.

While we implemented several arithmetic dunder methods for our `Fractions` class in Section 13.3, Figure 13.4 shows us that there are *many* more. The unary methods `__abs__`, `__minus__`, and `__pos__` allow us to implement the computation of the absolute value and the negated variant of a number, as well as the rather obscure unary plus. There are dunder methods for basically all binary arithmetic operators, ranging from `+`, `-`, `*`, `/`, `//`, `%`, to `**` and even the matrix multiplication operator `@`. Interestingly, these methods always exist in two variants, the plain one and one prefixed with "`r`." For example, when evaluating the expression `a + b`, Python will look whether `a` implements `__add__`. If so, it will call `a.__add__(b)`. If the return value of that method is different from `NotImplemented`, then this becomes the result of `a + b`. If either `__add__` is not implemented by `a` or if it returns `NotImplemented`, then the Python interpreter will check whether `b` implements `__radd__`. If so, then it will invoke `b.__radd__(a)`. If this method does not return `NotImplemented`, then its return value becomes the result of `a + b`. If it does return `NotImplemented` or if `b` does not implement `__radd__`, then a `TypeError` is raised. The same schematic exists for the other operators as well.

This allows us to develop new mathematical types independent from existing ones and support mixing them in arithmetic expressions with existing types. For example, the existing datatype `int` certainly would not have `__add__` implemented in a way that can handle our `Fraction` class. How could it? Nobody could have guessed that we would create our own class for rational numbers. However, we could have implemented `__radd__` in a way that handles `int`s. In that case, doing something

like `5 + Fraction(3, 5)` would work.

Besides these mathematical operators for useage in expressions, there are also in-place variants. They are usually prefixed with "`i`." For example, `__iadd__` corresponds to the `+=` operator. Doing `a += b` would invoke `a.__iadd__(b)`, which then should update the value of `a` to be the sum of `a` and `b`.

Finally, Figure 13.5 lists the dunder methods for operators such as `&`, `|`, `^`, `>>`, and `<<`, which we learned about when we first got in touch with integer numbers in Section 3.2.2. These dunder methods follow the same schematic as those for arithmetics and also have "`r`" and "`i`" variants.

Even more dunder methods are listed in the comprehensive overview [273].

13.7 Summary

Much of Python's syntax is implemented by the special methods, also called magical methods, or *dunder* methods. The *dunder* stands for *double underscore*, because the names of such methods begins and ends with `__`.

Knowing about dunder methods allows us to create classes which can seamlessly be used in arithmetic expressions, in `with` statements, as sequences to iterate over with `for` loops, that support indexing with `[...]`. By implementing dunder methods, we can use the Python syntax to construct new collections, support more complex mathematical structures, ensure that resources are properly managed (and eventually disposed), or create elegant and concise APIs.

Part IV

Working with the Ecosystem

A software developer rarely works on a stand-alone project all by themself. Instead, they develop projects that are part of an ecosystem of applications. Their programs will usually depend on libraries, i.e., Python packages that offer functionality. Often, their projects are stored in VCSes like Git repositories. In this part of the book, we will take a small glimpse on how to work within a system of existing projects and VCSes.

Chapter 14

Using Packages

As already mentioned very early on in this book, one important strength of Python is the wide range of available packages. A package in Python is a piece of software, a library, that bundles some functionality and that can be installed on a system to make that functionality usable. Many of these packages are open source software and they are available for anyone to use, free of charge. The number one source for such packages is the Python Package Index [559], a website from which they can be downloaded and installed, illustrated in Figure 14.1. In this section of the book, we will focus on how we can obtain and use packages.

14.1 pip and Virtual Environments

The go-to tool for installing Python packages is `pip` [286, 448].

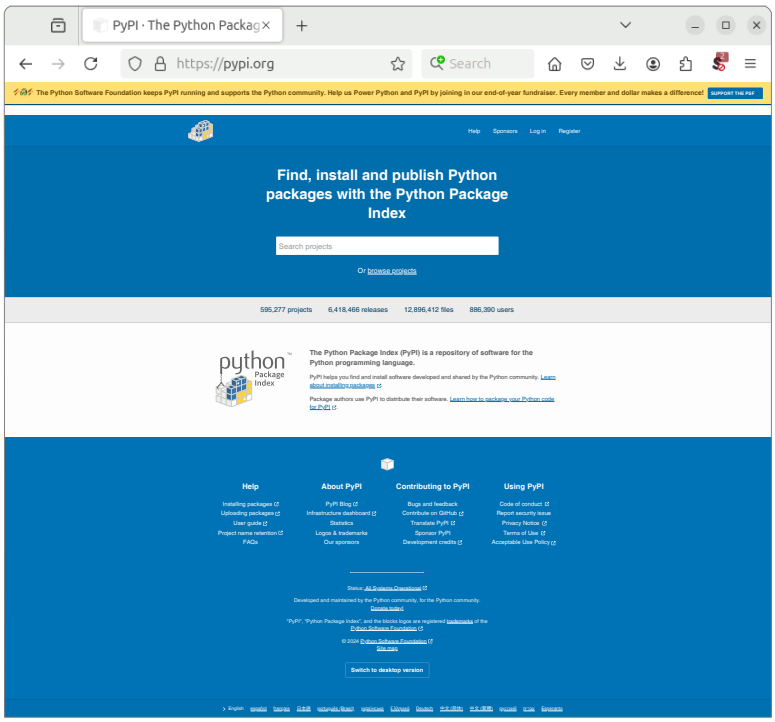


Figure 14.1: A screenshot of the website `https://pypi.org`, the Python Package Index [559], taken on 2024-12-24.

Listing 14.1: A small Python script using the NumPy library. (src)

```
"""An example of using numpy."""

import numpy as np

print(f"look, a numpy array: {np.array([1, 2, 3.0])}")
```

Useful Tool 11

pip is a software that can be used to install packages under Python.

- With the command `pip install thePackage`, the package `thePackage` is installed.
- With the command `pip install "thePackage==version"`, the version `version` of the package `thePackage` is installed.
- With the command `pip install -r requirements.txt`, the packages listed in the requirements file `requirements.txt` are installed. A requirements file allows to put multiple package/version dependencies that would otherwise command line arguments of pip into a single file [449].

pip should always be used in a virtual environment, see Best Practice 73 and 74.

The standard way to install packages for use with Python is in a so-called virtual environment. A virtual environment is a directory with an isolated Python installation and package directory [394, 605]. Multiple separate virtual environments can exist on a computer, all with their own set of installed packages. This allows you to have different Python setups for different applications by installing them into different virtual environments. Virtual environments are particularly useful if different applications need different versions of the same packages. This way, version clashes are avoided. They also give you better understanding about the actual dependencies of your applications: Installing the packages directly needed by an application into a new virtual environment will also install their dependencies and the dependencies of their dependencies, and so on.

Best Practice 73

Packages should *always* be installed in virtual environments and never system-wide (maybe with the exception of pip and venv).

Best Practice 74

The command `pip install` should always be used with the option `--require-virtualenv`, e.g., `pip install --require-virtualenv thePackage`. This enforces that pip is really executed in a virtual environment and will cause an error otherwise.

Useful Tool 12

The module venv is used for creating and managing virtual environments under Python.

For demonstration purposes, let us assume that we have written a program using the package NumPy [152, 252, 299]. The small program in Listing 14.1 only creates and prints a NumPy array, but for this, obviously, NumPy is needed. NumPy is not available in fresh Python installations and needs to be installed so that we can run our program. This will be the example that we will use to demonstrate the use of virtual environments in the following sections.

As prerequisites to install packages in virtual environments, we need to make sure that both pip and venv are installed on our system. The procedures for both differ under Linux and Microsoft Windows. Using pip and venv is often done in the terminal, i.e., by typing commands or executing shell scripts. This, naturally, too works differently under Linux and Microsoft Windows. We therefore will briefly explore how to achieve these things under both operating systems.

14.1.1 pip and Virtual Environments in Ubuntu Linux

First, we need to make sure that both pip and venv are installed. On some systems, they compare pre-installed with Python 3, which itself comes pre-installed. On others, at least venv needs to be installed. These installations need to be managed by the system via apt-get and not pip, because Python is used for several different things under Ubuntu Linux. Only the Linux package manager can make sure that not inconsistencies arise. We install the packages `python3-pip` and `python3-venv` using apt-get. This requires superuser privileges, i.e., sudo, so we write `sudo apt-get install python3-pip python3-venv`. After entering the password, both packages are installed (if they are not already installed). This process is illustrated in Figure 14.2. After apt-get completes, we can check the version of pip via `pip --version`. Whether venv is installed correctly can be checked via `python3 -m venv -h`.

Listing 14.2 presents a self-contained example where we execute the necessary commands to setup a virtual environment, install the needed package, run our program, and tear down the virtual environment. In the real world, you would set up the environment once and could use it many times to run your program.

Assume that we have already opened a terminal by pressing Ctrl + Alt + T and that we have entered the directory where our program `numpy_user.py` from Listing 14.1 is located. In Figure 14.3, we step-by-step execute the commands from Listing 14.2 and show their output. We begin by creating a new empty directory named `.venv` in our current directory by `mkdir -p .venv` command followed by ↵ (Figure 14.3.1). Inside this directory, we set up the new and empty virtual environment by typing `python3 -m venv .venv`, again followed by ↵ (Figure 14.3.2). The directory `.venv` now should contain a Python interpreter as well as all files needed for managing the environment.

The next step is to activate the environment. If we would type `python3` right now, we would still be using the system's Python interpreter and the packages installed system-wide. However, we actually want to use the virtual environment instead. For this, we need to execute *all* the commands in the file `.venv/bin/activate`, which was automatically created for us when we set up the virtual

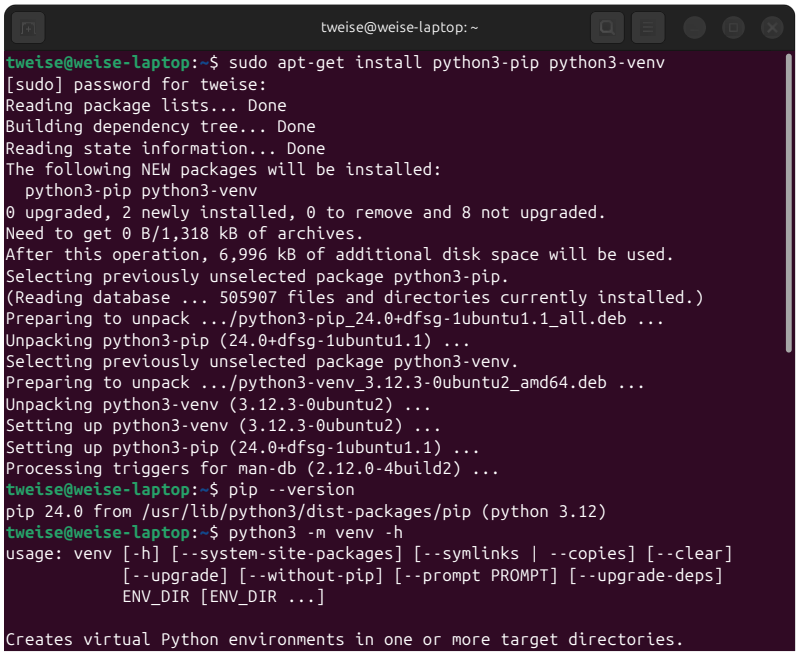


Figure 14.2: Installing pip and venv under Ubuntu Linux: pip is usually already installed, venv maybe or maybe not. We need to use the `apt-get` route to make sure that both `python3-pip` and `python3-venv` are installed.

environment. The simplest way to do this is to just copy all of them into the current terminal as if we had written by hand. This happens if we type `source .venv/bin/activate`, confirmed by ↵, as shown in Figure 14.3.3. If you are following this example on your own computer, then after executing this command, the Bash prompt (the little text on the left-hand side) changes, showing that now the virtual environment is active. This is also visible in Figure 14.3.3, where the prefix `(.venv)` is added to the prompt.

This means that any call to the Python interpreter will, from now on, use the interpreter stored in the virtual environment. We can also only use packages that were installed there. If we execute `pip install --require-virtualenv numpy`, this will install the NumPy package. But it uses the Python interpreter and package directory of the activated virtual environment. So in Figure 14.3.4, NumPy is not installed system-wide, but into the virtual environment.

We can now execute our small program `numpy_user.py` in Figure 14.3.5. We type `python3 numpy_user.py` and hit ↵. Indeed, the expected output `look, a numpy array: [1. 2. 3.]` appears in the terminal, as also already shown in Listing 14.3. Our program uses the local NumPy installation inside the virtual environment.

We are now finished with our application. To clean up, we deactivate the virtual environment by executing `deactivate` in Figure 14.3.6. This causes the prompt to change back to normal. Any invocation of Python would now use the system installation. It would no longer have access to our virtual environment and the packages installed within.

If this was a more complex and useful application, then this would be the steps to get it ready and usable: We create the virtual environment exactly once. Exactly once we need to install all the required packages into it. Whenever we want to run our application, we would open a new terminal, we enter our directory, activate the virtual environment, and then just run the program. After that, we would deactivate the environment. Deactivating the environment does not delete anything. All of our settings and installed packages are still there. The next time we activate the environment again, we can use them again, without the need to re-install them.

Anyway, we did deactivate the environment just now. To confirm that we really installed NumPy only locally and that our program was really using the package installed in the virtual environment, we try to run our program again *after* deactivating the environment in Figure 14.3.7. As you can see in Figure 14.3.7 and Listing 14.3, this second execution fails: A `ModuleNotFoundError` is raised and the interpreter terminates.

Finally, in our example in Listing 14.2 and Figure 14.3.8, we delete the virtual environment directory again via `rm -rf .venv`. Normally, you would *not* do this.[1] You do not want to re-create the virtual environment again every time you execute your application. As said, once you have installed the required packages, you can simply activate the environment and run your program whenever you want.

[1] I just clean up here because my examples are automatically executed whenever the book is built and I want to avoid that the examples interfer with each other or that, accidentally, some files from the environment land in my Git repositories.

Listing 14.2: An example of using virtual environments and `pip` under `Ubuntu Linux` to install `NumPy` and to run our program Listing 14.1.

```
echo "# We create the directory '.venv' for the virtual environment."
mkdir -p .venv

echo "# We create the (empty) virtual environment inside the directory."
python3 -m venv .venv

echo "# After creating the virtual environment, we activate it."
echo "# Any Python program now uses the activated virtual environment."
source .venv/bin/activate

echo "# We install the package 'numpy' into the virtual environment."
pip install --require-virtualenv --progress-bar off numpy

echo "# 'numpy' is now available for Python programs."
echo "$ python3 numpy_user.py"
python3 numpy_user.py 2>&1

echo "# We deactivate the virtual environment."
echo "# This means that programs now use the system environment only."
deactivate

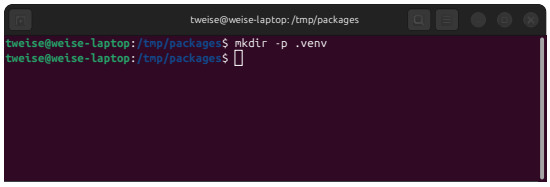
echo "# 'numpy' is no longer available (unless installed system-wide)."
echo "$ python3 numpy_user.py"
python3 numpy_user.py 2>&1 || true

echo "# We could re-use the virtual environment by activating it again."
echo "# However, we delete the directory to clean up after the example."
rm -rf .venv
```

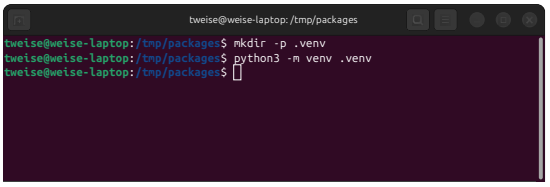
↓ `bash numpy_user_venv.sh` ↓

Listing 14.3: The stdout of the program `numpy_user.py` given in Listing 14.1.

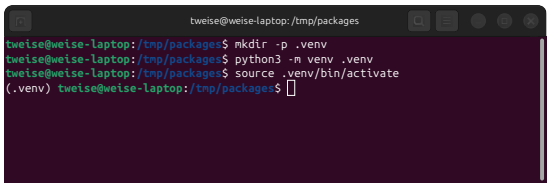
```
# We create the directory '.venv' for the virtual environment.
# We create the (empty) virtual environment inside the directory.
# After creating the virtual environment, we activate it.
# Any Python program now uses the activated virtual environment.
# We install the package 'numpy' into the virtual environment.
Collecting numpy
  Downloading numpy-2.5.4-cp314-cp314-musllinux_1_2_x86_64.whl.metadata
    ↪ (6.6 kB)
Downloading numpy-2.5.4-cp314-cp314-musllinux_1_2_x86_64.whl (18.5 MB)
Installing collected packages: numpy
Successfully installed numpy-2.5.4
# 'numpy' is now available for Python programs.
$ python3 numpy_user.py
look, a numpy array: [1. 2. 3.]
# We deactivate the virtual environment.
# This means that programs now use the system environment only.
# 'numpy' is no longer available (unless installed system-wide).
$ python3 numpy_user.py
Traceback (most recent call last):
  File "{...}/packages/numpy_user.py", line 3, in <module>
    import numpy as np
ModuleNotFoundError: No module named 'numpy'
# We could re-use the virtual environment by activating it again.
# However, we delete the directory to clean up after the example.
```



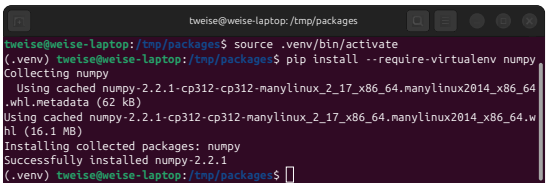
(14.3.1) Create the directory for the virtual environment by typing `mkdir -p .venv` into the terminal and hitting ↵.



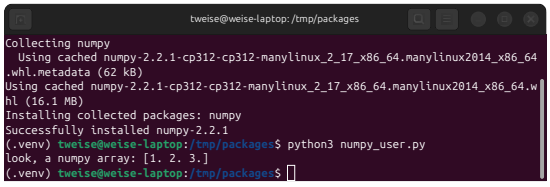
(14.3.2) Set up the virtual environment in directory `.venv` by typing `python3 -m venv .venv` and hitting ↵.



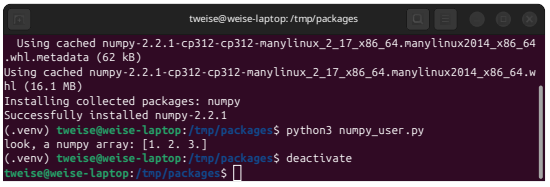
(14.3.3) We activate the virtual environment by `source .venv/bin/activate` + ↵. Notice that the prompt changes: It now has the prefix `(.venv)`.



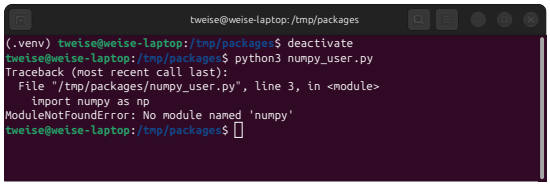
(14.3.4) We install the `NumPy` package into the activated virtual environment vis `pip install --require-virtualenv numpy` + ↵.



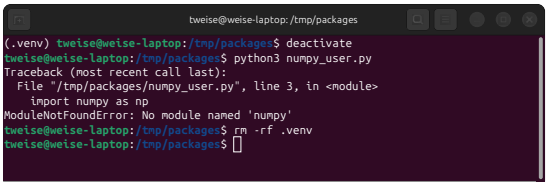
(14.3.5) Now we can execute the `Python` program `numpy_user.py` from Listing 14.1 via `python3 numpy_user.py` + ↵.



(14.3.6) We deactivate the virtual environment by calling `deactivate` and hitting ↵. We could activate it again at any point in time in the same way shown in Figure 14.3.3.



(14.3.7) Trying to execute `numpy_user.py` will now fail, because `NumPy` is only installed in the virtual environment, which is not active now.



(14.3.8) Finally, we delete the directory `.venv` via `rm -rf .venv` + ↵. Normally, you would retain this directory and use it again next time you want to execute our program.

Figure 14.3: A step-by-step execution of the commands in Listing 14.2 in the `Ubuntu Linux` terminal, which was opened by hitting Ctrl+Alt+T.

14.1.2 pip and Virtual Environments under Microsoft Windows

Like under Linux, we also need to make sure that pip and venv are installed before we can actually use either of them. Luckily, as shown in Figure 14.4, they already came pre-installed with my Python distribution.

We now want to execute our small program Listing 14.1, which uses NumPy. NumPy does not come pre-installed. So we need to install it first. Following general best practices, we will do so using a virtual environment. All the necessary commands for this are listed in the Microsoft Windows batch file in Listing 14.4. The complete corresponding output in the terminal is given in Figure 14.5.

Here, we will work our way through this step-by-step in Figure 14.6. First, you need to press ⊞ + R, type in `cmd`, and hit ↵ to open a Microsoft Windows terminal. We enter the directory where our program file `numpy_user.py` is located with the `cd` commend (not illustrated).

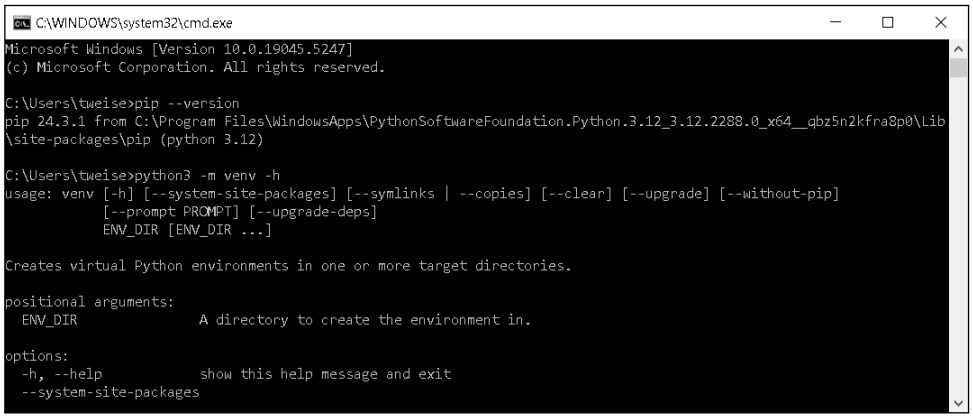


Figure 14.4: Installing pip and venv under Microsoft Windows: They are already installed.

Listing 14.4: An example of using virtual environments and pip under Microsoft Windows to install NumPy and to run our program Listing 14.1. The output is given in Figures 14.5 and 14.6 (src)

```
echo # We create the directory '.venv' for the virtual environment.
md .venv

echo # We create the (empty) virtual environment inside the directory.
python -m venv .venv

echo # After creating the virtual environment, we activate it.
echo # Any Python program now uses the activated virtual environment.
call .venv\Scripts\activate.bat

echo # We install the package 'numpy' into the virtual environment.
pip install --require-virtualenv --progress-bar off numpy

echo # 'numpy' is now available for Python programs.
echo $ python numpy_user.py
python numpy_user.py 2>&1

echo # We deactivate the virtual environment.
echo # This means that programs now use the system environment only.
call deactivate

echo # 'numpy' is no longer available (unless installed system-wide).
echo python numpy_user.py
python numpy_user.py 2>&1

echo # We could re-use the virtual environment by activating it again.
echo # However, we delete the directory to clean up after the example.
rd /S /Q .venv
```



Figure 14.5: The output of Listing 14.4 when executed in a Microsoft Windows terminal. To open the terminal, press ⊞+R, type in `cmd`, and hit ↵.

As a first step, we need to create a directory `.venv` to host the virtual environment. We therefore write `md .venv` and hit ↵ in Figure 14.6.1. To then set up a new and, initially, empty virtual environment in this directory, we type `python -m venv .venv` and hit ↵ (see Figure 14.6.2). Notice that under Ubuntu Linux, we always used the `python3` command, but here we always use `python` instead. Either way, executing the command has filled the directory `.venv` with the necessary files and scripts for an isolated Python environment.

In Figure 14.6.3, we activate this environment by running `.venv\Scripts\activate.bat` (and hitting ↵). This Microsoft Windows batch file has been created for us when we set up the virtual environment. It activates the environment, which leads to a change in the prompt, i.e., the little text that always appears left of where we type the input. As can be seen in Figure 14.6.3, the text `(.venv)` has been pre-pended to the prompt, which tells us that this is the currently active virtual environment.

We now want to install the NumPy package into this virtual environment. We can do this by typing `pip install --require-virtualenv numpy` and then pressing ↵. This causes NumPy to be downloaded (or copied from the internal cache) and installed, as shown in Figure 14.6.4.

We can now run our program `numpy_user.py` by typing `python numpy_user.py` and hitting ↵. Indeed, as you can see in Figure 14.6.5, the program runs without error and prints `look, a numpy array: [1. 2. 3.]`, exactly as expected. It found the NumPy package that was installed in the virtual environment.

Now that we have finished executing our program, we can deactivate the virtual environment. In Figure 14.6.6, we type `deactivate` and press ↵. This causes the prompt to change back to normal. The virtual environment is no longer active. This means that from now on, all interaction takes place



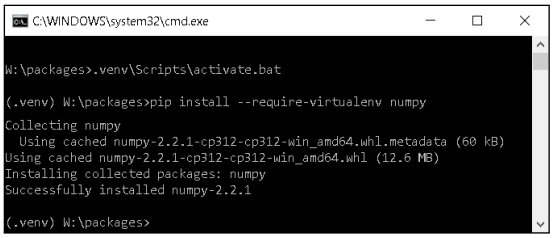
(14.6.1) Create the directory for the virtual environment by typing `md .venv` into the terminal and hitting ↵.



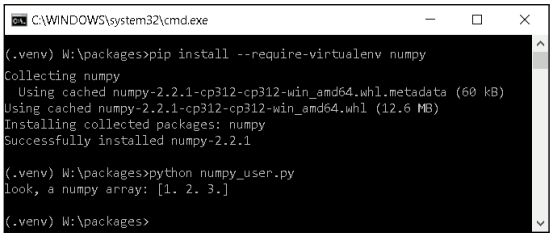
(14.6.2) Set up the virtual environment in directory `.venv` by typing `python -m venv .venv` and hitting ↵.



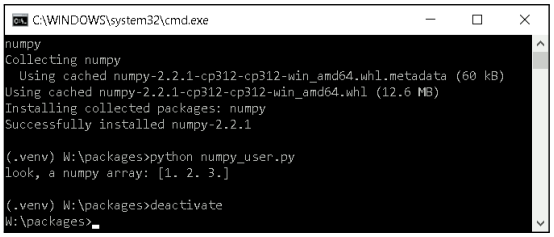
(14.6.3) We activate the virtual environment by `.venv\Scripts\activate.bat` + ↵. Notice that the prompt changes: It now has the prefix `(.venv)`.



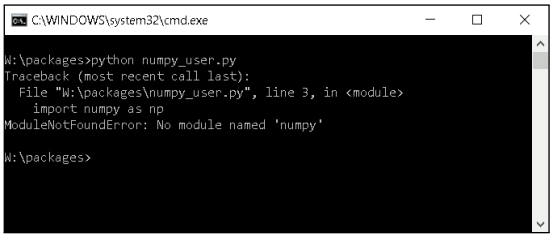
(14.6.4) We install the `NumPy` package into the activated virtual environment vis `pip install --require-virtualenv numpy` + ↵.



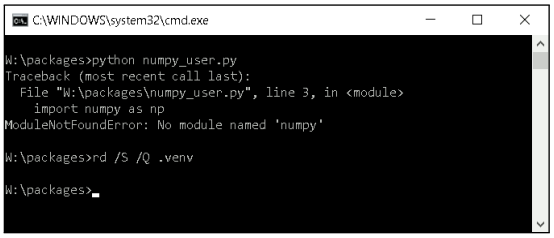
(14.6.5) Now we can execute the `Python` program `numpy_user.py` from Listing 14.1 via `python numpy_user.py` + ↵.



(14.6.6) We deactivate the virtual environment by calling `deactivate` and hitting ↵. We could activate it again at any point in time in the same way shown in Figure 14.6.3.



(14.6.7) Trying to execute `numpy_user.py` will now fail, because `NumPy` is only installed in the virtual environment, which is not active now.



(14.6.8) Finally, we delete the directory `.venv` via `rd /S /Q .venv` + ↵. Normally, you would retain this directory and use it again next time you want to execute our program.

Figure 14.6: A step-by-step execution of the commands in Listing 14.4 in the Microsoft `Windows` terminal. To open the terminal, press ⊞+R, type in `cmd`, and hit ↵.

with the system Python installation. If our program was a real, valuable application, then the next time we would like to use it, we would simply activate the virtual environment again. All of our files, settings, and installed packages are still there. Nothing has been deleted (yet). If we would activate the virtual environment again, we could just run the program again, we would not need to re-installed any package.

However, we now want to verify that deactivating the virtual environment really means that right now, we are just working with the system Python setup. For this purpose, we try to run our program again Figure 14.6.7 by again writing `python numpy_user.py` and hitting ↵. As you can see, this does not work. NumPy was only installed in the virtual environment and is not available system-wide. The first line of our program, which tries to `import numpy`, therefore raises a `ModuleNotFoundError`.

At the very end of this example, I want to clean up my folder by typing `rd /S /Q .venv` and hitting ↵. This deletes the directory `.venv` and everything within. You would normally not do this, because normally we want to re-use our virtual environments by activating and deactivating them as needed. Either way, this was a complete example of using virtual environments, pip, and venv under Microsoft Windows. It was actually not very much different from the Ubuntu Linux example in the previous section.

14.2 Requirements Files

It is very normal that our projects require multiple different packages. Sometimes, they require specific versions of specific packages. Instead of manually installing these dependencies with pip every time we want to use our program, it makes sense to *write them down*. Indeed, the dependencies of an application are a very important part of the documentation.

Requirements files offer a very simple format for this purpose. They are usually called `requirements.txt` and reside in the root folder of a project. As their name implies, they are simple text files. Each of their lines lists one package that is required, optionally with version constraints.

Listing 14.5 gives a trivial example of such a file. It contains the line `numpy==2.5.2`. This means that NumPy of exactly the version 2.5.2 is required for our application. Had we written `numpy>=2.5.2`, then a larger version of NumPy would also have been OK. The other operators, `>`, `<`, and `<=` can be used as well and have their natural corresponding meaning. If we had wanted, we could have written `matplotlib` in the second line of our `requirements.txt` file, which would then have meant that, besides NumPy of version `2.5.2`, the package Matplotlib is needed as well. Writing only `matplotlib` would be interpreted as "find the highest version of Matplotlib that is compatible with NumPy version 2.5.2." But we only need one package, NumPy, so our file just has a single line.

In Listing 14.6, we present a version of the script that we used to set up a virtual environment and run our two-line NumPy program under Ubuntu Linux (see Listing 14.6.) The only difference is that we now do not write `pip install numpy` but instead write `pip install -r requirements.txt`. This tells pip to install *all* the requirements from the requirements file `requirements.txt`. Under Microsoft Windows, it works exactly the same.

Having a `requirements.txt` file in the root directory of your project is very useful. It automatically informs anybody who works with your code or who uses your program which versions of which packages they need. Therefore, it makes sense to specify package versions as precisely as possible in `requirements.txt`.

Best Practice 75

The packages that a Python project requires or depends on should be listed in a file called `requirements.txt` in the root folder of the project.

Listing 14.5: A requirements file demanding that version `2.5.2` of NumPy be installed. (src)

```
1 numpy==2.5.2
```

Best Practice 76

All required packages in a `requirements.txt` file should be specified with the exact version, i.e., in the form of `package==version`. This ensures that the behavior of the project can be exactly replicated on other machines. It rules out that errors can be caused due to incompatible versions of dependencies, because it allows the users to exactly replicate the set up of the machine on which the project was developed.

Listing 14.6: A script that runs our example program using a virtual environment created by using the `requirements.txt` file given in Listing 14.5.

```
echo "# We create the directory '.venv' for the virtual environment."
mkdir -p .venv

echo "# We create the (empty) virtual environment inside the directory."
python3 -m venv .venv

echo "# After creating the virtual environment, we activate it."
echo "# Any Python program now uses the activated virtual environment."
source .venv/bin/activate

echo "# Install the packages listed in 'requirements.txt' in the venv."
pip install -q --require-virtualenv --progress-bar off -r requirements.txt

echo "# 'numpy' is now available for Python programs."
echo "$ python3 numpy_user.py"
python3 numpy_user.py 2>&1

echo "# We deactivate the virtual environment."
echo "# This means that programs now use the system environment only."
deactivate

echo "# 'numpy' is no longer available (unless installed system-wide)."
echo "$ python3 numpy_user.py"
python3 numpy_user.py 2>&1 || true

echo "# We could re-use the virtual environment by activating it again."
echo "# However, we delete the directory to clean up after the example."
rm -rf .venv
```

↓ bash ↓

Listing 14.7: The stdout of the program numpy_user_venv_req.sh given in Listing 14.6.

```
# We create the directory '.venv' for the virtual environment.
# We create the (empty) virtual environment inside the directory.
# After creating the virtual environment, we activate it.
# Any Python program now uses the activated virtual environment.
# Install the packages listed in 'requirements.txt' in the venv.
# 'numpy' is now available for Python programs.
$ python3 numpy_user.py
look, a numpy array: [1. 2. 3.]
# We deactivate the virtual environment.
# This means that programs now use the system environment only.
# 'numpy' is no longer available (unless installed system-wide).
$ python3 numpy_user.py
Traceback (most recent call last):
  File "{...}/packages/numpy_user.py", line 3, in <module>
    import numpy as np
ModuleNotFoundError: No module named 'numpy'
# We could re-use the virtual environment by activating it again.
# However, we delete the directory to clean up after the example.
```

14.3 Virtual Environments in PyCharm

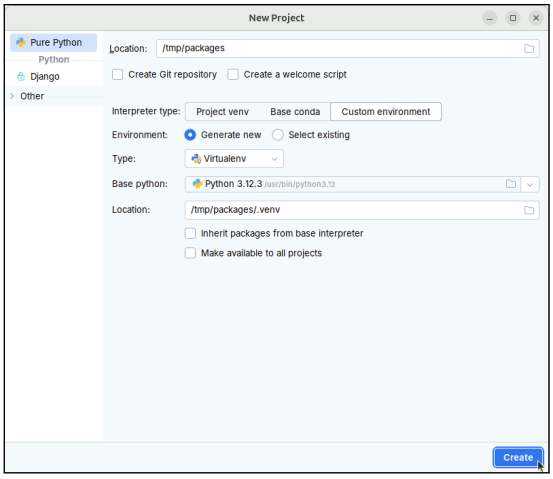
The IDE PyCharm also supports using virtual environments and `requirements.txt` files. Let us here demonstrate this on the example program that requires NumPy which was given as Listing 14.1 and `requirements.txt` file given in Listing 14.5. In Figure 14.7, we work through the steps to create and manage a project with a virtual environment in PyCharm on their basis. (A complete example on how to copy a source code repository from Git or GitHub and how to set up a virtual environment for its dependencies is given in Section 15.1.1.)

First, we need to create a new project (Figure 14.7.1). For this purpose, we already copied the files virtual environments and `requirements.txt` into a folder (here: `/tmp/packages`). In the New Project dialog of PyCharm, we select this folder as Location:. As Interpreter type:, we choose Custom Environment and as Environment:, we pick Generate new. As Type:, we choose Virtualenv. The default Location is `.venv` inside our project directory and we keep this setting. After clicking Create, a new project is created.

Well, almost: Figure 14.7.2 reminds me that I copied some files into the project folder before creating the project. PyCharm wants to make sure that this is right and asks me so. As answer, I click Create from Existing Sources.

We now are in the normal PyCharm project view in Figure 14.7.3. All the files that I placed into the folder are there and also a `.venv` directory has been created. Notice that you could as well create an empty project and copy the files there.

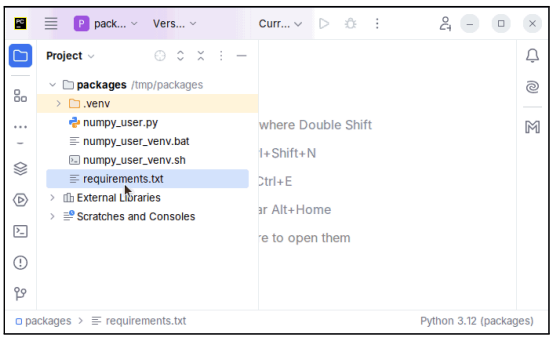
We click on `requirements.txt` and it opens in Figure 14.7.4. Indeed, the file prescribes a single requirement, NumPy in version 2.5.2.



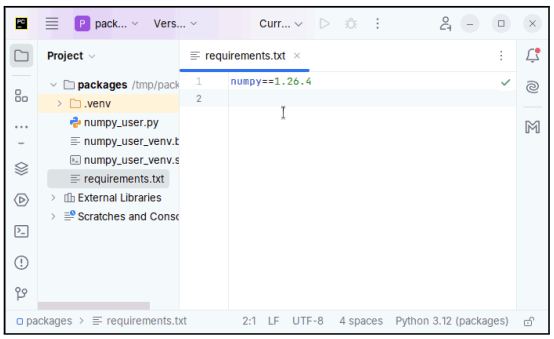
(14.7.1) We create a new project in PyCharm and select a virtual environment as Custom Environment. We click Create.



(14.7.2) Since I selected the folder where the other files (`requirements.txt`, `numpy_user.py`, ...) are already located, PyCharm asks whether this is OK. We select Create from Existing Sources if asked.

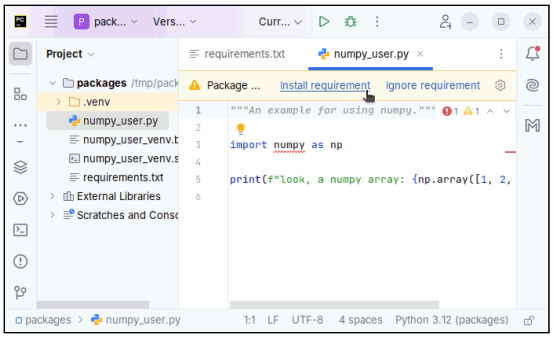


(14.7.3) Let's take a look at `requirements.txt`. We double-click on this file.

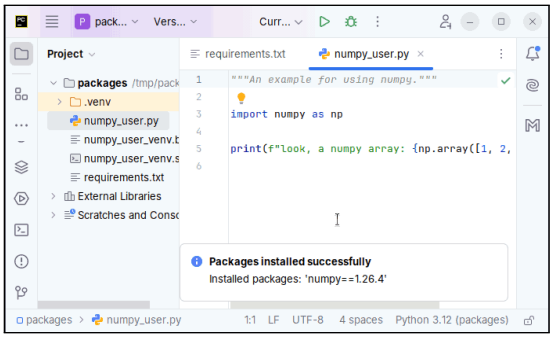


(14.7.4) It contains the one line `numpy==2.5.2`, which means that our project requires NumPy in version 2.5.2.

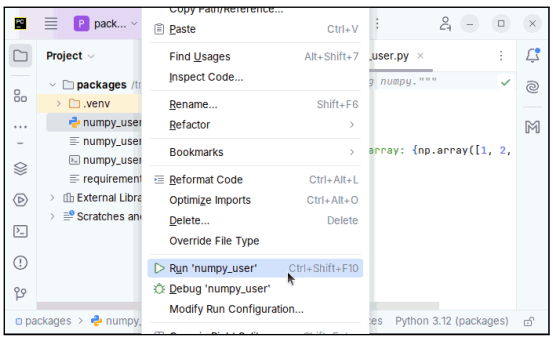
Figure 14.7: Using `requirements.txt` and virtual environments in PyCharm (part 1).



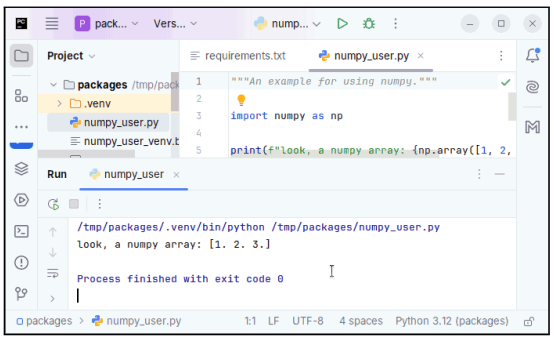
(14.7.5) We now click on `numpy_user.py` and get informed that required packages are missing. We can install them into our project's virtual environment by clicking Install requirement. Notice: This question could also have popped up when we opened `requirements.txt`, in which case we would have installed the requirements then.



(14.7.6) We get informed that the required packages were successfully installed.



(14.7.7) We right-click on `numpy_user.py` and click on Run 'numpy_user.py'.



(14.7.8) Our program is executed without error and produces the expected output.

Figure 14.7: Using `requirements.txt` and virtual environments in PyCharm (part 2).

When we click on the file `numpy_user.py` to open it in Figure 14.7.5, we notice a yellow bar at the top of the file's contents. This bar tells us that the required package NumPy is missing. It offers us the two choices to either Install requirement or to Ignore requirement. We select the first option and click it. Notice that this same yellow bar could also have appeared when we opened `requirements.txt`. I am not sure why it appears only now. Either way, we accept the choice to install the requirement.

In Figure 14.7.6, we are informed that the installation was successful. The small overlay also tells us that NumPy was installed in version 2.5.2, as prescribed by the `requirements.txt` file. At the time of this writing, NumPy is already out at version 2.2.1, which would have been used if we had just installed it without version specification. So this confirms that, indeed, our `requirements.txt` is used. We also can see that the red underline under `numpy` that was present in Figure 14.7.5 is now gone in Figure 14.7.6, because now the package and corresponding modules can be loaded without error.

As the final check that everything went well we run our program `numpy_user.py` in Figure 14.7.7. We right-click the file in three view on the left-hand side of the window. A popup menu appears, in which we click on Run 'numpy_user.py'. We could just as well have pressed Ctrl + ⇧ + F10. Our program is executed and the expected output appears (Figure 14.7.8). All is well.

However, it must be understood that the support for virtual environments and requirements files in PyCharm is for *software development*. If you want to actually use the software you have developed, you should *always* use the command line and terminals, as discussed in Section 14.1. PyCharm is not a runtime environment for the deployment of productive code. It is an Integrated Development Environment (IDE) for developing programs.

Best Practice 77

PyCharm must **never** be used for running an application in a productive setting. It is *only* to be used as IDE for software development. For actually executing programs, always use a virtual environments in the terminal as introduced in Section 14.1. See also Best Practice 1.

This holds also and especially for scenarios where we do use Python for scientific experiments. It is an even worse idea to try to run multiple concurrent instances of a program in a PyCharm window or to have multiple PyCharm instances open to run multiple applications in parallel.

Chapter 15

The Distributed Version Control System git

Today, Git [515, 577] is maybe the VCS with the most wide-spread use. It is the VCS on which GitHub [441, 478, 532] is based, which, in turn, is maybe the most important hub for open source software projects in the world. Git is based on a client-server architecture, where the server hosts and manages repositories of source code and other resources. The Git client is a command line application that is run in the terminal and which allows you to clone (i.e., download) source code repositories and upload (i.e., commit) changes to them. A repository is something like a directory with files and their editing history, i.e., you can work and improve source code, commit changes, and see the history of all past commits. This is what VCSes are for: They do not just provide the current state of a project and allow teams to cooperative and continuous develop software, they also store the history of the project so as to enable us to see which code was used in which version of our software and to track changes. How to correctly use Git thus is a very complex and involved topic beyond the scope of this book. However, we will here take at least a look into a very small subset of functions that provide you a starting point for working with Git repositories.

15.1 Cloning git Repositories

The most fundamental activity you will encounter is *cloning* repositories. Under Git, this basically means to download the complete repository and its history to your machine. You can now make local changes to the downloaded files. You can create commits, that will change the local version of your repository. Then you could push these changes back to the repository hosted by the Git server, making them accessible for other users. But, well, the first step is to clone - i.e., to download – the repository.

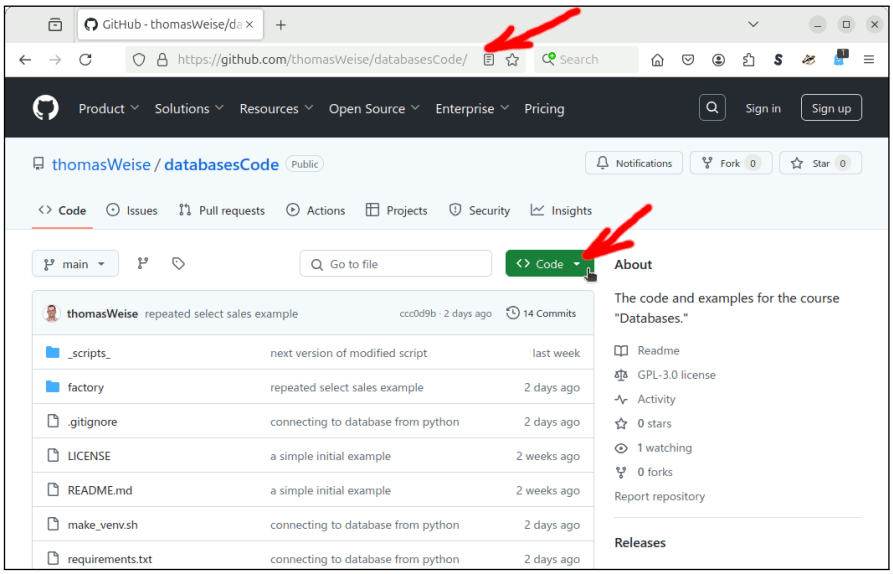
You can clone Git repositories with the command line `git` client program. However, PyCharm also has a Gitclient built in. We here outline both approaches based on the example of cloning a repository from GitHub [122].

15.1.1 Cloning git Repositories under Pycharm

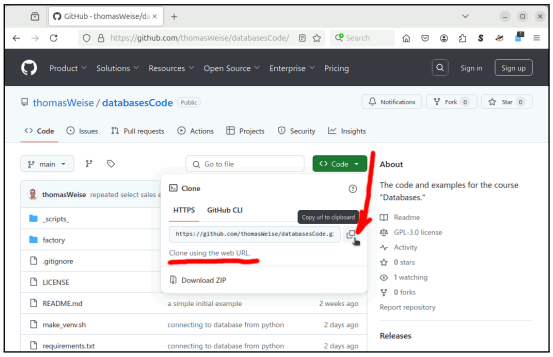
It is a very common scenario that we find an online repository with Python code that we are interested in. Many papers in deep learning, for example, publish their code in this way. Many such source code collections are Git repositories on GitHub. The example programs that ship with this book [615] are published like this at https://github.com/thomasWeise/programmingWithPythonCode, for example. We also have another book in progress, named *Databases* [611]. It, too, comes with a repository with sources for examples, this time at https://github.com/thomasWeise/databasesCode.

As usual, we work through a topic based on an example. This time, our goal is to *download and get to run code from a GitHub repository in PyCharm*. Matter of fact, we already exercised the whole process of cloning the GitHub repository in PyCharm with the examples of this book in Section 2.5. To complement the excursion from back then, we this time pick the companion code of our *Databases* book [611] at https://github.com/thomasWeise/databasesCode as an example. This repository comes with a file `requirements.txt`, which allows us to present the workflow of cloning a Git repository with setting up a virtual environment and installing required packages into one single example.

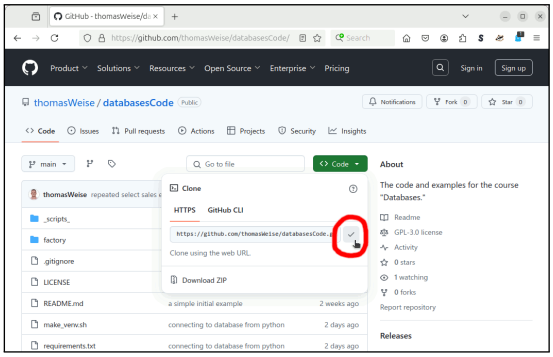
Therefore, in Figure 15.1.1, we pretend that you came across this interesting repository on GitHub



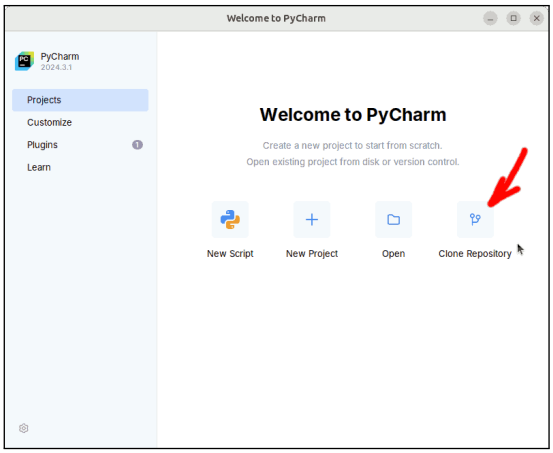
(15.1.1) Maybe we find an interesting repository on GitHub. Let's say it is `https://github.com/thomasWeise/databasesCode`. We click on the Code drop down menu.



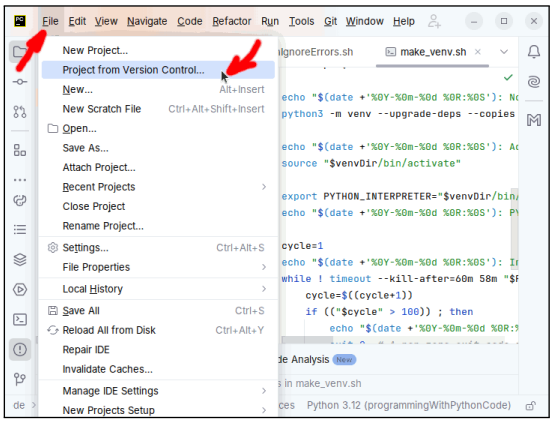
(15.1.2) We click on the button for copying the URL to the clipboard.



(15.1.3) Now the URL is copied to the clipboard and we can paste it wherever we like via Ctrl + V.

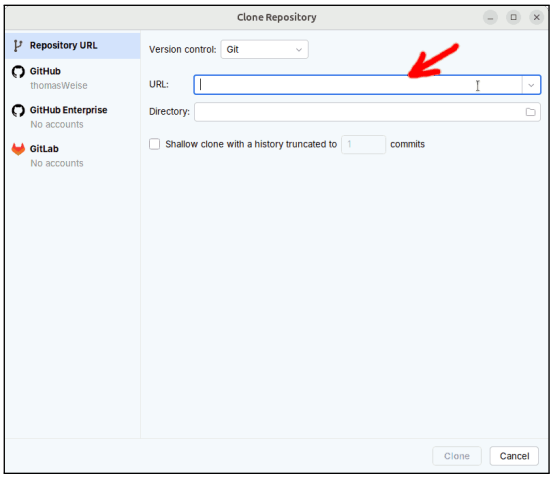


(15.1.4) We open PyCharm. If we get to the *Welcome to PyCharm* screen, we click Clone Repository.

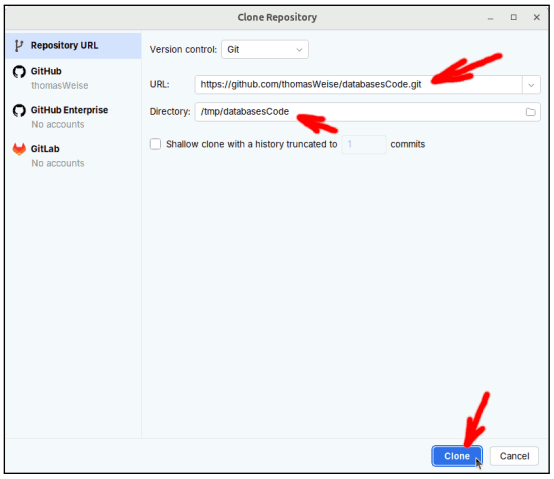


(15.1.5) If we already have a project open in PyCharm, we click on ☰ File Project from Version Control...

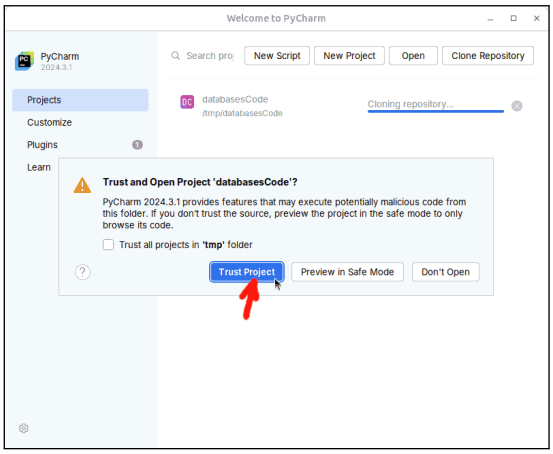
Figure 15.1: Cloning a Git (or GitHub) repository in PyCharm and configuring a virtual environment for it.



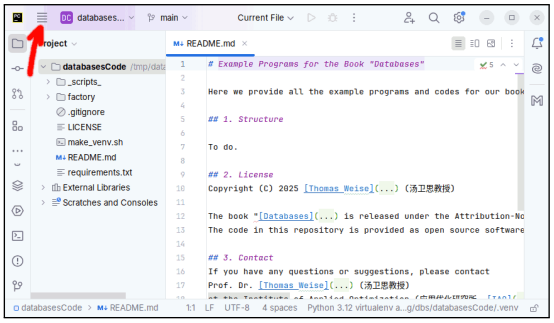
(15.1.6) The "Clone Repository" form appears.



(15.1.7) Normally, we would write the URL of the repository that we want to clone into the URL: field. Here, we paste the repository URL that we copied from the GitHub page with Ctrl + V. We also enter a directory where the repository should be copied to into the Directory: field. Here, I simply selected a folder on my Linux temporary files partition (because I will delete the project once I am done with this example). You would instead choose a more appropriate location. Then we click Clone.



(15.1.8) The download will begin. We may get asked whether we want to trust the downloaded project. If and only if we do trust it, we click Trust Project.

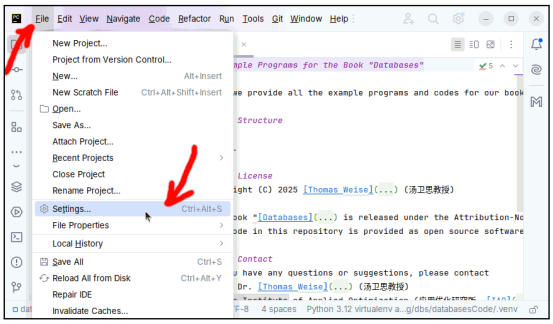


(15.1.9) The repository is downloaded and opens as new project in PyCharm. If this is a Python project, we now can configure its virtual environment settings (see Section 14.3). To do so, we click on ☰.

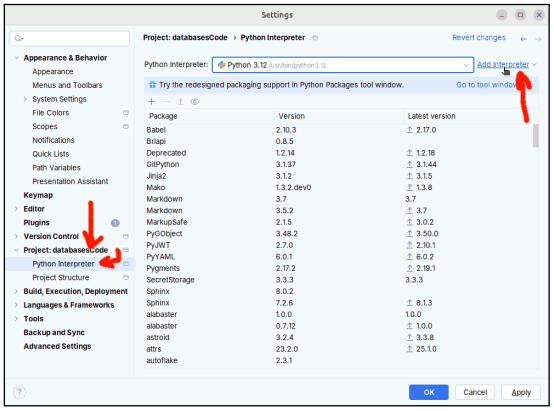
Figure 15.1: Cloning a Git (or GitHub) repository in PyCharm and configuring a virtual environment for it.

using your normal web browser. If you visit https://github.com/thomasWeise/databasesCode, you can see the big drop down menu Code. If you click on it, it shows the Hypertext Transfer Protocol Secure (HTTPS) URL under which the project can be found in Figure 15.1.2. If you work with GitHub, there are two ways to write a URL to a repository:

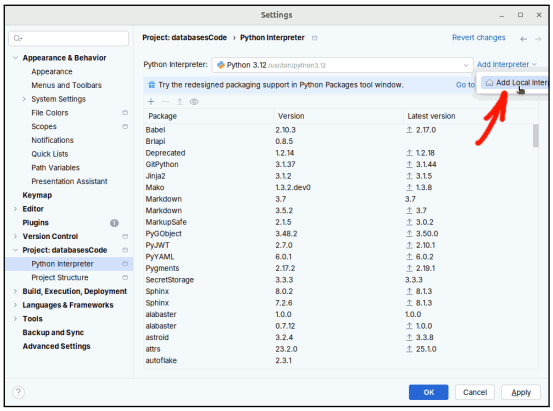
- https://github.com/user/repository (or https://github.com/user/repository.git) use HTTPS protocol to access the repository `repository` of user `user`. This form is often and commonly used.
- ssh://git@github.com/user/repository (or ssh://git@github.com/user/repository.git) use the Secure Shell Transport Layer Protocol (SSH) to access the repository `repository` of user `user`. I find this form more reliable when working with GitHub from China. However, it requires an SSH key to be configured for authentication [127].



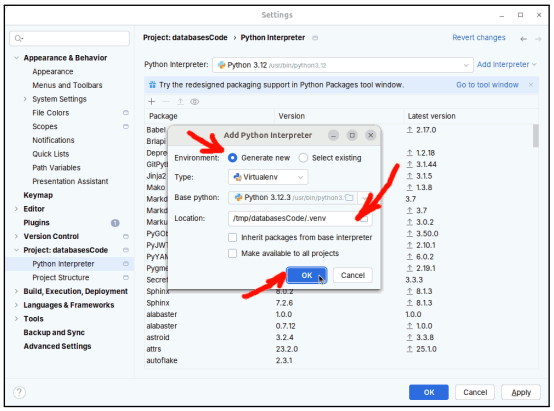
(15.1.10) In the menu that opens, we click on Settings... (we could also press Ctrl+Alt+S).



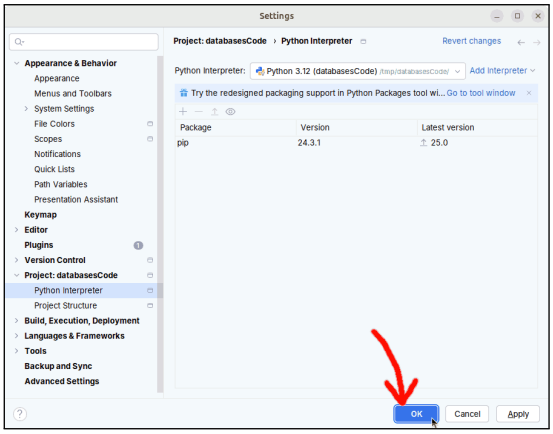
(15.1.11) The Settings menu opens. In the pane on the left-hand side, we click on the item Project: "our project" and then on Python Interpreter. This shows us something like the view on the left hand side: Maybe the system `Python` interpreter is selected or something else. We want to set up a virtual environment for our project, so we click on Add Interpreter.



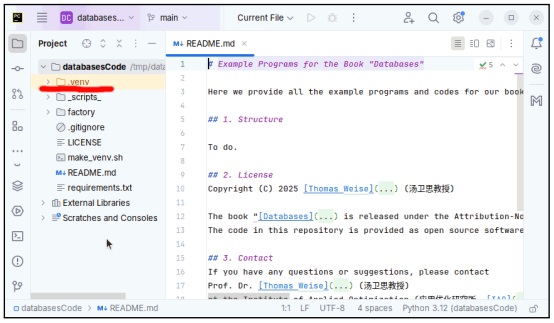
(15.1.12) We then click on Add Local Interpreter.



(15.1.13) In the "Add Python Interpreter" dialog that opens up, we select Generate New, choose Virtualenv, and type the sub-directory `.venv` relative to the path where we cloned the repository into as Location:. We then click OK.

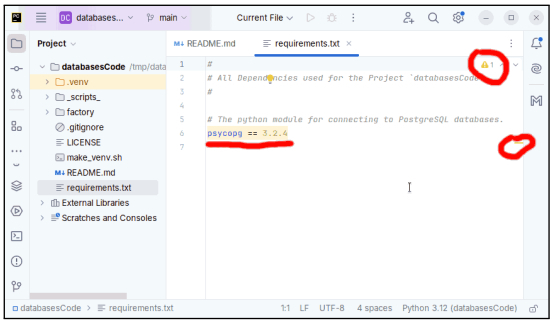


(15.1.14) The new environment is created and we click on OK.

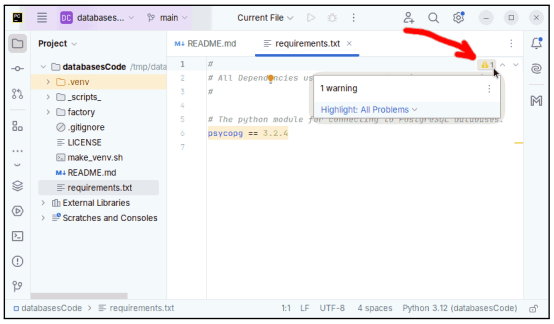


(15.1.15) Indeed, a directory called `.venv` appears in the directory view of our project.

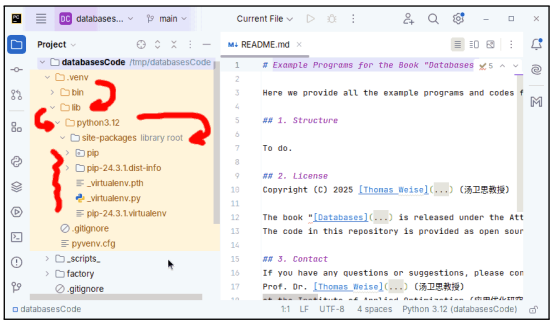
Figure 15.1: Cloning a Git (or GitHub) repository in `PyCharm` and configuring a virtual environment for it.



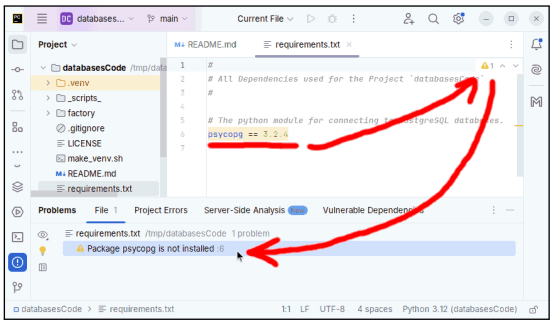
(15.1.16) Many `Python` projects come with a file `requirements.txt` or `requirements-dev.txt`. As discussed in Section 14.2, these list the libraries that the projects depend on. Our example repository also has a file `requirements.txt`, stating that it needs library `psycopg`. This dependency is marked with yellow color, because it is not installed in the virtual environment.



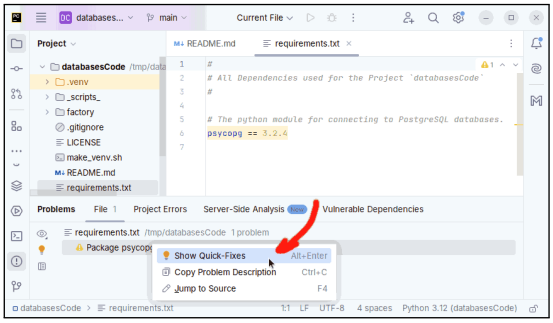
(15.1.17) Clicking on the warnings symbol ⚠ reveals this issue.



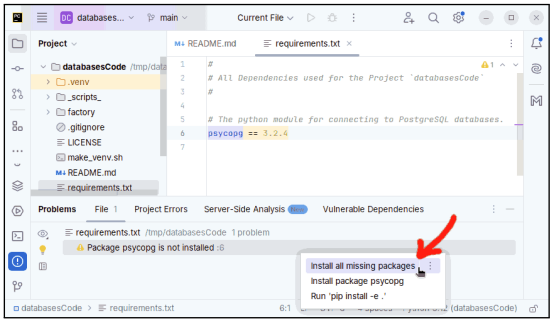
(15.1.18) Indeed: If we look at the `.venv` directory in the directory view, we cannot find the `psycopg` package.



(15.1.19) So we click on the requirements warning...



(15.1.20) ...and then on Show Quick-Fixes (or press Alt + ↵).

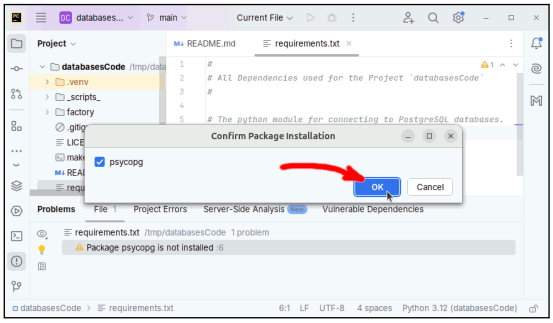


(15.1.21) In the menu that opens up, we select Install all missing packages.

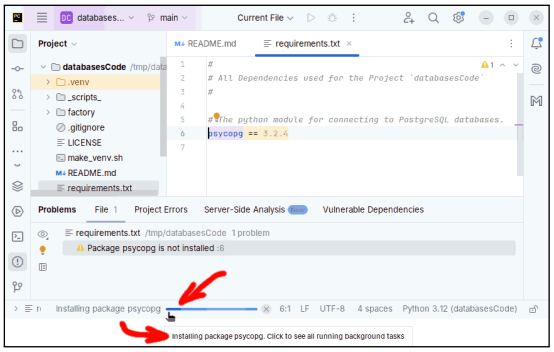
Figure 15.1: Cloning a Git (or GitHub) repository in `PyCharm` and configuring a virtual environment for it.

Here, obviously, `user` is `thomasWeise`, which is my personal GitHub account, and `repository` is `databasesCode`. The URL that will be copied to the clipboard by clicking the button in Figure 15.1.2 is `https://github.com/thomasWeise/databasesCode.git`. If you wanted to clone the repository with the example codes for this book instead, you would use `https://github.com/thomasWeise/programmingWithPythonCode.git`.

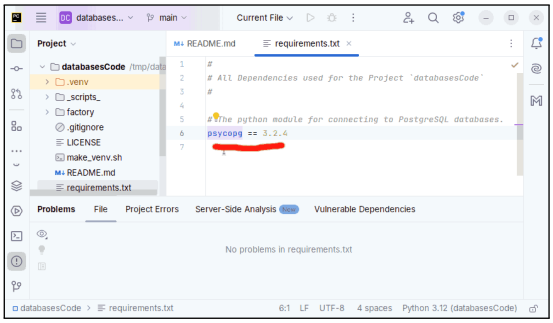
It is important to understand, however, that creating projects by cloning Git repositories is by no means restricted to GitHub. As stated before, Git is a client-server application. You could work in an enterprise that runs its own Git server. You could work with other Git-based repository hosts like gitee. Regardless of what Git service you use, you could use the very same way to type in the corresponding repository URL and then clone the repository in the same way. Only the structure of the URLs may be



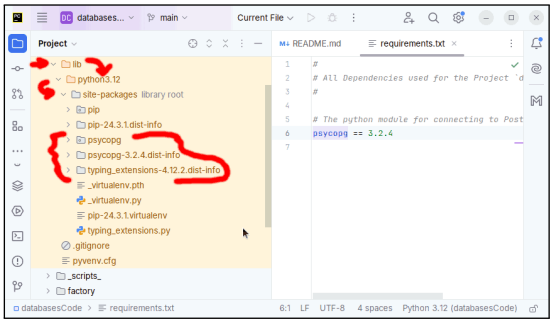
(15.1.22) We get presented a list of packages that will be installed. We click OK.



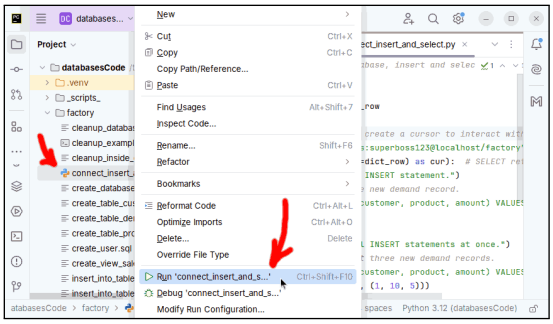
(15.1.23) The required package(s) will now be downloaded and installed.



(15.1.24) The yellow warnings marks in the `requirements.txt` file now disappear.



(15.1.25) And the packages have appeared in the `.venv` directory as well.



(15.1.26) All dependency packages are now installed. This means that the code in the repository that we have cloned will function now and we can run it.

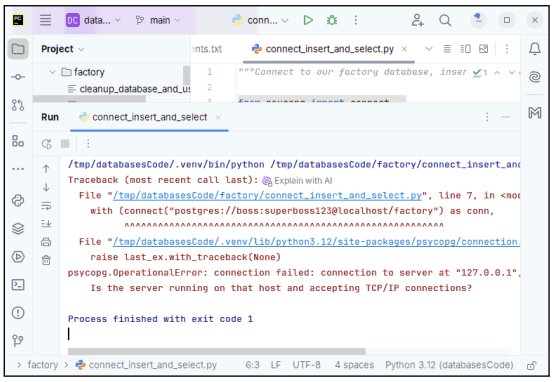
Figure 15.1: Cloning a Git (or GitHub) repository in PyCharm and configuring a virtual environment for it.

different.

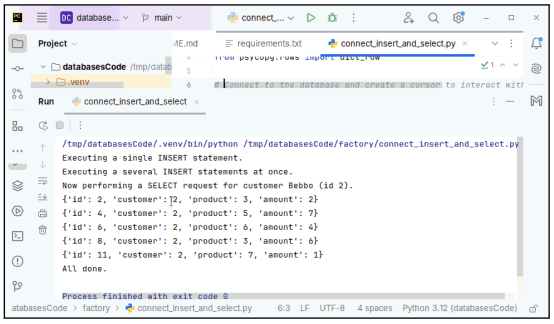
In Figure 15.1.2, we click on the button for copying the URL to the clipboard. Now the URL is copied to the clipboard (Figure 15.1.3) and we can paste it wherever we like via Ctrl+V. But where shall we paste it?

We open PyCharm. There are two things that can happen here: If we did not have a project open in PyCharm, the *Welcome to PyCharm* screen will pop up in Figure 15.1.4. We then click on Clone Repository. Alternatively, if we already had a project opened in PyCharm before, this project may be re-opened, as illustrated in Figure 15.1.5. In that case, we click through ≡ » File » Project from Version Control...

Either way, the "Clone Repository" form appears in Figure 15.1.6. Normally, we would now write the URL of the repository that we want to clone into the URL: field. Here, we paste the repository URL that we copied from the GitHub page with Ctrl+V. We also enter a directory where the repository should be copied to into the Directory: field. This directory is where all the files will be downloaded to. You would normally select some appropriate place in your filesystem where you store your program



(15.1.27) Well, there may be other issues unrelated to packages... ...the program I chose here is from our book on *Databases* [611], and it needs the PostgreSQL DBMS running with a specific DB ready. So likely, you cannot just run it ... it was just an example.



(15.1.28) If you also follow the *Databases* [611] course and have all other pieces of the software environment set up correctly, then you will see this output.

Figure 15.1: Cloning a Git (or GitHub) repository in PyCharm and configuring a virtual environment for it.

codes. I, however, simply selected a folder on my Linux temporary files partition, because I will delete the project once I am done with this example. Again, of course, you would instead choose a more suitable location. Once the information is entered, we click Clone in Figure 15.1.7.

Now the download of the source code and the whole project history will begin. Once this is finished, PyCharm imports the directory as a project. At this stage, we may get asked whether we want to trust the downloaded project. If and only if we do trust it, we click Trust Project in Figure 15.1.8. The repository is downloaded and opens as new project in PyCharm. At this point, we are basically done. The code and commit history is now on our machine. We can read it and work with it.

In many cases, though, we are not working with singular and monolithic stand-alone programs. Often, the programs we work with depend on other libraries. To use these programs, we also need to install these libraries. Luckily, we learned how to do that already in Chapter 14. In particular, we discussed virtual environments and even how to use them in PyCharm in Section 14.3. To make this example here more complete, we will exercise the full circle of how to get a virtual environment to work for PyCharm project based on a newly cloned GitHub repository.

Usually, after cloning a repository, we will want to configure its virtual environment settings. To do so, we click on ≡ in Figure 15.1.9. In the menu that opens, we click on Settings... or simply press Ctrl+Alt+S in Figure 15.1.10.

The Settings menu opens. It hosts a plethora of different options. Some are for PyCharm in general, some concern our new project. The latter is the group of settings we are after. In the pane on the left-hand side, we therefore click on the item Project: "our project", where "our project,", of course, is to be replaced with the name of our newly created project. We then click on Python Interpreter.

This shows us something like the view on the left hand side in Figure 15.1.11. Maybe the system Python interpreter is selected. Maybe the Python interpreter configured for the last project we used appears. Often, we do not want to use either of them. We want to set up a new virtual environment for our project, so we click on Add Interpreter.

We then click on the button Add Local Interpreter that pops up in Figure 15.1.12. (Sorry, I accidentally cut off a bit of this button in the screenshot. But I did not want to do the screenshot again, so I just used this slightly damaged one.)

The "Add Python Interpreter" dialog that opens up in Figure 15.1.13. We select Generate New and choose Virtualenv. In the Location: line, we type in a path to the sub-directory `.venv` relative to the path where we cloned the repository into. In other words, if we cloned the repository into path `/a/b`, we would write `/a/b/.venv`. Since I cloned the repository into `/tmp/databasesCode`, I now write `/tmp/.venv`. This directory `.venv` will be created and host the virtual environment after we click OK. The new environment is created and we click on OK in Figure 15.1.14. Indeed, a directory called `.venv` appears in the directory view of our project in Figure 15.1.15. It is still more or less empty, though.

Many Python projects come with a file `requirements.txt` or `requirements-dev.txt`. As discussed

in Section 14.2, these files list the libraries that the projects depend on. Our example repository also has a file `requirements.txt`, stating that it needs library psycopg. This dependency is marked with yellow color, because it is not installed yet in the virtual environment in Figure 15.1.16. Clicking on the warnings symbol ⚠ reveals this issue in Figure 15.1.17.

Indeed: If we look at the `.venv` directory in the directory view, we cannot find the psycopg package. Currently, this directory only contains the pip package in Figure 15.1.18. A long time ago, back in Section 4.2, we mentioned that all warnings and errors that our development IDE reports to us are important and should be fixed (by us). Clearly, a missing library is important, because without that package, the code we just downloaded would be useless and could not run.

To remedy this issue, we click on the requirements warning in Figure 15.1.19. A popup menu appears in which we click on Show Quick-Fixes (or press Alt+↵) in Figure 15.1.20. Another menu opens up and offers us several options to fix the warning. PyCharm cannot always know how an error or warning can be fixed (otherwise, programmers like you and me would no longer be needed...). But surely knows how to fix missing packages: We just have to click on Install all missing packages in Figure 15.1.21.

We get presented a list of packages that will be installed. We click OK in Figure 15.1.22. The required package(s) will now be downloaded and installed (see Figure 15.1.23).

After this is done, the yellow warnings marks in the `requirements.txt` file has disappeared in Figure 15.1.24. The packages have appeared in the `.venv` directory as well in Figure 15.1.25. All dependency packages are now installed. This means that the code in the repository that we have cloned will function now and we can run it in Figure 15.1.26.

To be honest, to actually run the code from `https://github.com/thomasWeise/databasesCode`, you would need some more ingredients. This is the code associated with our *Databases* course book [611]. Therefore, to actually work, it also needs a PostgreSQL server running and a DB set up based on several pieces of SQL code (that are also in the repository). So likely, you cannot just run exactly this code (see Figure 15.1.27). Well, I just used it as an example because it has a `requirements.txt` file. But it is also not unusual that you may need some additional tools. Yet, it is probably more common that you actually can directly work with the code after installing all dependencies. In the case that you are also following the *Databases* [611] course and have all other pieces of the software environment set up correctly, then the output you would see is given in Figure 15.1.28.

Regardless of the situation, you have now successfully cloned a repository from GitHub and installed all of its required libraries into a virtual environment.

15.1.2 Cloning git Repositories with the git Client

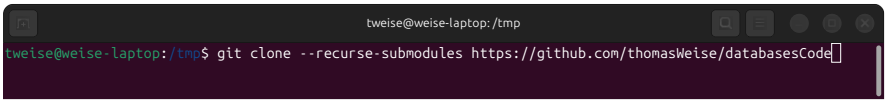
The most basic way to clone Git repositories is to use the Git client application. This is a command line tool that can be executed in a terminal. Here, we simply assume that you already have it installed the Git client program `git`.

In Figure 15.2, we illustrate how the Git client program is used under Ubuntu Linux. Under Microsoft Windows, it will work analogously. First, we need to open a terminal window. Under Ubuntu Linux, we therefore press Ctrl+Alt+T. Under Microsoft Windows, we instead press ⊞+R, type in `cmd`, and hit ↵. We then enter the directory into which we want to download the repository using `cd`. Notice that if we download a repository with the name `abc`, this will create a new directory named `abc` inside that current working directory.

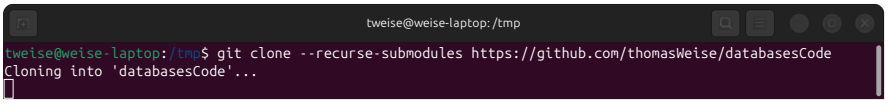
We then type the command `git clone https://github.com/thomasWeise/databasesCode` and hit ↵. Sometimes, repositories contain submodules. A submodules is basically a links to a specific state (commit) of *another* repository. The files of the referenced commit of the other repository would then exist inside a directory of the repository that we want to clone. If we want to have a full copy of all the files, including the files referenced this way, we need to add the option `--recurse-submodules` to the `git` command. We do this in Figure 15.2.1 to be on the safe side for demonstration purpose (although the repository `databasesCode` does *not* have any submodule).

After we hit ↵, the process of cloning the repository begins in Figure 15.2.2. Sadly, when we clone repositories using HTTPS URLs from GitHub, sometimes, there may be connectivity issues. I noticed that using the SSH URLs is much more reliable. An example of such an error is shown in Figure 15.2.3.

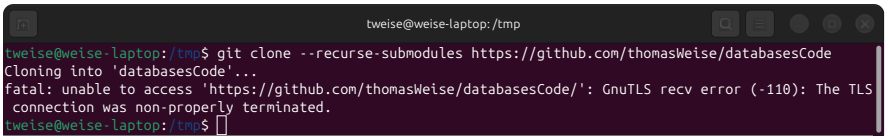
However, this is not a big problem. If such errors happen, you can simply try again after some time. I furthermore noticed that, often, changing your internet connection can solve the problem. Let's



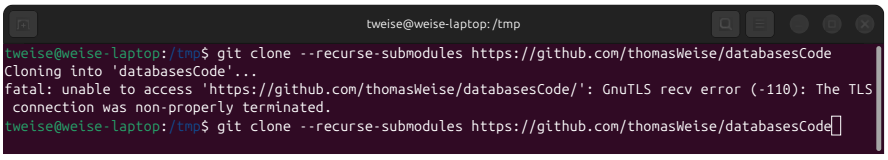
(15.2.1) After opening the Ubuntu Linux terminal using Ctrl + Alt + T, we enter the directory into which we want to download the repository. We then type the command `git clone https://github.com/thomasWeise/databasesCode` and hit ↵. The option `--recurse-submodules` can be added just in case.



(15.2.2) The process of cloning the repository begins.



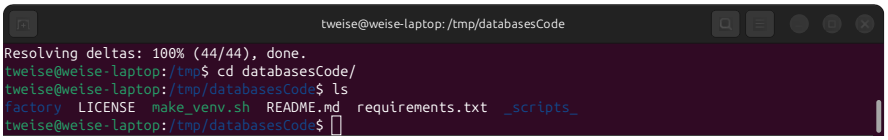
(15.2.3) Sometimes, there may be connectivity issues. But do not fret if that happens...



(15.2.4) ...simply try again (and, if necessary, again and again). Sometimes, it may make sense to switch the network, e.g., from using WLAN, instead connect your mobile phone to your computer using a USB cable and share the data connection of the mobile phone using USB tethering.



(15.2.5) Eventually, it will work.



(15.2.6) The repository has been downloaded and a new folder with the repository name (here: `databasesCode`) has appeared.

Figure 15.2: Cloning a Git repository using the Git client application `git` in an Ubuntu Linux terminal and how to deal with connection errors.

say that you are trying to clone the repository using your wired or wireless LAN and it does not work. Then it often helps to first switch of the network connection on your computer. On your mobile phone, you should also switch off wireless LAN and instead activate your mobile data connection. Then you can connect your mobile phone to your computer using a USB cable and select *USB tethering* on your phone. This will share the mobile data connection with your computer. Then, simply try to clone the repository again, as shown in Figure 15.2.4. The connection switching is just a quick work around. Often, `git clone` works at first attempt and, if not, works after you wait a minute or so.

Eventually, it will work, as shown in Figure 15.2.5. After the `git` command completes, the repository has been downloaded and a new folder with the repository name (here: `databasesCode`) has appeared. The Bash command `ls` lists the files in a directory. We see that the new directory contains the

downloaded files in Figure 15.2.6. Here, we will not deal with the issue of virtual environments, as we discussed them in the previous section about cloning repositories using PyCharm.

Backmatter

Best Practices

When writing programs in any programming language, there are always many best practices that should be followed. The same holds, of course, also for Python. Best practices are methods and styles that have crystallized over decades of programming experience. They help to avoid mistakes, increase the code maintainability, and improve performance.

Best Practice 1: The only proper way to run a Python application in a productive scenario is in the terminal, as shown in Section 2.4.1.

Best Practice 2: Always be careful with which division operator you use for `int`s. If you need an integer result, make sure to use `//`. Remember that `/` always returns a `float` (and see Best Practice 3), even if the result is a whole number.

Best Practice 3: Always assume that any `float` value is imprecise. Never expect it to be exact [38, 455].

Best Practice 4: If you need to specify large `int`s or `float`s, using underscores (`_`) to separate groups of digits can be very helpful [72]. For example, `37_859_378` is much easier to read than `37859378`.

Best Practice 5: We can use web-based resources, search tools, and AIs for finding answers to questions like "How do I do ... using [Programming Language]?". However, the answers that we get are not necessarily complete and may not be correct. We must make sure that we understand them fully. If the answers use functions that we are not familiar with, then we must look them up in the official authoritative documentation. Answers given by any non-authoritative source are never to be used verbatim without proper analysis.

Best Practice 6: AI tools and web-based non-authoritative resources can be used to *find* solutions. They should never be used to *document* solutions, because this must be done by a human. Documentation, i.e., the textual description of what the software does, must be done by actual people who fully understand the software.

Best Practice 7: When defining a string literal, the double-quotation mark variant (`"..."`) may be preferred over the single-quotation mark variant (`'...'`). (The *Style Guide for Python Code* [594] does not give a recommendation, but maybe for consistency with the *Docstring Conventions* [232], see also Best Practice 8.)

Best Practice 8: When defining a multi-line string literal, the double-quotation mark variant (`"""..."""`) is preferred over the single-quotation mark variant (`'''...'''`) [232, 594].

Best Practice 9: Comparisons to singletons like `None` should always be done with `is` or `is not`, never the equality operators `==` or `!=` [594].

Best Practice 10: Comments help to explain what the code in programs does and are a very important part of the *documentation* of code. Comments begin with a `#` character, after which all text is ignored by the Python interpreter until the end of the current line. Comments can either occupy a complete line or we insert two spaces after the last code character in the line and then start the comment [594].

Best Practice 11: Variable names should be lowercase, with words separated by underscores [594]. This style is called *snake case* or *snake_case* [656].

Best Practice 12: Regardless which programming language you are using, it is important to write code and scripts in a consistent style, to use a consistent naming scheme for all things that can be named, and to follow the generally established best practices and norms for that language.

Best Practice 13: The most important style guide for the Python programming language is PEP8: *Style Guide for Python Code* [594] at https://peps.python.org/pep-0008. Python code violating PEP8 is invalid Pythoncode.

Best Practice 14: Do never change the values of variables that are not supposed to be changed.

Best Practice 15: Always carefully *read* error messages. They often provide you very crucial information where to look for the mistake. Not reading error messages is wrong.

Best Practice 16: When writing code, we should always check whether the IDE notifies us about potential errors. In the case of PyCharm, these are often underlined in red or yellow color. We should always check all such marks!

Best Practice 17: Swapping of variable values can best be done with a multi-assignment statement, e.g., `a, b = b, a`.

Best Practice 18: The names we use in program code should clearly reflect our intentions.

Best Practice 19: Every program should pass static type checking with tools such as Mypy (see Useful Tool 4). Any issue found by the tools should be fixed. In other words, type check the program. If there is an error, fix the error and *type check it again*. Repeat this until no errors are found anymore.

Best Practice 20: A professional software engineer or programmer or, actually, any professional computer scientist knows many tools and is always keen to learn new tools.

Best Practice 21: Always use type hints.

Best Practice 22: It is important to integrate type hints from the very start of each project. The idea to first write code and later annotate it with type hints is wrong.

Best Practice 23: Each Python file should start with a string describing its purpose [232]. This can either be a single line, like a headline, or a longer text. In the second case, the first line must be a headline, followed by an empty line, followed by the rest of the text. Either way, it must be a string delimited by `"""..."""` [232, 594].

Best Practice 24: Use many static code analysis tools and use them always. They can discover a wide variety of issues, problems, or potential improvements. They can help you to keep your code clean and to enforce a good programming style. Do not just apply them, but also *implement* their suggestions wherever possible.

Best Practice 25: When you need to use an indexable sequence of objects, use a `list` only if you intent to modify this sequence. If you do not intend to change the sequence, use a `tuple`.

Best Practice 26: Only put immutable objects into tuples if you want to use the tuples as immutable objects. Mutable objects inside tuples makes the tuples modifiable as well, while other programmers may assume that they are immutable. Violating this assumption can lead to strange errors down the line.

Best Practice 27: Only put immutable objects into sets.

Best Practice 28: Sets are unordered. Never expect anything about how the objects you put into a set are actually stored there.

Best Practice 29: Careful when sorting or comparing strings: The default order is *uppercase characters before lowercase characters*, i.e., `"A" < "a"`. If you want that upper- and lowercase characters are treated the same (e.g., that `"A"` is considered as equal to `"a"`), as is the case in dictionary ordering, i.e., if you want to sort a collection `my_text` of strings in a case-insensitive manner, use `sorted(my_text, key=str.casefold)`.

Best Practice 30: Dictionary keys must be immutable (and hashable).

Best Practice 31: Blocks are indented by four spaces [594].

Best Practice 32: Prefer `elif` over nested `else ... if` constructs [378].

Best Practice 33: If your `if...else` statement is only used to decide which value to assign to a variable, use the inline ternary variant discussed in Section 6.4, as it is more compact [378].

Best Practice 34: If we do not care about the value of a variable (or parameter), we should name it `_` [324]. This information is useful for other programmers as well as static code analysis tools.

Best Practice 35: Due to the limited precision of floating point numbers, comparing the result of a floating point computation with another value using the strict `==` or `!=` operators is discouraged [26, 334]. It may lead to unanticipated results. For example `(0.1 + 0.2)== 0.3` gives `False`. Using functions like `isclose` from the `math` module that test whether two values are approximately the same based on their relative and absolute difference can be a (somewhat [227]) safer choice [26].

Best Practice 36: As a corollary of Best Practice 35: Do not use strict equality or inequality comparisons of `floats` as loop termination criteria [334], as they quite likely lead to endless loops that never terminate. There can always be inputs that cause endless oscillations between values or the appearance of `nan` values (see Section 3.3.5). The former issue can again be made somewhat less likely by using methods like the function `isclose` from the `math` module that checks whether two numbers approximately equal based on their relative and absolute difference [26].

Best Practice 37: Function names should be lower case, with underscores separating multiple words if need be [594]. This style is called *snake case* [656].

Best Practice 38: All parameters and the return value of a function should be annotated with type hints [638]. From my perspective: *A function without type hints is wrong.*

Best Practice 39: The body of a function is indented with four spaces.

Best Practice 40: Each function should be documented with a docstring. If you work in a team or intend to place your code in public repositories like on GitHub, then this very very much increases the chance that your code will be used correctly. From my perspective: *A function without docstring is wrong.*

Best Practice 41: After the function and its body are defined, leave *two* blank lines before writing the next code [594].

Best Practice 42: Package and module names should be short and lowercase. Underscores can be used to improve readability. [594]

Best Practice 43: Always attach a timeout to your unit tests. This timeout can be generous, maybe one hour, but it will serve as sentinel against either endless loops, deadlocks, or other congestion situations which all would be practical test failures. Timeouts protect automated builds or CI systems from clogging.

Best Practice 44: A function which is not unit tested is *wrong*.

Best Practice 45: Good unit tests for a given function should cover both expected as well as extreme cases. For a parameter, we should test both the smallest and largest possible argument values, as well as values from its normally expected range.

Best Practice 46: Good unit tests for a function should cover all branches of the control flow inside the function. If a function does one thing in one situation and another thing in another situation, then both of these scenarios should have associated unit tests.

Best Practice 47: Default parameter values must always be immutable [275].

Best Practice 48: Errors should *not* be ignored and input data should *not* be artificially sanitized. Instead, the input of functions should be checked for validity wherever reasonable. Faulty input should always be signaled by error conditions breaking the program flow. `Exceptions` should be raised as early as possible and whenever an unexpected situation occurs.

Best Practice 49: Any function that may raise an exception should explain any exception that it explicitly raises in the docstring. This is done by writing something like `:raises ExceptionType: why`

where `ExceptionType` is to be replaced with the type of the exception raised and `why` with a brief explanation why it will be raised.

Best Practice 50: The stack trace and error information printed on the Python console in case of an uncaught exception are essential information to identify the problem. They should *always* be read and understood before trying to improve the code. See also Best Practice 15.

Best Practice 51: Only `Exceptions` should be caught by `except` blocks that we can meaningfully handle [213, 594]. The `except` block is not to be used to just catch any exception, to implement GIGO, or to try to sanitize erroneous input.

Best Practice 52: Remember that, if an exception is raised, be aware that the control flow will immediately leave the current block. The statement in which the exception was raised will not be completed but aborted right away. Therefore, no variable assignments or other side effects can take place anymore and it is possible that variables remain undefined. Remember this when accessing variables that are assigned in a `try`-block.

Best Practice 53: It is important to cover both the reasonable expected use of our functions as well as unexpected use with incorrect arguments with test cases. The latter case should raise `Exceptions`, which we should verify with unit tests.

Best Practice 54: When measuring the runtime of code for one specific set of inputs, it makes sense to perform multiple measurements and to take the *minimum* of the observed values [571]. The reason is that there are many factors (CPU temperature, other processes, ...) that may *negatively* impact the runtime. However, there is no factor that can make your code faster than what your hardware permits. So the minimum is likely to give the most accurate impression of how fast your code can theoretically run on your machine. Notice, however, that there might be effects such as caching that could corrupt your measurements.

Best Practice 55: Whereever possible, the docstrings of functions, classes, and modules should contain doctests. This provides unit tests as well as examples as how the code should be used. Since doctests are usually brief, they are a quick and elegant way to complement more comprehensive unit tests in separate files (see Best Practice 44).

Best Practice 56: Generator expressions shall be preferred over list comprehension if the sequence of items only needs to be processed once. Generator expressions require less memory. If the iteration over the elements can stop early, which can happen, e.g., when using the `all` or `any` functions, they may also be faster.

Best Practice 57: Class names should follow the CapWords convention (often also called *camel case* [656]), i.e., look like `MyClass` or `UniversityDepartment` (not `my_class` or `university_department`) [594].

Best Practice 58: At the beginning of a `class`, a docstring is placed which describes what the class is good for. This docstring can include doctests to demonstrate the class usage. Such doctests can also be placed in the docstring of the module.

Best Practice 59: Object attributes must only be created inside the initializer `__init__`. There, an initial value must immediately be assigned to each attribute.

Best Practice 60: Every attribute of an object must be annotated with a type hint when created in the initializer `__init__` [348]. Here, type hints work exactly like with normal variables.

Best Practice 61: The type hint `Final` marks a variable or attribute as immutable. All attributes that you do not intend to change should be annotated with `Final`. Notice that this is a type hint, i.e., it will not be enforced by the Python interpreter [546] and malicious code can *still* change the attribute value. Type checkers like Mypy can detect such incorrect changes, though, and issue warnings.

Best Practice 62: An attribute is documented in the line *above* the attribute initialization by writing a *comment* starting with `#:`, which explains the meaning of the attribute [525]. (Sometimes, the documentation is given as string directly below the attribute definition [235], but we stick to the former method, because it has proper tool support, e.g., by Sphinx.)

Best Practice 63: All methods of `classes` must be annotated with docstrings and type hints.

Best Practice 64: When using a class `C` as type hint *inside* the definition of the class `C`, you must write `"C"` instead of `C`. (Otherwise, static code analysis tools and the `Python` interpreter get confused.)

Best Practice 65: Names of attributes and methods that start with a double leading underscore (`__`) are to be treated as *private* [457, 458]. They should not be accessed or modified from outside the class. All internal attributes and methods of a class that should not be exposed to the outside therefore should be named following this convention (with two leading underscores). `Python` applies some internal name mangling to make such attributes harder to access from outside the class[594]

Best Practice 66: For implementing `__eq__` and `__hash__`, the following rules hold [31]:

- Only immutable classes are allowed to implement `__hash__`, i.e., only classes where all attributes have the `Final` type hint and are only assigned on the initialize `__init__`.
- The result of `a.__hash__()` must never change (since `a` must never change either).
- If a class does not define `__eq__`, it cannot implement `__hash__` either.
- Instances of a class that implements `__eq__` but not `__hash__` cannot be used as keys in a dictionary or set.
- Only instances of a class that implements both `__eq__` and `__hash__` can be used as keys in dictionaries or sets.
- The results of `__eq__` and `__hash__` must be computed using the exactly same attributes. In other words, the attributes of an object `a` that determine the results of `a.__eq__(...)` must be exactly the same as those determining the results of `a.__hash__()`.
- It is best to compute `a.__hash__()` by simply putting all of these attributes into a `tuple` and then passing this `tuple` to `hash`.
- Two objects that are equal must have the same hash value, i.e., Equations 13.1 and 13.2 must hold.

Best Practice 67: Constants are module-level variables which must be assign a value upon definition and which must be annotated with the type hint `Final` [546].

Best Practice 68: The names of constants contain only capital letters with underscores separating words. Examples include `MAX_OVERFLOW` and `TOTAL` [594].

Best Practice 69: Constants are documented by writing a comment starting with `#:` immediately above them [525].

Best Practice 70: Every time you declare a variable that you do not intend to change, mark it with the type hint `Final` [546]. On the one hand, this conveys the intention "this does not change" to anybody who reads the code. On the other hand, if you do accidentally change it later, tools like `Mypy` can notify you about this error in your code.

Best Practice 71: Methods that return `self` should be annotated with the type hint `Self` [529]. Static code analysis tools then see that the method always returns an object of the same class as the object itself.

Best Practice 72: If a parameter of a function can take on only some special constant values `X`, say certain integer or string literals, the proper type hint is `Literal[X]` [346, 562]. The same holds if your function always returns the same, simple constant(s) `X`.

Best Practice 73: Packages should *always* be installed in virtual environments and never system-wide (maybe with the exception of `pip` and `venv`).

Best Practice 74: The command `pip install` should always be used with the option `--require-virtualenv`, e.g., `pip install --require-virtualenv thePackage`. This enforces that `pip` is really executed in a virtual environment and will cause an error otherwise.

Best Practice 75: The packages that a `Python` project requires or depends on should be listed in a file called `requirements.txt` in the root folder of the project.

Best Practice 76: All required packages in a `requirements.txt` file should be specified with the exact version, i.e., in the form of `package==version`. This ensures that the behavior of the project can be exactly replicated on other machines. It rules out that errors can be caused due to incompatible versions of dependencies, because it allows the users to exactly replicate the set up of the machine on which the project was developed.

Best Practice 77: `PyCharm` must **never** be used for running an application in a productive setting. It is *only* to be used as IDE for software development. For actually executing programs, always use a virtual environments in the terminal as introduced in Section 14.1. See also Best Practice 1.

Useful Tools

When developing software with Python, we can rely on a wide variety of tools that can help us to achieve high code quality and maintainability. We introduce such tools and show how they can be used throughout this book at different locations.

Useful Tool 1: The first place to look for information about Python is the official *Python 3 Documentation* at https://docs.python.org/3. This is the *authoritative* source about Python: Everything that is written there is *true* and *exact* regarding Python. All other sources may contain errors, be imprecise, ambiguous, or outdated. Therefore, always first consult the official Python documentation when in doubt or looking for information.

Useful Tool 2: Search engines are useful tools to find information about certain functionality. Writing a precise description of the problem or functionality into the search bar of a search engine can lead to pages that describe answers or take us to the official Python documentation. However, search engines can also lead us to pages containing wrong, incomplete, ambiguous, outdated, or otherwise useless information. It is important to compare whatever information we found with the official Python documentation [463] (see also Best Practice 5).

Useful Tool 3: The IDE and the error messages (`Exception` stack traces) are your most important tools to find errors. Read error messages. If your IDE – regardless whether it is PyCharm or something else – annotates your code with some marks, then you should check every single one of them.

Useful Tool 4: Mypy [347] is a static type checking tool for Python. This tool can warn you if you, e.g., assign values to a variable that have a different type than the values previously stored in the variable, which often indicates a potential programming error. It can be installed via `pip install mypy`, as illustrated in Figure 4.7 on page 103. You can then apply Mypy using the command `mypy fileToScan.py`, where `fileToScan.py` is the name of the file to check (you can also specify a whole directory). We use the Bash script given in Listing 16.1 on page 438 to apply Mypy to the example programs in this book.

Useful Tool 5: Ruff is a very fast Python linter that checks the code for all kinds of problems, ranging from formatting and style issues over missing documentation to performance problems and potential errors [378]. It can be installed via `pip install ruff` as shown in Figure 5.1 on page 117. You can then apply Ruff using the command `ruff check fileToScan.py`. We provide a script for using Ruff with a reasonable default configuration in Listing 16.2 on page 440.

Useful Tool 6: Pylint is a Python linter that analyzes code for style, potential errors, and possible improvements [461]. It can be installed via `pip install pylint` as shown in Figure 7.1 on page 153. You can then apply Pylint using the command `pylint fileToScan.py`. We provide a script for using Pylint with a reasonable default configuration in Listing 16.3 on page 441.

Useful Tool 7: pytest is a Python framework for writing and executing software tests [329]. It can be installed via `pip install pytest pytest-timeout` as shown in Figure 8.2 on page 170. You can then apply pytest using the command `pytest --timeout=toInS file(s)`, where `toInS` should be replaced with a reasonable timeout in seconds and `file(s)` is one or multiple files with test cases. We provide a script for using pytest with a reasonable default configuration in Listing 16.4 on page 442. See also Useful Tool 9 later on.

Useful Tool 8: `timeit` is a tool for measuring execution time of small code snippets that ships directly with Python. This module avoids a number of common traps for measuring execution times, see [442, 571].

Useful Tool 9: A doctest is a unit test written directly into the docstring of a function, class, or module. We therefore insert a small snippet of Python code followed by its expected output. The first line of such codes is prefixed py `>>>`. If a statement needs multiple lines, any following line is prefixed by `...`. After the snippet, the expected output is written. The doctests can be performed by modules like `doctest` [160] or tools such as `pytest` [328] (Useful Tool 7). They collect the code, run it, and compare its output to the expected output in the docstring. If they do not match, the tests fail. We use `pytest` in this book, with the default configuration given in Listing 16.5.

Useful Tool 10: A debugger is a tool that ships with many programming languages and IDEs. It allows you to execute a program step-by-step while observing the current values of variables. This way, you can find errors in the code more easily [5, 483, 636]. A comprehensive example on how to use the debugger in `PyCharm` is given in Section 13.4.

Useful Tool 11: `pip` is a software that can be used to install packages under `Python`.

- With the command `pip install thePackage`, the package `thePackage` is installed.
- With the command `pip install "thePackage==version"`, the version `version` of the package `thePackage` is installed.
- With the command `pip install -r requirements.txt`, the packages listed in the requirements file `requirements.txt` are installed. A requirements file allows to put multiple package/version dependencies that would otherwise command line arguments of `pip` into a single file [449].

`pip` should always be used in a virtual environment, see Best Practice 73 and 74.

Useful Tool 12: The module `venv` is used for creating and managing virtual environments under `Python`.

Glossary

$i!$ The factorial $a!$ of a natural number $a \in \mathbb{N}_1$ is the product of all positive natural numbers less than or equal to a, i.e., $a! = 1 * 2 * 3 * 4 * \cdots * (a-1) * a$ [106, 175, 370]. See also Equation 8.1 and Listing 8.1

ϕ The golden ratio (or golden section) ϕ is the irrational number $\frac{1+\sqrt{5}}{2}$. It is the ratio of a line segment cut into two pieces of different lengths such that the ratio of the whole segment to that of the longer segment is equal to the ratio of the longer segment to the shorter segment [96, 187]. The golden ratio is approximately $\phi \approx 1.618\,033\,988\,749\,894\,848\,204\,586\,834$ [517]. Represented as `float` in Python, its value is `1.618033988749895`.

π is the ratio of the circumference U of a circle and its diameter d, i.e., $\pi = U/d$. $\pi \in \mathbb{R}$ is an irrational and transcendental number [199, 306, 414], which is approximately $\pi \approx 3.141\,592\,653\,589\,793\,238\,462\,643$. An early method to approximate π was developed by the Chinese mathematician LIU Hui (刘徽) based on inscribing regular e-gons into a circle somewhere in the third century CE [420, 649] in his commentary to the famous Chinese mathematics book *Jiu Zhang Suanshu* (九章算术) [140, 144, 311, 420, 540]. In Python, π is provided by the `math` module as variable `pi` with value `3.141592653589793`. In PostgreSQL, it is provided by the SQL function `pi()` with value `3.141592653589793` [381].

$(1+1)$ **EA** The $(1+1)$ EA is a local search algorithm that retains the best solution x_c discovered so far during the search [99, 172]. In each step, it applies a unary search operator to this best-so-far solution x_c and derives a new solution x_n. If the new solution x_n is *better or equally good* when compared with x_c, i.e., not worse, then it replaces it, i.e., is stored as the new x_c. If the search space are bit strings of length n, then the $(1+1)$ EA uses a unary search operator that flips each bit independently with probability m/n, where usually $m = 1$. This operator is the main difference to randomized local search (RLS). The $(1+1)$ EA is a special case of the $(\mu+\lambda)$ evolutionary algorithm ($(\mu+\lambda)$ EA) where $\mu = \lambda = 1$.

$(1+1)$ **FEA** The $(1+1)$ evolutionary algorithm ($(1+1)$ EA) with Frequency Fitness Assignment (FFA) plugged in [620, 621, 654]. On the discrete bit-string domains, this algorithm has a very specific structure: Its unary search operator flips each bit with probability $1/n$, where n is the dimension of the search space, distinguishing it from the unary operator in RLS with FFA (FRLS) which flips exactly one bit. On other domains, such as permutations, the term is used synonymously with FRLS.

$(\mu+\lambda)$ **EA** The $(\mu+\lambda)$ EA is an evolutionary algorithm (EA) where, in each generation, λ offspring solutions are generated from the current population of μ parent solutions. The offspring and parent populations are merged, yielding $\mu+\lambda$ solutions, from which then the best μ solutions are ratained to form the parent population of the next generation. If the search space is the bit strings of length n, then this algorithm usually applies a mutation operator flipping each bit independently with probability $1/n$.

$i..j$ with $i, j \in \mathbb{Z}$ and $i \leq j$ is the set that contains all integer numbers in the inclusive range from i to j. It always holds that $i..j \subset \mathbb{Z}$. For example, 5..9 is equivalent to $\{5, 6, 7, 8, 9\}$

BinInt The *Binary Integer Problem* is a benchmark problem for bit-string based black-box search [567, 654]. It is very similar to OneMax: It tries to maximize the number of 1 bits in the bit string of a chosen dimension n. Different from OneMax, where each bit has the same weight in the objective function, the bits of BinInt have a exponentially increasing weight: The bit string is treated as a binary-encoded integer whose value should be maximized. The objective function $f(x)$ of the

minimization variant of this problem can be defined as $f(x) = 2^n - 1 - \sum_{i=1}^{n} 2^{i-1} \cdot x_i$. The problem is unimodal, i.e., has only a single global optimum and no local optimum. It is fully separable, i.e., we can optimize each variable separately.

LinHarm The Linear Harmonic Function is a benchmark problem for bit-string based black-box search [164, 172, 654]. It is very similar to OneMax: It tries to maximize the number of 1 bits in the bit string of a chosen dimension n. Different from OneMax, where each bit has the same weight in the objective function, the bits of LinHarm have a linear increasing weight. The objective function $f(x)$ of the minimization variant of this problem can be defined as $f(x) = \frac{1}{2}n(n+1) - \sum_{i=1}^{n} i \cdot x_i$. The problem is unimodal, i.e., has only a single global optimum and no local optimum. It is fully separable, i.e., we can optimize each variable separately. The $(1+1)$ EA can solve this problem with an expected runtime in $\mathbf{\Theta}(n \ln n)$ [164, 172].

TSP Suite is a open source benchmarking framework for the Traveling Salesperson Problem (TSP) written in Java[616]. It uses the instances of the TSPLib and offers both automated experiment execution as well as several metrics to measure runtime. It is available at `https://thomasweise.github.io/tspSuite` and has been used in many research works, such as [622]. It can be viewed as a predecessor of `moptipy`.

2BP In the two-dimensional Bin Packing Problem (BP), the goal is to pack rectangles into rectangular bins without overlap and such that their edges are parallel to the edges of the bins [287, 361, 363]. The 2BP is strongly $\mathcal{NP}$-hard [363]. The variants of this problem include the 2BP|O|F, 2BP|R|F, 2BP|O|G, and 2BP|R|G [362, 363]. Different objective function for the 2BP|O|F and 2BP|R|F are presented in [657]. 2BPs have many practical applications, including cutting (wood, glas) [363], packing (transportation, warehouse) [363], and newspaper page layout [541].

2BP|O|F In the oriented two-dimensional Bin Packing Problem (2BP) with free cuts, the rectangles to pack have fixed orientations, i.e., cannot be rotated, but can otherwise be arranged freely [362, 363, 379, 658]. Permutation-based search spaces can be used for this problem by using the Improved Bottom Left (IBL) encoding. The state-of-the-art (SOTA) for this problem comprises [57, 62, 118, 229, 435, 658].

2BP|O|G In the oriented 2BP with guillotine cuts, the rectangles to pack have fixed orientations, i.e., cannot be rotated, and must be arranged such that they can be extracted by edge-to-edge cuts [362, 363].

2BP|R|F In this variant of the 2BP with free cuts, the rectangles to pack can be rotated by 90° if need be and can freely be arranged [50, 362, 363, 657, 658]. Besides packing and cutting, this problem also occurs in scheduling and build formation in additive manufacturing [352, 447]. The almost-squares-in-almost-squares (Asqas) problem [585] is also a 2BP|R|F. Search spaces based on signed permutations can be used for this problem by using the IBL encoding. The SOTA for this problem comprises [57, 118, 229, 657, 658].

2BP|R|G In this variant of the 2BP with guillotine cuts, the rectangles to pack can be rotated by 90° if need be, but must be arranged such that they can be extracted by edge-to-edge cuts [362, 363].

A the $n \times n$ distance matrix. $A[a, b]$ indicates the distance between locations a and b with $a, b \in 1..n$. A distance matrix can be symmetric, in which case $A[a, b] = A[b, a]\ \forall a, b \in 1..n$. Distance matrices represent the problem instance data in TSPs and part of the instance data of Quadratic Assignment Problems (QAPs).

ACO *Ant Colony Optimization* is a nature-inspired optimization method for combinatorial problems where solutions are generated by "ants" that move from node to node in a graph choosing edges based on (1) the simulated pheromone on the edges and (2) a per-edge heuristic value [167–169]. If an ant produced a good solution, "pheromone" is distributed over the edges it visited, making it more likely to be re-visited by other ants.

AI Artificial Intelligence, see, e.g., [487]

Android is a common operating system for mobile phones [514].

API An *Application Programming Interface* is a set of rules or protocols that enables one software application or component to use or communicate with another [236].

apt-get is a command used to install Debian `deb` packages under Ubuntu Linux [253, 511, 596]. Using `apt-get` requires superuser privileges (`sudo`), i.e., one usually does `sudo apt-get install...`. Learn more at `https://salsa.debian.org/apt-team/apt`.

Asqas An "almost-square" be a rectangle with size $i \times (i+1)$ with $i \in \mathbb{N}_1$. The *almost-squares-in-almost-squares* problem [585] is a special case of the 2BP|R|F. Here, almost-square rectangles of sizes 1×2, 2×3, ..., $n \times (n+1)$ must be placed into an almost-square bin without waste. This requires that there exists a natural number $p \in \mathbb{N}_1$ such that $p(p+1) = \sum_{i=1}^{n} i(i+1)$ [585]. There are exacty five Asqas instances [585], namely with bin sizes 1×2 for $n = 1$, $p = 1$, 5×4 for $n = 3$, $p = 4$, 15×16 for $n = 8$, $p = 15$, 55×56 for $n = 20$, $p = 55$, and 119×120 for $n = 34$, $p = 119$.

ATSP The general case for TSPs are *asymmetric Traveling Salesperson Problems*, where the $n \times n$ distance matrix A does not need to be symmetric. In other words, it is possible that $\exists a, b \in 1..n$ with $A[a, b] \neq A[b, a]$. Notice that while such index tuple *may* exist, this is not always or necessarily the case. In this case, one may consider the STSP as a special case of the ATSP, while ATSPs may be considered as synonymous to TSPs.

B the $n \times n$ flow matrix. $B[a, b]$ is the flow of goods between locations a and b with $a, b \in 1..n$. Flow matrices are one half of the problem instance data of QAPs.

B&B *Branch and bound* is a family of optimization methods that combines systematic exploration of the search space with both and objective function f subject to minimization and a lower bound function. The idea is that we divide the search space into two sets. We then pick the set that seems more interesting and divide it again. We do this recursively until we have arrived at a set of size 1, which is our first solution y. We can compute its objective value $z = f(y)$. Now we continue with the sets that we have created and not yet explored. However, we can discard all sets for which the lower bound of the solution quality is $\geq z$, because they can never give us a better solution than y. If we have divided another set down to size 1 and it yields a better solution than y, of cause we update these values. Eventually, we have either discarded or fully expanded all sets – and then the solution we got must be optimal. B&B-algorithms have been applied to the TSP [359], the Job Shop Scheduling Problem (JSSP) [95], the oriented two-dimensional Bin Packing Problem with free cuts (2BP|O|F) [379], and maritime logistics [642], for instance.

banker's rounding means rounding a real number to the closest integer and, in case of ties, to the closest *even* integer [25, 440]. Since ties can only occur for numbers ending in `.5`, this means that all numbers of the form `z.5` will be rounded to `z+1` if `z` is odd and to `z` if `z` is even. This is different from the "normal" understanding of rounding as we often learn in school, where all numbers of the form `z.5` are rounded to `z+1`. Banker's rounding has the advantage that it is not biased in any direction [271, 383]. If `z.5` is always rounded toward `z+1` and we have many numbers of the form `z.5`, then the sum or average of the results tends to be somewhat larger than the sum or average of the unrounded numbers. Banker's rounding avoids this problem by sometimes rounding numbers ending in `5` to larger and sometimes to smaller values. In the IEEE Floating-Point Arithmetic Standard 754 [282], this method corresponds to the *roundTiesToEven* direction introduced in its section *4.3.1 Rounding-direction attributes to nearest*. Python uses banker's rounding in its `round` function [86], whereas Java's `Math.round` uses the "school-type" rounding [297].

Bash is a shell used under Ubuntu Linux, i.e., the program that "runs" in the terminal and interprets your commands, allowing you to start and interact with other programs [73, 410, 653]. Learn more at `https://www.gnu.org/software/bash`.

BCE The time notation *before Common Era* is a non-religious but chronological equivalent alternative to the traditional *Before Christ (BC)* notation, which refers to the years *before* the birth of Jesus Christ [102]. The years BCE are counted down, i.e., the larger the year, the farther in the past. The year 1 BCE comes directly before the year 1 CE [499, 659].

bias The bias is subtracted from the value stored in the exponent field of a floating point number. This allows for representing both positive and negative exponents. In the 64 bit double precision IEEE Standard 754 floating point number layout [267, 282], the bias is 1023. See Section 3.3.1.

BP The *Bin Packing Problem* [176, 493] is one of the classical problems from Operations Research (OR). The goal is to pack a set of objects into containers, using as few containers as possible. Different flavors of BPs can be distinguished, e.g., by the number of dimensions [150, 361] that the objects and containers have (see, e.g., the 2BP), and/or whether the objects can be rotated [658], as well as other constraints and limitations. Both the one- and two-dimensional BP are strongly $\mathcal{NP}$-hard [363].

breakpoint A breakpoint is a mark in a line of code in an IDE at which the debugger will pause the execution of a program.

BRKGA The *Biased Random Key Genetic Algorithm* is a EA with a continuous search space $\mathbb{X} \subset [0,1]^n$ and a decoding function γ translating this space to something else and biased, uniform, and elitist mating strategies [231, 366].The *bias* comes from the fact that one of the parents used for the binary operation is biased to be of higher fitness than the other parent [231]. Often, γ is the Random Keys encoding [39, 40] procedure translating $\mathbb{X}$ to permutations of the first n natural numbers [230]. Sometimes, it is actually a Memetic Algorithm (MA) further refining the translation result with a local search (LS) [230]. Sometimes the permutations are then further translated to more complex data structures, such as Gantt charts for the JSSP [230], or packing plans for 2BP 3D packing [229].

C is a programming language, which is very successful in system programming situations [159, 459, 614].

case-insensitive refers to a way to compare text and characters, in which uppercase letters like "A" or "Z", respectively, are considered to be equal to their lowercase counterparts such as "a" and "z", respectively. It is thus the opposite of case-sensitive comparisons, where the letters are considered to be the different.

case-sensitive refers to a way to compare text and characters, in which uppercase letters like "A" or "Z" are considered to be different from lowercase letters like "a" and "z". It is thus the opposite of case-insensitive comparisons, where the letters are considered to be the same. In `Python`, `Java`, and `C`, most string operations such as sorting, equality comparison, and finding are case-sensitive.

CE The time notation *Common Era* is a non-religious but chronological equivalent alternative to the traditional *Anno Domini (AD)* notation, which refers to the years *after* the birth of Jesus Christ [102]. The years CE are counted upwards, i.e., the smaller they are, the farther they are in the past. The year 1 CE comes directly after the year 1 BCE [499, 659].

CI *Continuous Integration* is a software development process where developers integrate new code into a codebase hosted in a VCS, after which automated tools run an automated build process including code analysis (such as linters) and unit test execution [350]. If the build succeeds and no errors or problems with the code are, the code may automatically be deployed (if the CI system is configured to do so).

client In a client-server architecture, the client is a device or process that requests a service from the server. It initiates the communication with the server, sends a request, and receives the response with the result of the request. Typical examples of clients are web browsers in the internet as well as clients for DBMSes, such as `psql`.

client-server architecture is a system design where a central server receives requests from one or multiple clients [55, 367, 427, 471, 480]. These requests and responses are usually sent over network connections. A typical example of such a system is the World Wide Web (WWW), where web servers host websites and make them available to web browsers, the clients. Another typical example is the structure of DB software, where a central server, the DBMS, offers access to the DB to the different clients. Here, the client can be some terminal software shipping with the DBMS, such as `psql`, or the different applications that access the DBs.

Concorde is an exact solver for the TSP[15, 130, 131], which could even solve the largest instance of TSPLib (with 85 900 cities) to optimality.

CSP The *Cutting Stock Problem* is about cutting shapes of a larger shape while reducing waste [363, 372]. It is closely related to the BP.

CSV *Comma-Separated Values* is a very common and simple text format for exchanging tabular or matrix data [504]. Each row in the text file represents one row in the table or matrix. The elements in the row are separated by a fixed delimiter, usually a comma (","), sometimes a semicolon (";"). Python offers some out-of-the-box CSV support in the `csv` module [138].

DB A *database* is an organized collection of structured information or data, typically stored electronically in a computer system. Databases are discussed in our book *Databases* [611].

DBMS A *database management system* is the software layer located between the user or application and the DB. The DBMS allows the user/application to create, read, write, update, delete, and otherwise manipulate the data in the DB [645].

debugger A debugger is a tool that lets you execute a program step-by-step while observing the current values of variables. This allows you to find errors in the code more easily [5, 483, 636]. See also Useful Tool 10.

denominator The number b of a fraction $\frac{a}{b} \in \mathbb{Q}$ is called the *denominator*.

docstring Docstrings are special string constants in Python that contain documentation for modules or functions [232]. They must be delimited by `"""..."""` [232, 594].

doctest *doctests* are unit tests in the form of as small pieces of code in the docstrings that look like interactive Python sessions. The first line of a statement in such a Python snippet is indented with Python»> and the following lines by `...`. These snippets can be executed by modules like `doctest` [160] or tools such as `pytest` [328]. Their output is the compared to the text following the snippet in the docstring. If the output matches this text, the test succeeds. Otherwise it fails.

DS Data Science, see, e.g., [243].

e is Euler's number [189], the base of the natural logarithm. $e \in \mathbb{R}$ is an irrational and transcendental number [199, 306], which is approximately $e \approx 2.718\,281\,828\,459\,045\,235\,360$. In Python, it is provided by the `math` module as variable `e` with value `2.718281828459045`. In PostgreSQL, you can obtain it via the SQL function `exp(1)` as value `2.718281828459045` [381].

EA An *evolutionary algorithm* is a metaheuristic optimization method that maintains a population of candidate solutions, which undergo selection (where better solutions are chosen with higher probability) and reproduction (where mutation and recombination create a new candidate solution from one or two existing ones, repectively) [24, 612].

escape sequence Escaping is the process of presenting "forbidden" characters or symbols in a sequence of characters or symbols. In Python [615], string escapes allow us to include otherwise impossible characters, such as string delimiters, in a string. Each such character is represented by an *escape sequence*, which usually starts with the backslash character ("\") [184]. In Python strings, the escape sequence `\"`, for example, stands for `"`, the escape sequence `\\` stands for `\`, and the escape sequence `\n` stands for a newline or linebreak character, see Section 3.6.4. In Python f-strings, the escape sequence `{{` stands for a single curly brace `{`. In PostgreSQL [611], similar C-style escapes (starting with "\") are supported [542].

exit code When a process terminates, it can return a single integer value (the exit status code) to indicate success or failure [307]. Per convention, an exit code of 0 means success. Any non-zero exit code indicates an error. Under Python, you can terminate the current process at any time by calling `exit` and optionally passing in the exit code that should be returned. If `exit` is not explicitly called, then the interpreter will return an exit code of 0 once the process normally terminates.If the process was terminated by an uncaught `Exception`, a non-zero exit code, usually 1, is returned.

exponent The exponent is the part of a floating point number that stores a power of 2 with which the significand is multiplied. This allows for covering a wide range of different precisions and representing both very large and very small numbers. In the 64 bit double precision IEEE Standard 754 floating point number layout [267, 282], the exponent is 11 bits with a bias of 1023. See Section 3.3.1.

$f(y)$ The *objective function* $f : \mathbb{Y} \mapsto \mathbb{R}$ can compute a cost or rating for each candidate solution y in the solution space $\mathbb{Y}$. The smaller this value, the better the solution, i.e., in the context of this work, objective functions are subject to minimization. Objective functions are not necessarily continuous and it is also quite common that they have integer or natural numbers as results, e.g., we often have $f : \mathbb{Y} \mapsto \mathbb{N}_0$ instead of $f : \mathbb{Y} \mapsto \mathbb{R}$.

f-string let you include the results of expressions in strings [79, 210, 218, 237, 384, 521]. They can contain expressions (in curly braces) like `f"a{6-1}b"` that are then transformed to text via (string) interpolation, which turns the string to `"a5b"`. F-strings are delimited by `f"..."`. f-strings are discussed in Section 3.6.2.

FFA *Frequency Fitness Assignment* (中文: 频率适应度分配) is a algorithm plugin for optimization methods applied to discrete or combinatorial problems with not-too-many different possible objective values [618, 620, 621]. It replaces the objective values in all comparisons with their absolute encounter frequency so far during the search. FFA has successfully been applied to the Maximum Satisfiability (MaxSAT) [620, 621], the TSP [353, 354], and the QAP [112, 569], amongst others. Its properties have been intensely researched on discrete benchmark functions such as OneMax, LeadingOnes, Trap, TwoMax, Jump, and Plateau [620, 621, 654]. The most common realizations of FFA so far are the $(1+1)$ FEA and FRLS.

Flask is a lightweight `Python` framework that allows developers to quickly and easily build web applications [4, 114, 574]. It is based on the `Python` WSGI standard [179]. Learn more at `https://flask.palletsprojects.com`.

FSSP The *Flow Shop Scheduling Problem* is a variant of a JSSP where all products go through the same sequence of machines [238, 303].

$\gamma(x)$ The decoding function $\gamma : \mathbb{X} \mapsto \mathbb{Y}$ translates point from the search space $\mathbb{X}$ to one point y in the solution space $\mathbb{Y}$. It is a very common case that $\mathbb{X} = \mathbb{Y}$ and that the decoding function γ is the identity mapping. Examples for encodings that translate from a simple search space $\mathbb{X}$ to a complex solution data structure $\mathbb{Y}$ are the IBL algorithm that translates (potentially signed) permutations to two-dimensional packings on the 2BP, the permutation-based encoding of Gantt charts for the JSSP from [58, 59, 219, 508, 617], or the permutation-based encoding of game plans for the Traveling Tournament Problem (TTP) from [641]. Encodings can also translate between types of search spaces. The Random Keys encoding, for example, translates from $[0,1]^n$ to permutations of the first n natural numbers..

GA A *Genetic Algorithm* is an EA usually for bit-string based search spaces [24, 147, 228, 395, 402, 612] (although many researchers use the terms EA and GA synonymously). It is a population-based metaheuristic applying a unary and a binary search operater as well as a selection scheme. The original GA is the Standard Genetic Algorithm (SGA), which today is considered outdated. Modern variants are the Greedy $(2+1)$ Genetic Algorithm (Greedy $(2+1)$ GA) and the Self-Adjusting $(1+(\lambda,\lambda))$ Genetic Algorithm (Self-Adjusting $(1+(\lambda,\lambda))$ GA) as well as bit-string based $(\mu+\lambda)$ EAs.

Gantt chart Gantt charts are two-dimensional diagrams illustrating the steps of a process [318, 635]. Their horizontal axis represents the time, the vertical axis represents work packages, machines, or workers. In the diagram area, bars located at certain vertical locations indicate that the work package, machine, or worker is processed or works at the time range covered by the bar as projected to the horizontal axis.

GECCO The *Genetic and Evolutionary Computation Conference* is one of the largest conferences in the field of EAs. It has been established in 1999 and its proceedings are published by ACM. Learn more at `https://sig.sigevo.org/GECCOs`.

GIGO Garbage In–Garbage Out, see, e.g., [445]

Git is a distributed Version Control System (VCS) which allows multiple users to work on the same code while preserving the history of the code changes [515, 577]. Learn more at `https://git-scm.com`.

gitee is a China-based GitHub alternative. Learn more at `https://gitee.com`.

GitHub is a website where software projects can be hosted and managed via the Git VCS [441, 577]. Learn more at `https://github.com`.

GRASP A *Greedy Randomized Adaptive Search Procedure* is a metaheuristic where each cycle consists of two main components [193, 195, 196]: First, a starting point is generated using a heuristic method (as opposed to being randomly sampled). Then, a LS is applied to refine the solution. Then, another cycle starts and proceeds in the same way. The overall best solution is retained as result. Many examples of applications of GRASPs are listed in [196]. GRASP in combination with VND has been applied to the 2BP|O|F in [435].

Greedy $(2+1)$ **GA** The Greedy $(2+1)$ GA [98, 99, 544, 545] is an EA for bit-string based search spaces, i.e., a Genetic Algorithm (GA). It was shown to be twice as fast as any mutation-only EA on OneMax [544, 545].

HTML The *Hyper Text Markup Language* is the text format used by the WWW [52, 263, 575].

HTTP The Hyper Text Transfer Protocol (HTTP) is the protocol linking web browsers to web servers in the WWW [52, 53, 197, 198, 241].

HTTPS The *Hypertext Transfer Protocol Secure* is the encrypted variant of Hyper Text Transfer Protocol (HTTP) where data is sent over Transport Layer Security (TLS) [198, 533].

I/O In computer science, the I/O stands for *input/output*, which refers to receiving and transmitting information. The term is commonly used as reading data from files (input) and writing data to files (output). Another common usage is receiving data over a network (input) or sending data over a network (output).

IBL The *improved bottom left* heuristic is a one-shot algorithm constructing a solution for 2BP [360]. For the 2BP|O|F problem variant, the input of the IBL algorithm is a permutation of the first n natural numbers, each number identifying one rectangle to be packed. For the two-dimensional Bin Packing Problem with free cuts and rotation (2BP|R|F) problem flavor, the permutation elements are signed, where a negative sign indicates a rotation by 90° [360]. In [657, 658] it is adapted to a multi-bin situation as follows: The algorithm iterates over the (potentially signed) permutation x and places the rectangles into the packing plan in the prescribed order. It first places the rectangle outside on top of the current bin b, with its bottom-right corner onto the top-right corner of the current bin b. It then moves the rectangle downwards and leftwards as far as possible, prioritizing the downward movement whenever possible. If the rectangle cannot be moved any further, it checks if it is completely contained in the bin b. If yes, it can remain there. Otherwise, a new bin is opened ($b \leftarrow b+1$) and the rectangle is placed at its bottom-left corner. Once a new bin is opened, this bin is used for all further insertions. IBL is suitable as a decoding function γ inside an EA [360]. Historically, it improves upon the BL algorithm [294]: That algorithm cannot represent all possible optimal packings using a permutation, but the IBL can [360].

IDE An *Integrated Developer Environment* is a program that allows the user do multiple different activities required for software development in one single system. It often offers functionality such as editing source code, debugging, testing, or interaction with a distributed version control system. For `Python`, we recommend using `PyCharm`. On Apple systems, `Xcode` is often used.

ILS An *iterated local search* is a variant of the LS which applies cycles of local search and perturbation [543]. First, an initial solution is generated and refined by the local search. Then the cycle begins: A perturbation, i.e., a large search move that likely destroys a good part of the solution, is applied. The resulting new solution is refined by local search. Then, an acceptance criterion decides whether we should use this or the previous solution to continue in the next cycle. ILS has successfully been applied to the QAP.

IME Input Method Editor [313].

infinitesimal an infinitesimal value is immeasurably or incalculably small, i.e., arbitrarily close to but greater than zero [181, 312].

iOS is the operating system that powers Apple iPhones [104, 519]. Learn more at `https://www.apple.com/ios`.

iPadOS is the operating system that powers Apple iPads [104]. Learn more at `https://www.apple.com/ipados`.

irrational An *irrational* number z is a real number that cannot be expressed as fraction $\frac{p}{q}$ of two integers $p \in \mathbb{Z}$ and $q \in \mathbb{N}_1$ [47, 624]. In other words, it holds that $z \in \mathbb{R}$ and $z \notin \mathbb{Q}$. The set of irrational numbers could be denoted as $\mathbb{R} \setminus \mathbb{Q}$. Irrational numbers have decimal expansions that neither terminate nor become periodic. Examples of irrational numbers are $\sqrt{2}$, ϕ, π, and e.

Ising Model The Ising models are benchmark problems for bit-string based black-box search [164, 200, 201, 654]. Here, a neighborhood is defined around each bit. The bit needs to have the same value as all bits in the neighborhood structure T. Thus, we try to achieve that all bits have the same value. The objective function $f(x)$ of the minimization variant of this problem family can be defined as $f(x,T) = \sum_{(i,j)\in T} |x_i - x_j|$, where T contains the index pairs of neighboring bits and is defined differently for the Ising model instances Ising 1D and Ising 2D. The problems are highly epistatic, as all variables interact with some other variables.

Ising 1D The 1D is a benchmark problem for bit-string based black-box search [164, 201, 654]. The bits are arranged in a one-dimensional ring and we try to minimize the number of bits that have different values than their neighbors. Hence, the neighborhood structure T is defined as $T_{1D} = \{(i+1, (i+1 \bmod n)+1) \mid i \in [0..n-1]\}$. The problems are highly epistatic, as all variables interact with some other variables. The $(1+1)$ EA has an expected runtime in $\mathcal{O}(n^3)$ on 1-Dimensional Ising Model (Ising 1D) [201].

Ising 2D The 2D is a benchmark problem for bit-string based black-box search [164, 200, 654]. The bits are arranged in a two-dimensional torus and we try to minimize the number of bits that have different values than their neighbors. Hence, the neighborhood structure $T = T_{2D}$ is defined as the set of all $\{(\alpha+\beta N, \gamma+\delta N)\}$ pairs with $\alpha, \beta, \gamma, \delta \in 0..N-1$ and either ()$\gamma = (\alpha+1) \bmod N$ and $\delta = \beta$) or ($\gamma = \alpha$ and $\delta = (\beta+1) \bmod N$) with $n = N^2$. The problems are highly epistatic, as all variables interact with some other variables. On a similar class of problems, the $(1+1)$ EA has an exponential expected runtime [200].

IT information technology

Java is another very successful programming language, with roots in the C family of languages [61, 368].

JavaScript JavaScript is the predominant programming language used in websites to develop interactive contents for display in browsers [180].

JSON *JavaScript Object Notation* is a data interchange format [74, 556] based on JavaScript [180] syntax.

JSSP The *Job Shop Scheduling Problem* [60, 115, 238, 342] is one of the most prominent and well-studied scheduling tasks. In a JSSP instance, there are m machines and k jobs [547]. Each job must be processed once by each machine in a job-specific sequence and has a job-specific processing time on each machine. The goal is to find an assignment of jobs to machines that results in an overall shortest makespan, i.e., the schedule (Gantt chart) which can complete all the jobs in the shortest time. The JSSP is $\mathcal{NP}$-complete [110, 341, 342]. The 8 most common sets of JSSP benchmark instances are *abz* [3], *dmu* [151], *ft* [202], *la* [344], *orb* [16], *swv* [539], *ta* [547], and *yn* [643].Their data can be found (sometimes partially) in several repositories, such as the OR-Library [41, 42], [509], and [613], the latter of which also contains many results from existing works. Encodings based on permutations with repetition are proposed in [58, 59, 219, 508] and used in [620]. The SOTA of the JSSP includes [230, 258, 434, 439, 510, 603, 655].

Jump The Jump problem is a benchmark problem for bit-string based black-box search [172, 209, 654]. Like the OneMax, it tries to maximize the number of 1 bits in a bit string of chosen length n. However, the solutions representing the last ω bits around the optimum to flip are deceptive, meaning that they are actually worse and getting worse than surrounding solutions. The objective function $f(x)$ of the minimization variant is $n - \|x\|_1$ if $\|x\|_1 = n$ or $\|x\|_1 \leq n - \omega$ hold, and $\omega + \|x\|_1$ otherwise, where $\|x\|_1 = \sum_{i=1}^{n} x_i$. This means that, the larger ω, the larger the deceptive basin around the optimum that an algorithm needs to cross. An $(1+1)$ EA will converge to the resulting local optimum and thus have an expected runtime of $\Theta(n^\omega + n \ln n)$ [172].In experiments, the $(1+1)$ EA with FFA ($(1+1)$ FEA) achieves polynomial runtime on Jump, seemingly in $\mathcal{O}(n^2 \ln n)$ [620, 621, 654].

k the number of jobs in an instance of the JSSP.

LABS *Low-Autocorrelation Binary Sequences* are bit strings of length n whose autocorrelation is as low as possible [68, 69, 77, 157, 158, 214, 392, 397, 405, 428, 456]. This means that sub-sequences of any length $1 \leq k < n$ should ideally never repeat. The objective function $f : \{0,1\}^n \mapsto \mathbb{N}_0$ represents the energy of a bit string x and is subject to minimization. We can define it as $f(x) = \sum_{k=1}^{n-1} \left(\sum_{i=0}^{n-1-k} s_i \cdot s_{i+k} \right)^2$ with $s_j = 2x_j - 1$. Such sequences are needed for communication engineering [69, 77, 405], cryptography[157, 405], as well as modulation pulses in radar and sonar ranging[428]. The LABS is $\mathcal{NP}$-hard [572]. The SOTA on LABS contains [68, 77, 157, 158, 405].

LAMP Stack A system setup for web applications: `Linux`, Apache (a web server), `MySQL`, and the server-side scripting language PHP [94, 255].

LeadingOnes The LeadingOnes problem is a benchmark problem for bit-string based black-box search [485, 632, 654]. It tries to maximize the number of *leading* 1 bits in the bit string of a chosen dimension n. The objective function $f(x)$ of the minimization variant of this problem can be defined as $f(x) = n - \sum_{i=1}^{n} \prod_{j=1}^{i} x_j$. Different from OneMax, the contribution to the objective value of the bit at index i thus depends on the state of all the bits *before* this index. The problem is unimodal, i.e., has only a single global optimum and no local optimum. Different from OneMax, it is not separable, i.e., we can optimize each variable separately. The expected runtime of $(1+1)$ EA on LeadingOnes is in $\Theta(n^2)$ [172]. The average runtime $(1+1)$ FEA on LeadingOnes seems to be slower by factor proportional to n, i.e., seems to be in $\mathcal{O}(n^3)$ [620].

LibreOffice is on open source office suite [215, 356, 498] which is a good and free alternative to Microsoft Office. It offers software such as LibreOffice `Writer`, LibreOffice `Calc`, and LibreOffice `Base`. See [611] for more information and installation instructions.

LibreOffice Base is a DBMS that can work on stand-alone files but also connect to other popular relational databases [191, 498]. It is part of LibreOffice [215, 356, 498] and has functionality that is comparable to Microsoft `Access` [48, 116, 581].

LibreOffice Calc is a spreadsheet software that allows you to arrange and perform calculations with data in a tabular grid. It is a free and open source spread sheet software [356, 498], i.e., an alternative to Microsoft `Excel`. It is part of LibreOffice [215, 356, 498].

LibreOffice Writer is a free and open source text writing program [652] and part of LibreOffice [215, 356, 498]. It is a good alternative to Microsoft `Word`.

linter A linter is a tool for analyzing program code to identify bugs, problems, vulnerabilities, and inconsistent code styles [304, 488]. `Ruff` is an example of a linter used in the `Python` world.

Linux is the leading open source operating system, i.e., a free alternative to Microsoft `Windows` [27, 253, 511, 573, 596]. We recommend using it for this course, for software development, and for research. Learn more at `https://www.linux.org`. Its variant `Ubuntu` is particularly easy to use and install.

literal A literal is a specific concrete value, something that is written down as-is [346, 562]. In `Python`, for example, `"abc"` is a string literal, `5` is an integer literal, and `23.3` is a `float` literal. In contrast, `sin(3)` is not a literal. Also, while `5` is an integer literal, if we create a variable `a = 5` then `a` is

not a literal either (it is a variable). Hence, literals are values that the Python interpreter reads directly from the source code and creates as objects in memory. They are not something that is the result from a computation or the result of a variable lookup. Python supports some type hints for literals, including the type `LiteralString` for string literals and the type `Literal[xyz]` for arbitrary literals `xyz`.

LK The *Lin-Kernighan* (LK) algorithm [358] is a stochastic local search (SLS) method for the TSP. The LK algorithm performs iterative k-opt moves, i.e., in each step, it constructs the two sets X and Y of k edges such that removing the edges in X from the current tour T and then adding the edges from Y to T results in a better tour.The LK algorithm is one of the most classical approaches to the TSP.

LKH The *Lin-Kernighan-Helsgaun* algorithm [131, 256, 257] is a SLS method for the TSP based on the Lin-Kernighan (LK) algorithm [358]. LKH-1 was proposed in [257] and LKH-2 in [256]. It holds the record in the *World TSP* [132] as of 2025.

LLM Large Language Model, see, e.g., an AI technique using (large/deep) neural networks to models trained on huge bodies of text to predict the next word in a text or conversation and continue to do so until a request is answered [388]. Typical examples include DeepSeek [248] and GPT [466].

LS A *local search* is an algorithm where a single "current" candidate solution is maintained and new solutions are selected from its neighborhood in each step. Most LSes are randomized, i.e., stochastic local search (SLS).

m the number of machines in an instance of the JSSP.

MA A *Memetic Algorithm* is an EA where each new solution is refined by a LS before it enters the population [612]. As LS, we can use, e.g., RLS, but Tabu Search (TS) [113, 203] is maybe a better choise. Some SOTA algorithms in some domains may be classified as MAs, e.g., Lin-Kernighan Helsgaun (LKH) basically uses a population that is refined via a local search. MAs also worked well on the JSSP [113, 258] and the TSP [203].

macOS or Mac OS is the operating system that powers Apple Mac(intosh) computers [484, 519]. Learn more at `https://www.apple.com/macos`.

mantissa See significand.

MariaDB An open source relational database management system that has forked off from `MySQL` [18, 19, 30, 177, 376, 472]. See `https://mariadb.org` for more information.

Matplotlib is a `Python` package for plotting diagrams and charts [279, 281, 299, 429]. Learn more at at `https://matplotlib.org` [281].

MaxSAT The goal of satisfiability problems is to find an assignment for n Boolean variables that make a given Boolean formula $F : \{0,1\}^n \mapsto \{0,1\}$ become true. In the *Maximum Satisfiability* problem (MaxSAT) [268], F is given in conjunctive normal form, i.e., the variables appear either directly or negated in m "or" clauses, which are all combined into one "and." The objective function $f(x)$, subject to minimization, computes the number of clauses which are false under the variable setting x. If $f(x) = 0$, then all clauses of F are true, which solves the problem. The MaxSAT problem is $\mathcal{NP}$-complete [129] and exhibits low epistasis but deceptiveness [469], making it hard for local searches. Typical benchmark instances can be found in SATLIB, which contains smaller-scale instances, and the up-to-date MaxSAT Evaluation (MSE), providing SOTA benchmarks, algorithms, and results.

$\mathrm{mean}(A)$ The *arithmetic mean* $\mathrm{mean}(A)$ is an estimate of the expected value of a distribution from which a data sample A was, well, sampled. Its is computed on data sample $A = (a_0, a_1, \ldots, a_{n-1})$ as the sum of all n elements a_i in the sample data A divided by the total number n of values, i.e., $\mathrm{mean}(A) = \frac{1}{n}\sum_{i=0}^{n-1} a_i$. In Python, you can use function `mean` from the module `statistics` to compute the arithmetic mean.

metaheuristic A *metaheuristic* is an optimization algorithm which performs a cycle of trial and error during which candidate solutions are sampled from the solution space $\mathbb{Y}$ and evaluated using an objective function $f : \mathbb{Y} \mapsto \mathbb{R}$ [224, 612]. The results of the objective function are used to

decide which solution is promising. The neighborhoods around promising solutions *or* similarly-structured solutions are then sampled in the next cycle. Prominent metaheuristics are SLSes, Simulated Annealing (SA), TS, and EAs.

Microsoft Access is a DBMS that can work on DBs stored in single, stand-alone files but also connect to other popular relational databases [48, 116, 385, 581]. It is part of Microsoft Office. A free and open source alternative to this commercial software is LibreOffice `Base`.

Microsoft Excel is a spreadsheet program that allows users to store, organize, manipulate, and calculate data in tabular structures [63, 240, 336]. It is part of Microsoft Office. A free alternative to this commercial software is LibreOffice `Calc` [356, 498].

Microsoft Office is a commercial suite of office software, including Microsoft `Excel`, Microsoft `Word`, and Microsoft `Access` [336]. LibreOffice is a free and open source alternative.

Microsoft Windows is a commercial proprietary operating system [71]. It is widely spread, but we recommend using a `Linux` variant such as `Ubuntu` for software development and for our course. Learn more at `https://www.microsoft.com/windows`.

Microsoft Word is one of the leading text writing programs [166, 396, 652] and part of Microsoft Office. A free alternative to this commercial software is the LibreOffice `Writer`.

ML Machine Learning, see, e.g., [506]

modulo division is, in `Python`, done by the operator `%` that computes the remainder of a division. `15 % 6` gives us `3`. Modulo division is mentioned in Section 3.2.

moptipy is the *Metaheuristic Optimization in `Python`* library [619]. It has been used in several different research works, including [112, 353–355, 569, 641, 657, 658]. Algorithms implemented by `moptipy` include EAs, $(1+1)$ EA, RLS, SA, and the $(1+1)$ FEA, amongst others. Learn more at `https://thomasweise.github.io/moptipy` and `https://thomasweise.github.io/moptipyapps`.

MSE The *MaxSAT Evaluation* is a regularly updated set of benchmark instances for the MaxSAT [291]. It also includes solver software and benchmark results [597].

Mypy is a static type checking tool for `Python` [347] that makes use of type hints. Learn more at `https://github.com/python/mypy` or in Section 4.4.

MySQL An open source relational database management system [70, 177, 475, 553, 634]. `MySQL` is famous for its use in the `LAMP` Stack. See `https://www.mysql.com` for more information.

n We usually use the value $n \in \mathbb{N}_0$ to denote the *scale*, i.e., the size of the problem to solve. This notion is very general in computer science and not limited to optimization. For example, if we sort a list or search in a list, the scale n is the length of the list, i.e., the number of elements in it. In a TSP, this may be the number of cities. In a discrete benchmark problem such as OneMax or Trap, this may be the number of bits over which the problem is defined. In a QAP, this may be the number of facilities and locations. In a MaxSAT problem, this may be the number of variables that are input to the Boolean formula that we want to become true. There are also problems where the scale is not defined straightforwardly like this. For example, in a JSSP, we have a number of machines and a number of jobs, which together define the size of the problem. But even then, the scale n relates to the size of the solution space $\mathbb{Y}$, and, hence, also of the search space $\mathbb{X}$, which usually grow exponentially with n. And this is already the main issue: The larger n, the (exponentially) larger the number of possible solutions and thus, the larger the runtime of algorithms. This is known as the "curse of dimensionality" [49, 111]. Scales n are also often used in conjunction with the $\mathcal{O}$-notation to denote algorithm complexity. For example, comparison-based sorting a list of n items takes $\Theta(n \log n)$ steps [322] (because they need $\lceil \log_2 n! \rceil$ comparisons in the worst case).

$\mathbb{N}_0$ the set of the natural numbers *including* 0, i.e., 0, 1, 2, 3, and so on. It holds that $\mathbb{N}_0 \subset \mathbb{Z}$ and that $\mathbb{N}_0 = \mathbb{N}_1 \cup \{0\}$.

$\mathbb{N}_1$ the set of the natural numbers *excluding* 0, i.e., 1, 2, 3, 4, and so on. It holds that $\mathbb{N}_1 \subset \mathbb{Z}$ and that $\mathbb{N}_1 = \mathbb{N}_0 \setminus \{0\}$.

N-Queens The N-Queens Problem is a benchmark problem for bit-string based black-box search [164, 654]. The goal is to arrange N queens on an $N \times N$ chessboard so that none can attack the other. The search space has thus dimension $n = N^2$, with each bit standing for a field and encoding whether or not a queen is placed there. The objective function $f(x)$ of the minimization variant is $N - (\sum_{i=1}^{n} x_i) - N \sum_{\xi} \max\{0, Q_\xi(x) - 1\}$ where $Q_\xi(x)$ is the number of queens in the horizontal, vertical, or diagonale ξ. The problem is quite epistatic.

$\mathcal{NP}$ is the class of computational problems that can be solved in polynomial time by a non-deterministic machine and can be verified in polynomial time by a deterministic machine (such as a normal computer) [234].

$\mathcal{NP}$-complete A decision problem is $\mathcal{NP}$-complete if it is in $\mathcal{NP}$ and all problems in $\mathcal{NP}$ are reducible to it in polynomial time [234, 467]. A problem is $\mathcal{NP}$-complete if it is $\mathcal{NP}$-hard and if it is in $\mathcal{NP}$.

$\mathcal{NP}$-hard A problem is $\mathcal{NP}$-hard if all problems in $\mathcal{NP}$ are reducible to it in polynomial time [234]. Algorithms that guarantee to find the correct solutions of $\mathcal{NP}$-hard problems [110, 129, 342] have a worst-case runtime that grows exponentially with the problem scale.

numerator The number a of a fraction $\frac{a}{b} \in \mathbb{Q}$ is called the *numerator*.

NumPy is a fundamental package for scientific computing with `Python`, which offers efficient array data structures [152, 252, 299]. Learn more at `https://numpy.org` [418].

$\boldsymbol{\Omega}(g(x))$ If $f(x) = \boldsymbol{\Omega}(g(x))$, then there exist positive numbers $x_0 \in \mathbb{R}^+$ and $c \in \mathbb{R}^+$ such that $f(x) \geq c * g(x) \geq 0$ for all $x \geq x_0$ [133, 319, 320]. In other words, $\boldsymbol{\Omega}(g(x))$ describes a lower bound for function growth.

$\mathcal{O}(g(x))$ If $f(x) = \mathcal{O}(g(x))$, then there exist positive numbers $x_0 \in \mathbb{R}^+$ and $c \in \mathbb{R}^+$ such that $0 \leq f(x) \leq c * g(x)$ for all $x \geq x_0$ [23, 133, 319, 320, 338]. In other words, $\mathcal{O}(g(x))$ describes an upper bound for function growth.

$\boldsymbol{\Theta}(g(x))$ If $f(x) = \boldsymbol{\Theta}(g(x))$, then $f(x) = \mathcal{O}(g(x))$ and $f(x) = \boldsymbol{\Omega}(g(x))$ [319, 320], meaning that there exist positive constants $c_1, c_2, x_0 \in \mathbb{R}^+$ such that $0 \leq c_1 g(x) \leq f(x) \leq c_2 g(x)$ for all $x \geq x_0$ [133]. In other words, $\boldsymbol{\Theta}(g(x))$ describes an exact order of function growth.

$\mathbf{o}(g(x))$ If $f(x) = \mathbf{o}(g(x))$, then there exist positive numbers $x_0 \in \mathbb{R}^+$ such that $0 \leq f(x) \leq c * g(x)$ for all $x \geq x_0$ *all* $c \in \mathbb{R}^+$ [133]. Different from $\mathcal{O}(g(x))$, $\mathbf{o}(g(x))$ holds for *all* constants $c > 0$, whereas $\mathcal{O}(g(x))$ implies the existence of *some* constant $c > 0$. This basically means that $f(x)$ becomes insignificant compared to $g(x)$ as x_0 grows toward infinity. It is often written that $\lim_{x_0 \to \infty} \frac{f(x)}{g(x)} = 0$. In other words, $\mathbf{o}(g(x))$ describes an upper bound for function growth that is not asymptotically tight, i.e., an upper bound that grows faster than $f(x)$.

OneMax OneMax is the simplest possible benchmark problem for bit-string based black-box search [173, 406, 654]. The goal is to maximize the number of `1` bits in the bit string of a chosen dimension n. The objective function $f(x)$ of the minimization variant of this problem can be defined as $f(x) = n - \sum_{i=1}^{n} x_i$. The problem is unimodal, i.e., has only a single global optimum and no local optimum. It is fully separable, i.e., we can optimize each variable separately. The expected runtime of $(1+1)$ EA on OneMax is in $\boldsymbol{\Theta}(n \ln n)$ [162, 406]. The average runtime $(1+1)$ FEA on OneMax seems to be slower by factor proportional to n, i.e., seems to be in $\mathcal{O}(n^2 \ln n)$ [620].

OOP Object-Oriented Programming [492]

OR Operations Research (or Operational Research) is the application of sciences such as mathematics and computer science to the management and organization of systems, organizations, enterprises, factories, or projects. It encompasses the development and application of problem-solving methods and techniques (such as mathematical optimization, simulation, queueing theory and other stochastic models) with the goal to improve decision-making and efficiency [148].

OR-Library is a library of benchmark instances for various OR problems [41, 42] located at `https://people.brunel.ac.uk/~mastjjb/jeb/info.html`. It contains instances of the JSSP, 1D- and 2D-Packing, Knapsack, periodic TSPs, amongst others.

OS Operating System, the system that runs your computer, see, e.g., Linux, Microsoft Windows, macOS, and Android.

OSS Open source software, i.e., software that can freely be used, whose source code is made availabe in the internet, and which is usually developed cooperatively over the internet as well [266]. Typical examples are Python, Linux, Git, and PostgreSQL.

package A Python package is basically a directory containing Python files. This allows us to group functionality together as a library that can be used by different applications. Many popular Python packages are offered as open source at PyPI and can be installed with pip. We discuss this in Section 14.1.

Pandas is a Python data analysis and manipulation library [43, 351]. Learn more at https://pandas.pydata.org [432].

PDF The *Portable Document Format* [205, 630] is the format in which provide this book. It is the standard format for the exchange of documents in the internet.

PDF *Probability Density Function* [2, 190, 607] of a continuous random variable $\mathcal{X}$ that can take on values from range $\Omega \subseteq \mathbb{R}$ is a function $f_{\mathcal{X}} : \Omega \mapsto \mathbb{R}^+$ such that

- $f_{\mathcal{X}}(x) > 0$ for all $x \in \Omega$ (non-negativity),
- $\int_{\Omega} f_{\mathcal{X}}(x)dx = 1$ (normalization), and
- the probability that $x \in A$ for all $A \subseteq \Omega$ is $\int_A f_{\mathcal{X}}(x)dx$.

pip is the standard tool to install Python software packages from the PyPI repository [286, 448]. To install a package `thepackage` hosted on PyPI, type `pip install thepackage` into the terminal. Learn more at https://packaging.python.org/installing.

Plateau The Plateau problem is a benchmark problem for bit-string based black-box search [13, 654]. It is structured exactly like Jump, but instead of being deceptive, the last ω bits to flip around the optimum are neutral. The objective function $f(x)$ of the minimization variant is $n - \|x\|_1$ if $\|x\|_1 = n$ or $\|x\|_1 \leq n - \omega$ hold, and ω otherwise, where $\|x\|_1 = \sum_{i=1}^{n} x_i$. This means that, the larger ω, the larger the neutral plane around the optimum that an algorithm needs to cross. An $(1+1)$ EA has an expected runtime of $\Theta(n^\omega)$ on plateaus [13]. The average runtime of the $(1+1)$ FEA on Plateaus is slower by a factor seemingly proportional in n.

PostgreSQL An open source object-relational DBMS [194, 421, 452, 553]. See https://postgresql.org for more information.

psql is the client program used to access the PostgreSQL DBMS server.

psycopg or, more exactly, `psycopg 3`, is the most popular PostgreSQL adapter for Python, implementing the Python DB API 2.0 specification [349]. Learn more at https://www.psycopg.org [598].

PyCharm is the convenient Python IDE that we recommend for this course [586, 636, 644]. It comes in a free edition, so it can be downloaded and used at no cost. Learn more at https://www.jetbrains.com/pycharm.

Pylint is a linter for Python that checks for errors, enforces coding standards, and that can make suggestions for improvements [461]. Learn more at https://www.pylint.org and Useful Tool 6 and in [615]

PyPI The Python Package Index (PyPI) is an online repository that provides the software packages that you can install with pip [67, 559, 584]. Learn more at https://pypi.org.

pytest is a framework for writing and executing unit tests in Python [154, 329, 425, 430, 636]. Learn more at https://pytest.org and in Section 8.3 and in [615]

Python The Python programming language [278, 345, 371, 615], i.e., what you will learn about in our book [615]. Learn more at https://python.org.

PyTorch is a Python library for deep learning and AI [436, 470]. Learn more at https://pytorch.org.

$\mathbb{Q}$ the set of the rational numbers, i.e., the set of all numbers that can be the result of $\frac{a}{b}$ with $a, b \in \mathbb{Z}$ and $b \neq 0$. a is called the numerator and b is called the denominator. It holds that $\mathbb{Z} \subset \mathbb{Q}$ and $\mathbb{Q} \subset \mathbb{R}$.

QAP The *Quadratic Assignment Problem* is an optimization problem where the goal is to assign a set of n facilities to a set of n locations [90, 112, 325, 364]. Such an assignment can be represented as a permutation x of the first n natural numbers, where x_i specifies the location where facility i should be placed. For each QAP instance, a distance matrix A is given, where $A[a, b]$ specifies the distance from location a to location b, as well as a flow matrix B, where $B[i, j]$ is the amount of material flowing (e.g., daily) from facility i to facility j. The objective function $f(x)$ then rates a permutation x as $f(x) = \sum_{i=1}^{n} \sum_{j=1}^{n} A[x_i, x_j] * B[i, j]$. The QAP is $\mathcal{NP}$-complete [171, 491] and even among this problem class, is considered as particularly hard for metaheuristics [46, 90, 364, 543]. The QAP has applications such as building layout [139, 182, 327], keyboard layout [92], circuit design [185], wiring [534], and scheduling [524]. Typical benchmark instances can be found in QAPLIB. SOTA algorithms for the QAP include [46, 203, 393, 401, 549, 550].

QAPLIB is a library of 136 benchmark instances for the Quadratic Assignment Problem (QAP) [89, 91], 28 of which have not yet been solved to optimality [11] (at least by 2023).

$\mathbb{R}$ the set of the real numbers.

$\mathbb{R}^+$ the set of the positive real numbers, i.e., $\mathbb{R}^+ = \{x \in \mathbb{R} : x > 0\}$.

Random Keys encoding is a decoding procedure $\gamma : \mathbb{X} \mapsto \mathbb{Y}$ that maps a continuous search space $\mathbb{X} \subset [0, 1]^n$ which is a subset of the n-dimensional real numbers to the space $\mathbb{Y}$ of permutations of the first n natural numbers [39, 40]. For example, on $n = 5$, the vector $x = (0.49, 0.91, 0.33, 0.75, 0.51)$ is translated to the permutation $y = (3, 1, 5, 4, 2)$, because $x_3 = 0.33$ has the smallest value, putting 3 at the beginning, $x_1 = 0.49$ the second-smallest, placing 1 second, ..., and $x_2 = 0.91$ being the largest value meaning that the last number in y will be 2 [40]. The advantage of the encoding is that we can search in $\mathbb{X}$ applying arbitrary mathematical operators, crossover, and mutation procedures, but will always get correct permutations. The disadvantage is that $|\mathbb{X}| \gg |\mathbb{Y}|$ and that there actually is no meaning underpinning moves in $\mathbb{X}$ in relation to their projected result in $\mathbb{Y}$.

regex A *Regular Expression*, often called "regex" for short, is a sequence of characters that defines a search pattern for text strings [283, 331, 407, 411]. In `Python`, the `re` module offers functionality work with regular expressions [331, 473]. In `PostgreSQL`, regex-based pattern matching is supported as well [450].

relational database A relational DB is a database that organizes data into rows (tuples, records) and columns (attributes), which collectively form tables (relations) where the data points are related to each other [123, 246, 251, 522, 551, 611, 631].

RLS Randomized local search retains the best solution x_c discovered so far during the search and, in each step, it applies a unary search operator to this best-so-far solution x_c and derives a new solution x_n. If the new solution x_n is *better or equally good* when compared with x_c, i.e., not worse, then it replaces it, i.e., is stored as the new x_c. If the search space are bit strings of length n, then RLS uses a unary search operator that flips exactly one bit. This operator is the main difference to $(1 + 1)$ EA.

FRLS The RLS with FFA plugged in [654]. On the discrete bit-string domains, this algorithm has a very specific structure: Its unary search operator flips exactly one bit, distinguishing it from the unary operator in the $(1 + 1)$ FEA, which flips each bit with probability $1/n$. On other domains, such as permutations, we use the term synonymously with $(1 + 1)$ FEA.

RTS *Reactive Tabu Search* is a variant of TS where the length of the tabu list is adapted to the problem and to the current evolution of the search, and an escape strategy for diversifying the search is used when that is not sufficient [34–37]. It has successfully been applied to a wide range of problems, from the QAP [35, 36], to Knapsack problems [34, 36], the N-K landscape [34] to the training of neural networks [37].

Ruff is a linter and code formatting tool for `Python` [377, 378]. Learn more at `https://docs.astral.sh/ruff` or in Useful Tool 5.

SA *Simulated Annealing* is a SLS that sometimes accepts a worse solution [105, 293, 315, 446, 612]. The probability to do so decreases over time and with the difference in objective values, i.e.,is the lower the worse the new solution is. SA performed well on a variety of problems, including the QAP [399, 570] and the JSSP [434].

SATLIB is a library of benchmark instances for the Maximum Satisfiability (MaxSAT) available at `https://www.cs.ubc.ca/~hoos/SATLIB/index-ubc.html` [269, 270].

Scikit-learn is a `Python` library offering various machine learning tools [438, 470]. Learn more at `https://scikit-learn.org`.

SciPy is a `Python` library for scientific computing [299, 604]. Learn more at `https://scipy.org`.

Self-Adjusting $(1+(\lambda,\lambda))$ **GA** The Self-Adjusting $(1+(\lambda,\lambda))$ GA is a theory-inspired EA for bit string-based search spaces [99]. It has linear runtime on the OneMax problem [161].

server In a client-server architecture, the server is a process that fulfills the requests of the clients. It usually waits for incoming communication carring the requests from the clients. For each request, it takes the necessary actions, performs the required computations, and then sends a response with the result of the request. Typical examples of servers are web servers [94] in the internet as well as DBMSes. It is also common to refer to the computer running the server processes as server as well, i.e., to call it the "server computer" [333].

SGA The Standard Genetic Algorithm [24, 147, 228, 395, 402, 612] was the first EA. It maintains a population of solutions and applies mutation and crossover to generate offspring solutions. It uses fitness proportionate selection to choose which solutions should "survive" into the next generation, which today is considered a very bad design choice, see, e.g., [632].

sign bit The sign bit indicates whether a floating point number is positive or negative in the 64 bit double precision IEEE Standard 754 floating point number layout [267, 282]. See Section 3.3.1.

signature The signature of a function refers to the parameters and their types, the return type, and the exceptions that the function can raise [387]. In `Python`, the function `signature` of the module `inspect` provides some information about the signature of a function [93].

significand The significand is the part of a floating point number that stores the digits of the number (in binary representation). In the 64 bit double precision IEEE Standard 754 floating point number layout [267, 282], the exponent is 52 bits. See Section 3.3.1.

SimPy is a `Python` library for discrete event simulation [661]. Learn more at `https://simpy.readthedocs.io`.

SLS *Stochastic local search* is a local search (LS), one of the simplest metaheuristic search paradigms [268]: A single "current" candidate solution is maintained and new solutions are *stochastically* sampled from its neighborhood, i.e., in a way that involves random numbers. If these solutions are deemed interesting based on their objective value, they replace the "current". The cycle continues. Basic examples are RLS and $(1+1)$ EA. More complex examples that add mechanisms to escape local optima on top of the SLS paradigm are TS and SA.

SOTA State-of-the-Art, i.e., the best-performing methods for a field or problem type

Sphinx `Sphinx` is a tool for generating software documentation [579]. It supports `Python` can use both docstrings and type hints to generate beautiful documents. Learn more at `https://www.sphinx-doc.org`.

SQL The *Structured Query Language* is basically a programming language for querying and manipulating relational databases [107, 143, 145, 146, 284, 390, 527, 535, 536, 551]. It is understood by many DBMSes. You find the SQL commands supported by `PostgreSQL` in the reference [527].

SQLi attack A SQL injection attack is an attack that is used to target data stored in DBMS by injecting malicious input into code that constructs SQL queries by string concatenation in order to subvert application functionality and perform unauthorized operations [137, 332, 437, 495, 611]. In order to prevent such attacks, queries to DBs should *never* be constructed via string concatenation or the likes of `Python` f-strings. Assume that `user_id` was a string variable in a `Python`

program and we construct the query `f"SELECT * FROM data WHERE user_id = {user_id}"`. Notice that the `{user_in}` will be replaced with the value of variable `user_id` during (string) interpolation. If `user_id == "user123; DROP TABLE data;"`, mayhem would ensue when we execute the query [528]. Some programming languages, like Python, offer built-in datatypes (such as `LiteralString` [528]) to annotate string constants that can be used by static type-checkers. At the time of this writing, Mypy does not support this yet [599, 660].

SQLite is an relational DBMS which runs as in-process library that works directly on files as opposed to the client-server architecture used by other common DBMSes. It is the most wide-spread SQL-based DB in use today, installed in nearly every smartphone, computer, web browser, television, and automobile [107, 212, 265, 637]. Learn more at `https://sqlite.org` [526].

SSH Secure Shell Transport Layer Protocol, a protocol that provides users a secure way to access computers over an unsecure network [28, 103, 650].

stack trace A stack trace gives information the way in which one function invoked another. The term comes from the fact that the data needed to implement function calls is stored in a stack data structure [320]. The data for the most recently invoked function is on top, the data of the function that called is right below, the data of the function that called that one comes next, and so on. Printing a stack trace can be very helpful when trying to find out where an `Exception` occurred. See, for instance, Chapter 9.

`stderr` The *standard error stream* is one of the three pre-defined streams of a console process (together with the `stdin` and the `stdout`) [309]. It is the text stream to which the process writes information about errors and exceptions. If an uncaught `Exception` is raised in Python and the program terminates, then this information is written to `stderr`. If you run a program in a terminal, then the text that a process writes to its `stderr` appears in the console.

`stdin` The *standard input stream* is one of the three pre-defined streams of a console process (together with the `stdout` and the `stderr`) [309]. It is the text stream from which the process reads its input text, if any. The Python instruction `input` reads from this stream. If you run a program in a terminal, then the text that you type into the terminal while the process is running appears in this stream.

`stdout` The *standard output stream* is one of the three pre-defined streams of a console process (together with the `stdin` and the `stderr`) [309]. It is the text stream to which the process writes its normal output. The `print` instruction of Python writes text to this stream. If you run a program in a terminal, then the text that a process writes to its `stdout` appears in the console.

(string) interpolation In Python, string interpolation is the process where all the expressions in an f-string are evaluated and the final string is constructed. An example of string interpolation is turning `f"Rounded {1.234:.2f}"` to `"Rounded 1.23"`. This is discussed in Section 3.6.2.

STSP A *symmetric Traveling Salesperson Problem* is a special case of the TSP, where the $n \times n$ distance matrix A is symmetric, i.e., $A[a, b] = A[b, a] \forall a, b \in 1..n$.

sudo In order to perform administrative tasks such as installing new software under Linux, root (or "super") user privileges as needed [121]. A normal user can execute a program in the terminal as super user by pre-pending `sudo`, often referred to as "super user do." This requires the root password.

SVG The *Scalable Vector Graphics* format is an XML-based format for vector graphics [141]. Vector graphics are composed of geometric shapes like lines, rectangles, circles, and text. As opposed to raster / pixel graphics, they can be scaled seamlessly and without artifacts. They are stored losslessly.

TensorFlow is a Python library for implementing machine learning, especially suitable for training of neural networks [1, 337]. Learn more at `https://www.tensorflow.org`.

terminal A terminal is a text-based window where you can enter commands and execute them [27, 121]. Knowing what a terminal is and how to use it is very essential in any programming- or system administration-related task. If you want to open a terminal under Microsoft Windows, you can press ⊞ + R, type in `cmd`, and hit ↵. This is shown, e.g., in Figure 2.2.1. Under Ubuntu Linux, Ctrl + Alt + T opens a terminal, which then runs a Bash shell inside.

TLS Transport Layer Security, a protol for encrypted communication over the internet [174, 479, 533], used by, e.g., HTTPS.

transcendental A *transcendental* number is a number that is not the root of any integer polynomial, meaning that it is not an algebraic number of any degree [628]. Every real transcendental number must also be irrational, since a rational number is, by definition, an algebraic number of degree one [628]. Examples of transcendental numbers are π and e.

Trap The Trap problem is a benchmark problem for bit-string based black-box search [172, 412, 654]. It is structured exactly like OneMax, however the worst and the best solution are swapped. This makes the problem highly deceptive, as following the "gradient" in the search space leads the search away from the global optimum. For bit strings of length n, the objective function $f(x)$ of the minimization variant is 0 if $\|x\|_1 = 0$ and $n - \|x\|_1 + 1$ otherwise, where $\|x\|_1 = \sum_{i=1}^{n} x_i$. An $(1+1)$ EA will converge to the resulting local optimum and thus have an expected runtime of $\mathbf{\Theta}(n^n)$[172]. In experiments, the $(1+1)$ FEA achieves polynomial runtime on Trap, seemingly in $\mathcal{O}(n^2 \ln n)$ [620, 621, 654].

TS *Tabu Search* is one of the most successful SLSes [222, 223, 225, 298]. In order to avoid returning to previous solutions, it declares certain solutions or search moves as "tabu." For example, if a bit in a discrete problem has been flipped by the search operator, it may be forbidden to flip the bit again for some time, thus preventing the algorithm from getting trapped in a cycle. This rule may be overridden if a new best-so-far solution is discovered that outperforms all previous solutions, in which case it would be accepted (this is the so-called aspiration criterion). TS has performed well on the QAP [35, 36, 398, 401, 548, 549] and on the JSSP [416, 439, 444, 510, 655].

TSP In an instance of the *Traveling Salesperson Problem*, also known as *Traveling Salesman Problem*, a set of n cities or locations as well as the distances between them are given [15, 245, 343, 353, 616, 622]. The goal is to find the shortest round-trip tour that starts at one city, visits all the other cities one time each, and returns to the origin. The TSP is one of the most well-known $\mathcal{NP}$-hard combinatorial optimization problems [217, 245, 330]. The TSP has many applications[15], including in transportation, logistics, (circuit) plotting and drilling problems[242], telescope aiming, and genome sequencing. Solutions are often encoded as permutations x of the first n natural numbers, where each city gets a number and the order of numbers in the permutation corresponds to the order of cities in the tour. The distances between the n cities are usually given in a $n \times n$ distance matrix A, where $A[a,b]$ is the distance from city a to city b with $a, b \in 1..n$. Then, the objective function subject to minimization is $f(x) = A[x_n, x_1] + \sum_{i=1}^{n-1} A[x_i, x_{i+1}]$. Typical benchmark instances can be found in TSPLib [476, 477], the 8th DIMACS challenge[300–302], and the *World TSP* repository[132].

TSPLib is a library of benchmark instances for the Traveling Salesperson Problem (TSP) [476, 477]. Its original URL was available until 2025 and then became defunct. The repository `https://github.com/shredderzwj/TSPLIB` contains the data. Additionally, you may also check `https://github.com/thomasWeise/tspSuite`, which contains all the TSPLib instances as well as Java code to load them. The TSPLib contains 110 instances of the symmetric Traveling Salesperson Problems (STSPs) with scales n ranging from 14 to 85 900, but 93 instances have less than 2000 nodes. There also are 19 asymmetric Traveling Salesperson Problems (ATSPs) with scales $n \in 17..443$.

TTP The *Traveling Tournament Problem* is the combinatorial optimization problem of both efficiently and fairly organizing a tournament of n teams that play against each other in a pairwise fashion [178, 641]. The efficient part boils down to arranging the games such that the total travel length is short, which is somewhat similar to the classical TSP. Initially, each team is at its home location. On each day, a team needs to travel if its scheduled game is not at its present location. On the last day, each team may need to travel back home unless their last game is a home game. The total travel length sums up the lengths of all travels over all teams. The fair part is represented in several constraints, such as doubleRoundRobin, compactness, maxStreak, and noRepeat. The TTP is $\mathcal{NP}$-hard [601].

Turbo Pascal is/was a programming language supporting OOP with an IDE providing a debugger and compiler that was popular in the 1990s [578].

TwoMax The TwoMax problem is a benchmark problem for bit-string based black-box search [209, 587, 654]. It has a global and a local optimum, both of about the same size, at opposite ends of the search space. For bit strings of length n, the objective function $f(x)$ of the minimization variant is 0 if $\|x\|_1 = n$ and $1 + n - \max\{\|x\|_1, n - \|x\|_1\}$ otherwise, where $\|x\|_1 = \sum_{i=1}^{n} x_i$. An $(1+1)$ EA has a 50/50 chance to converge to the local or global optimum and thus has an expected runtime in $\mathbf{\Theta}(n^n)$ [208, 209]. In experiments, the $(1+1)$ FEA achieves polynomial runtime seemingly in $\mathcal{O}(n^2 \ln n)$ on TwoMax [620, 621, 654].

type hint are annotations that help programmers and static code analysis tools such as `Mypy` to better understand what type a variable or function parameter is supposed to be [339, 592]. `Python` is a dynamically typed programming language where you do not need to specify the type of, e.g., a variable. This creates problems for code analysis, both automated as well as manual: For example, it may not always be clear whether a variable or function parameter should be an integer or floating point number. The annotations allow us to explicitly state which type is expected. They are *ignored* during the program execution. They are a basically a piece of documentation. See Section 4.4.4.

Ubuntu is a variant of the open source operating system `Linux` [121, 255]. We recommend that you use this operating system to follow this class, for software development, and for research. Learn more at `https://ubuntu.com`. If you are in China, you can download it from `https://mirrors.ustc.edu.cn/ubuntu-releases`.

UCS Universal Coded Character Set, see Unicode

Unicode A standard for assigning characters to numbers [285, 565, 582]. The Unicode standard supports basically all characters from all languages that are currently in use, as well as many special symbols. It is the predominantly used way to represent characters in computers and is regularly updated and improved.

unit test Software development is centered around creating the program code of an application, library, or otherwise useful system. A *unit test* is an *additional* code fragment that is not part of that productive code. It exists to execute (a part of) the productive code in a certain scenario (e.g., with specific parameters), to observe the behavior of that code, and to compare whether this behavior meets the specification [44, 424, 426, 430, 486, 568]. If not, the unit test fails. The use of unit tests is at least threefold: First, they help us to detect errors in the code. Second, program code is usually not developed only once and, from then on, used without change indefinitely. Instead, programs are often updated, improved, extended, and maintained over a long time. Unit tests can help us to detect whether such changes in the program code, maybe after years, violate the specification or, maybe, cause another, depending, module of the program to violate its specification. Third, they are part of the documentation or even specification of a program. See also Definition 8.4.

URI A *Uniform Resource Identifier* is an identifier for an abstract or physical resource in the internet [648]. It can be a URL, a name, or both. URIs are supersets of URLs. The connection strings of the `PostgreSQL` DBMS are examples of Uniform Resource Identifiers (URIs).

URL A *Uniform Resource Locator* identifies a resource in the WWW and a way to obtain it by describing a network access mechanism. The most notable example of URLs is the text you write into web browsers to visit websites [54]. URLs are subsets of URIs.

UTF-8 The *UCS Transformation Format 8* is one standard for encoding Unicode characters into a binary format that can be stored in files [285, 648]. It is the world wide web's most commonly used character encoding, where each character is represented by one to four bytes. It is backwards compatible with ASCII (see Section 3.6.6).

$\mathrm{var}(A)$ The *variance* of a distribution is the expectation of the squared deviation of the underlying random variable from its mean. The variance $\mathrm{var}(A)$ of a data sample $A = (a_0, a_1, \ldots, a_{n-1})$ with n observations can be estimated as $\mathrm{var}(A) = \frac{1}{n-1}\sum_{i=0}^{n-1}(a_i - \mathrm{mean}(A))^2$. It can also be computed using $\mathrm{var}(A) = \left(\sum_{i=0}^{n-1} a_i^2\right) - \frac{1}{n}\left(\sum_{i=0}^{n-1} a_i\right)^2$, which affords computing it in a single pass, but this method may lead to disastrous rounding errors [108]. A better choice is Welford's

algorithm [321, 629]: It begins by setting its running mean $m \leftarrow 0$, running variance $v \leftarrow 0$, and its running counter $i \leftarrow 0$. It then iterates over A with the current value being a_i. In each step, it sets $delta \leftarrow a_i - m$, then updates $m \leftarrow m + \delta/(i+1)$, $v \leftarrow v + \delta * (a_i - m)$, and finally $i \leftarrow i + 1$. Once all the elements of A have been processed, the sample variance is $\frac{1}{n-1}v$.

VCS A *Version Control System* is a software which allows you to manage and preserve the historical development of your program code [577]. A distributed VCS allows multiple users to work on the same code and upload their changes to the server, which then preserves the change history. The most popular distributed VCS is Git.

venv is a `Python` module and tool for creating virtual environments [600]. Learn more at `https://docs.python.org/3/library/venv.html` [600].

virtual environment A virtual environment is a directory that contains a local `Python` installation [394, 605]. It comes with its own package installation directory. Multiple different virtual environments can be installed on a system. This allows different applications to use different versions of the same packages without conflict, because we can simply install these applications into different virtual environments. See Section 14.3 for details.

VND *Variable Neighborhood Descend*; see Variable Neighborhood Search (VNS). VND in combination with GRASP has been applied to the 2BP|O|F in [435].

VNS *Variable Neighborhood Search* is a metaheuristic framework combining a systematic change of neighborhood with LS [249, 403]. A *neighborhood* structure in LS defines which solutions can be reached from a given solution within one search step. Usually, one uses a single unary operator and, hence, has a single neighborhood structure. However, for many problems, we can easily design many operators. For permutations, for example, one operator may swap two elements, another one may reverse a section of the permutation, and yet another one, while yet another one may rotate a subsequence one step to the left or right [616]. In the VNS framework, a multiple neighborhoods are investigated in a systematic fashion. Assume that we have a list of neighborhood structures. Variable Neighborhood Descend (VND) works in a deterministic fashion. It finds the best solution that can be reached in one step from the current solution under the current neighborhood. If this is better than the current solution, we accept it and continue with the first neighborhood. Otherwise, we continue with the next neighborhood in the list. In Basic VNS, instead of picking the best move from the current neighborhood, the first improving move is used. Also, at the beginning of each cycle, a random point from the current neighborhood around the current solution is chosen as starting point.

$\omega(g(x))$ If $f(x) = \omega(g(x))$, then for every positive constant $c > 0$, there exists a number $x_0 \in \mathbb{R}^+$ such that $0 \leq c * g(x) < f(x)$ for all $x \geq x_0$ [133]. In other words, $f(x) = \omega(g(x))$ if and only if $g(x) = \mathbf{o}(f(x))$. In other words, ω describes a lower bound that is not asymptotically tight.

WWW World Wide Web [52, 149]

$\mathbb{X}$ The search space $\mathbb{X}$ is the space through which an optimization algorithm navigates in order to find good solutions. The search operators of metaheuristics are used to sample points $x \in \mathbb{X}$, for example. On many benchmark problems, the objective functions f directly evaluate the elements $x \in \mathbb{X}$, i.e., are defined as $f : \mathbb{X} \mapsto \mathbb{R}$. However, very often these elements x are not the same data structures that the user is interested in and/or that the objective functions evaluate: It is common that the search space $\mathbb{X}$ is a simple and plain data structure that can be tackled with standard operations or mathematical tools, such as real vectors $\mathbb{R}^n$, permutations, or bit strings, whereas the solution space $\mathbb{Y}$ can be something more complicated. Then, the user only cares about the solutions $y \in \mathbb{Y}$ and not about the search space $\mathbb{X}$. A decoding $\gamma : \mathbb{X} \mapsto \mathbb{Y}$ is employed to translate the $x \in \mathbb{X}$ to $y = \gamma(x)$ with $y \in \mathbb{Y}$.

x a point in the search space $\mathbb{X}$.

x-axis The x-axis is the horizontal axis of a two-dimensional coordinate system, often referred to abscissa.

Xcode is offers the tools for developing, testing, and distributing applications as well as an IDE for Apple platforms such as `macOS` and `iOS` [490].

XML The *Extensible Markup Language* is a text-based language for storing and transporting of data [75, 125, 323]. It allows you to define elements in the form `<myElement myAttr="x">...text..</myElement>`. Different from CSV, elements in XML can be hierarchically nested, like `<a><b><c>test</c></b><b>bla</b></a>`, and thus easily represent tree structures. XML is one of most-used data interchange formats. To process XML in `Python`, use the `defusedxml` library [254], as it protects against several security issues.

$\mathbb{Y}$ the set of candidate solutions of an optimization problem is called the *solution space* $\mathbb{Y}$. The solution space corresponds to the data structure holding the candidate solutions y. These are the elements that are passed to the user after the optimization process is completed. They are also the parameters of the objective function(s) $f : \mathbb{Y} \mapsto \mathbb{R}$.

y a candidate solution $y \in \mathbb{Y}$ is one possible solution to an optimization problem. After the optimization process is completed, one or multiple such solutions are passed to the user.

YAML *YAML Ain't Markup Language™* is a human-friendly data serialization language for all programming languages [124, 170, 323]. It is widely used for configuration files in the DevOps environment. See `https://yaml.org` for more information.

$\mathbb{Z}$ the set of the integers numbers including positive and negative numbers and 0, i.e., ..., -3, -2, -1, 0, 1, 2, 3, ..., and so on. It holds that $\mathbb{Z} \subset \mathbb{R}$.

Python Commands

!=, 46, 63, 64, 111, 123, 281, 295
!==, 323
!r, 149, 160, 280
!s, 279
(, 33, 38, 64
(...), 121, 162
), 33, 38, 64
*, 290, 324, 325
  function parameter, 180
  list, 116
  multiplication, 32, 38
**, 179, 324, 325
  function parameter, 180
  power, 33, 38
**=, 324
*=, 89, 163, 324
+, 32, 38, 68, 116, 152, 290, 324, 325
+=, 152, 324
,, 99, 112
,.1f, 75
-, 33, 38, 290, 295, 324, 325
-=, 324
->, 162
-inf, 45, 47, 175, 190
., 169
.2f, 75, 378
.3f, 75
/, 33, 38, 46, 55, 102, 105, 290, 324, 325, 355
//, 33, 55, 102, 105, 133, 324, 325, 355
//=, 324
/=, 324
:, 68, 73, 105, 133–135, 143, 162, 184, 208, 229
:.2%, 75
:.3g, 75
:.4%, 75
:.4g, 75
:.5f, 75
:#b, 74
:#x, 74
:b, 73
:e, 75
:param:, 163
:returns:, 163
:x, 73
<, 63, 123, 295, 323
<=, 63, 123, 295, 323
<<, 35, 325, 326
<<=, 325
=, 85
  multiple, 99
  swap, 99, 100, 356
==, 45, 46, 63, 64, 81, 82, 109, 111, 116, 123, 173, 281, 295, 323
>, 63, 123, 136, 295, 323
>=, 63, 123, 295, 323
>>, 35, 325, 326
>>=, 325
>>>, 227, 362, 367
@, 324, 325
@=, 324
[...], 68, 112, 116, 128, 324–326
[], 113
[i:j:k], 116
[i:j], 68, 116
#, 84, 355
#: , 254, 292, 358, 359
%, 33, 38, 133, 148, 164, 299, 324, 325
%=, 324
&, 34, 325, 326
&=, 325
^, 34
_, 44, 145, 183, 355, 357
__, 264, 266, 278, 323–326, 359
__abs__, 294, 295, 324, 325
__add__, 292, 293, 295, 324, 325
__and__, 325
__bool__, 77, 323
__bytes__, 323
__call__, 323
__ceil__, 51, 52, 324
__complex__, 323
__contains__, 323, 325
__delitem__, 323, 325
__divmod__, 324
__enter__, 316, 317, 323
__eq__, 281–283, 285–288, 295, 297, 323, 324, 359
__exit__, 316–318, 323
__float__, 323
__floor__, 324
__floordiv__, 324
__format__, 323
__ge__, 295, 297, 323, 324
__getitem__, 323, 325
__gt__, 295, 297, 323, 324
__hash__, 285–288, 323, 359
__hash]__, 285
__iadd__, 324, 326
__iand__, 325
__ifloordiv__, 324
__ilshift__, 325
__imatmul__, 324
__imod__, 324
__imul__, 324
__init__, 251, 253–257, 261, 264, 266, 268, 270, 276, 278, 280, 286, 290, 291, 295, 305, 317, 323, 358, 359
__int__, 323
__invert__, 324
__ior__, 325
__ipow__, 324
__irshift__, 325
__isub__, 324
__iter__, 323, 325
__itruediv__, 324
__ixor__, 325
__le__, 295, 297, 323, 324
__len__, 77, 323, 325
__lshift__, 325

__lt__, 295, 297, 323, 324
__matmul__, 324
__minus__, 324, 325
__mod__, 324
__mul__, 294, 295, 324
__ne__, 281, 282, 287, 295, 297, 323, 324
__new__, 323
__next__, 323, 325
__or__, 325
__pos__, 324, 325
__pow__, 324
__radd__, 295, 324, 325
__rand__, 325
__rdivmod__, 324
__repr__, 278, 280–283, 292, 323
__reversed__, 323, 325
__rfloordiv__, 324
__rlshift__, 325
__rmatmul__, 324
__rmod__, 324
__rmul__, 324
__ror__, 325
__round__, 324
__rpow__, 324
__rrshift__, 325
__rshift__, 325
__rsub__, 324
__rtruediv__, 324
__rxor__, 325
__setitem__, 323, 325
__str__, 278, 280–283, 292, 299, 323
__sub__, 293, 295, 324
__truediv__, 294, 295, 324
__trunc__, 324
__xor__, 325
..., 121, 122, 227, 362, 367
^, 325, 326
^=, 325
\, 77
\\, 77
\", 77
\', 77
\n, 78
\r\n, 78
\t, 78
\u, 80, 81, 89
|, 325, 326
  bit-wise or, 34
  type hints, 106, 183
|=, 325
{, 75
{...}
  dict, 112, 128
  f-string, 73
  set, 112, 124
{{, 75, 78
{}, 129
}, 75
}}, 75, 78
"...", 68, 77, 79, 355
"", 68, 81
"""...""", 79, 119, 355, 356, 367
', 68, 77, 355
'...', 68, 79
'', 79, 355
'''...''', 79
~, 324
0.0, 45, 46
0b, 34, 74
0o, 35
0x, 35, 74
1.7976931348623157e+308, 38, 45
2.718281828459045, 367
3.141592653589793, 363
5e-324, 44, 173

0, 47

abs, 140, 264, 295, 324
acos, 40
add, 125
all, 234, 235, 358
and, 64, 66, 67, 133
any, 234, 235, 358
append, 113, 148, 222–225
ArithmeticError, 189, 190, 193–195, 197, 210–214, 225, 267, 312
as, 165, 195, 209, 317, 318, 323
asin, 40
assert, 171, 172, 194
AssertionError, 172, 194
asterisk, 180
atan, 40, 51
AttributeError, 194
Axes, 180

BaseException, 194, 267
bin, 34
BlockingIOError, 194
bool, 30, 45, 63–65, 67, 77, 81, 323
  conjunction, 64, 66
  disjunction, 64, 66
  negation, 64, 66
bool(0), 77
bool(""), 77
bool(""), 81
bool("False"), 77, 81
bool("True"), 77, 81
break, 146, 148, 151, 158–160, 162, 236, 241, 312
BrokenPipeError, 194
BufferError, 194
bytes, 323

Callable, 183, 185, 317
casefold, 127, 356
ceil, 40, 41, 51–56, 59, 63, 324
ChildProcessError, 194
class, 194, 251, 252, 254–256, 267–269, 358
close, 206–208
collections.abc, 183
complex, 249, 323, 324
comprehension
  dict, 229, 230
  list, 220–222, 256
  set, 228, 229
ConnectionAbortedError, 194
ConnectionError, 194
ConnectionRefusedError, 194
ConnectionResetError, 194
continue, 146, 160
cos, 39, 40, 51
csv, 367

datetime, 279, 280
  datetime, 279
    now, 279
  UTC, 279
def, 161, 163, 240
defusedxml, 382
del, 115, 129, 323
dict, 112, 144, 148, 149, 153, 180, 216, 217, 227, 237, 239, 317
  comprehension, 229, 230
  del, 129
  empty, 129
  items, 129, 149
  iterate over, 149
  keys, 129, 149
  len, 129
  pop, 129
  type hint, 128
  update, 129
  values, 129, 149
dict.items, 129, 218
dict.keys, 129
dict.values, 129, 218

difference, 127, 229
difference_update, 127
divmod, 324
doctest, 227, 362, 367
dunder, 278, 323–325
__abs__, 294, 295, 324, 325
__add__, 292, 293, 295, 324, 325
__and__, 325
__bool__, 77, 323
__bytes__, 323
__call__, 323
__ceil__, 51, 52, 324
__complex__, 323
__contains__, 323, 325
__delitem__, 323, 325
__divmod__, 324
__enter__, 317, 323
__eq__, 281–283, 285–288, 295, 297, 323, 324, 359
__exit__, 317, 318, 323
__float__, 323
__floor__, 324
__floordiv__, 324
__format__, 323
__ge__, 295, 297, 323, 324
__getitem__, 323, 325
__gt__, 295, 297, 323, 324
__hash__, 285–288, 323, 359
__hash]__, 285
__iadd__, 324, 326
__iand__, 325
__ifloordiv__, 324
__ilshift__, 325
__imatmul__, 324
__imod__, 324
__imul__, 324
__init__, 251, 253–257, 261, 264, 266, 268, 270, 276, 278, 280, 286, 290, 291, 295, 305, 317, 323, 358, 359
__int__, 323
__invert__, 324
__ior__, 325
__ipow__, 324
__irshift__, 325
__isub__, 324
__iter__, 323, 325
__itruediv__, 324
__ixor__, 325
__le__, 295, 297, 323, 324
__len__, 77, 323, 325
__lshift__, 325
__lt__, 295, 297, 323, 324
__matmul__, 324
__minus__, 324, 325
__mod__, 324
__mul__, 294, 295, 324
__ne__, 281, 282, 287, 295, 297, 323, 324
__new__, 323
__next__, 323, 325
__or__, 325
__pos__, 324, 325
__pow__, 324
__radd__, 295, 324, 325
__rand__, 325
__rdivmod__, 324
__repr__, 278, 280–283, 292, 323
__reversed__, 323, 325
__rfloordiv__, 324
__rlshift__, 325
__rmatmul__, 324
__rmod__, 324
__rmul__, 324
__ror__, 325
__round__, 324
__rpow__, 324
__rrshift__, 325
__rshift__, 325
__rsub__, 324
__rtruediv__, 324
__rxor__, 325
__setitem__, 323, 325
__str__, 278, 280–283, 292, 299, 323
__sub__, 293, 295, 324
__truediv__, 294, 295, 324
__trunc__, 324
__xor__, 325

e, 38, 46, 84, 92, 367
elif, 137, 138, 356
else, 136–138, 158, 160, 201–204, 236
else:, 135
empty, 113
endswith, 69
enumerate, 153–155
EOFError, 194
escaping, 77–79, 149
except, 193, 195–204, 214, 358
Exception, 46, 98, 158, 172, 177, 186–190, 193–195, 197, 202, 209, 214, 251, 254, 267, 318, 357, 358, 361, 367, 378
Exceptions, 196, 215
exit, 25, 30, 194, 367
exp, 51, 177, 178
extend, 114, 216

f-string, 73, 85
=, 75
interpolation, 73
multi-line, 79
f"...", 73, 368
f"""...""", 79
False, 30, 45–47, 63, 64, 66, 69, 77, 81, 134–137, 235
FileExistsError, 194
FileNotFoundError, 193, 194
filter, 244, 245, 247
Final, 254, 256–258, 266, 268, 286, 290, 299, 317, 358, 359
Final[int], 290
finally, 202–204, 206–208, 215
find, 69, 159, 195, 196
float, 30, 33, 37–47, 59, 63, 64, 73, 75, 77, 81, 86, 88, 100, 172, 173, 192, 203, 259, 323, 355
function, 77
largest, 38, 45
smallest, 44, 173
FloatingPointError, 194
floor, 40, 51, 324
flush, 190
for, 143–151, 158, 160, 163, 190, 191, 217, 222, 225–229, 236, 240, 241, 245, 324–326
else, 312

found, 242
Fraction, 290
fractions, 290
from, 38, 40, 46, 47, 75, 165, 168
fsum, 263, 265, 270
function
  argument, 163
    *, 180
    **, 180
    asterisk, 180
    by name, 179
    default, 177, 179, 357
    keyword, 180
    star, 180
    wildcard, 180
  body, 162
  call, 162, 178
  def, 161
  docstring, 163
  import, 165, 166
  module, 166
  parameter, 161, 162
    *, 180
    **, 180
    asterisk, 180
    by name, 179
    default value, 177, 179, 357
    star, 180
    wildcard, 180
  return, 162
  return value, 162
  signature, 162, 377
  testing, 169
  type hint, 162, 357
  unit test, 169

gcd, 165, 290, 295
Generator, 194, 216, 231–233, 235–237, 240–243
GeneratorExit, 194

hash, 285–287, 359
hex, 35

if, 133–137, 221, 222
if...elif...else, 138, 140
  nested, 139
if...else, 135, 136
  inline, 142
  nested, 136
import, 38, 40, 46, 47, 75, 89, 148, 165, 168, 171, 184, 194, 224
ImportError, 194
in, 69, 115, 122, 127, 143, 285, 323, 324
IndentationError, 194
index, 115, 122, 195, 196
IndexError, 68, 186, 194
inf, 45–48, 50, 63, 77, 84, 92, 93, 172, 173, 175, 189–191, 256
input, 194, 378
insert, 115
inspect, 377
int, 30, 31, 33, 37, 38, 40–42, 44, 45, 47, 63, 64, 73, 76, 81, 85, 100, 203, 245, 323, 355
  function, 76
intersection, 127
IO, 206, 208
  close, 206, 207
  readline, 206, 209
  readonline, 207
  write, 206, 209, 318, 322
is, 82, 109, 111, 116, 256, 281
is not, 109, 111
IsADirectoryError, 194
isclose, 157, 170, 357
isfinite, 47, 50, 175, 190, 254
isinf, 47, 50
isinstance, 191, 212, 217, 231, 254, 255, 269, 295
isnan, 47, 50, 173
isqrt, 148, 151, 228
issubclass, 267, 269
items, 129, 218
iter, 217, 227, 231, 234, 323, 325
Iterable, 216–219, 222, 226, 227, 231, 244–247, 261, 265, 268, 270, 301, 325
iteration, 143
Iterator, 151, 216–219, 227, 231, 234, 236, 240, 241, 245–247, 301, 318, 325
itertools, 244, 247
  takewhile, 244

join, 150, 256, 301, 317

KahanSum, 265
KeyboardInterrupt, 194
KeyError, 129, 194
keys, 129

lambda, 183–185, 245
len, 68, 77, 82, 113, 116, 129, 148, 151, 323, 325
list, 112, 116, 119, 121, 126, 129, 144, 147, 148, 152, 153, 180, 216–218, 220, 223, 224, 226, 230, 231, 234, 237, 239, 240, 356
  *, 116
  +, 116
  append, 113, 148, 222, 224, 225
  comprehension, 220–222, 256
  del, 115
  extend, 114, 216
  in, 115
  index, 115
  insert, 115, 301
  iterate over, 148
  len, 113, 148, 151
  not in, 115
  remove, 116
  reverse, 116
  slicing, 116
  sort, 116, 123
  type hint, 113
  unpacking, 116
list_iterator, 217
lists, 226
Literal, 318, 359
Literal[xyz], 372
LiteralString, 108, 372, 378
log, 39, 40, 46, 51, 82, 220
LookupError, 194
loop, 143
  for, 143, 148, 149
lower, 69
lst, 234
lstrip, 69

maketrans, 317
map, 245–247, 301
math, 38–40, 46, 47, 51–54, 59, 75, 82, 84, 89, 92, 93, 109, 148, 157, 161, 162, 165, 167, 168, 172, 173, 175, 178, 179, 184, 190, 228, 247, 254–256, 263,

265, 290, 357, 363, 367
ceil, 56, 324
floor, 324
trunc, 324
max, 136, 234, 235, 246
mean, 372
MemoryError, 193, 194, 241
min, 136, 224, 225, 234, 235, 246
module, 167, 168
import, 166
ModuleNotFoundError, 194, 332, 338
Mypy, 103, 373

NameError, 94, 95, 194, 197, 198, 201, 202
nan, 46, 47, 49, 50, 64, 77, 81, 84, 92, 93, 157, 172, 173, 175, 189–191, 198, 256, 281, 357
next, 217, 231–235, 241, 242, 325
None, 30, 81, 82, 163, 179, 194
NoneType, 81, 82, 323
not, 64, 66, 67, 134
Not a Number, 46, 47, 49, 64
not in, 115, 116, 122, 127, 229
not is, 116
NotADirectoryError, 194
NotImplemented, 281, 295, 324, 325
NotImplementedError, 194, 250, 268
NotImplementedType, 281, 283
now, 279
numpy, 332, 334–337, 342

object, 269
oct, 35
open, 206, 207, 209, 316, 318, 322
encoding, 206, 207
mode, 206, 207
or, 64, 66, 67
os, 206, 207
remove, 206, 207, 209, 322
OSError, 194
OverflowError, 46, 47, 64, 186, 193–195, 203, 213, 214

package, 167
PermissionError, 194
pi, 38, 75, 84, 88, 89, 92, 93, 177–179, 184, 256, 363
pip, 103, 152, 170, 375
plot, 180
pop, 129
print, 22, 30, 77, 79, 81, 82, 85, 124, 134, 140, 145, 146, 150, 161, 162, 165, 190, 203, 232, 292, 378
flush, 190, 232
ProcessLookupError, 194
pytest, 209, 213, 214
raises, 209, 210
PythonFinalizationError, 194

raise, 174, 188, 190, 193, 197, 198, 203, 209, 213, 214, 217, 241, 250, 251, 254, 268, 270, 290, 358
raises, 209–211, 213, 214
range, 144, 146–148, 150, 151, 160, 163, 216, 218–220, 222, 229, 230, 236
range_iterator, 218
re, 376
readline, 206, 207, 209
RecursionError, 194
ReferenceError, 194
remove, 116, 126, 206–209, 322
repeat, 224, 225
replace, 69
repr, 278–280, 292, 323
return, 158, 162, 163, 165, 240, 243
reverse, 116
reversed, 325
rfind, 69
round, 40, 41, 51, 324, 365
rstrip, 69
ruff, 376
RuntimeError, 194

Self, 317, 359
self, 251, 253, 255, 317, 324, 325, 359
Sequence, 68
set, 112, 124, 125, 144, 148, 153, 216, 217, 226, 229, 237, 239, 246
add, 125, 228
comprehension, 228, 229
difference, 127, 229
difference_update, 127
empty, 126
in, 127
intersection, 127
iterate over, 148
not in, 127
remove, 126
symmetric_difference, 127
type hint, 124
union, 127
update, 125
signature, 377
sin, 39, 40, 51, 75, 184
slice, 68, 116, 144
slicing, 68, 116
sort, 116
sorted, 126, 127, 237, 356
split, 236, 245
sqrt, 89, 109, 110, 157, 161, 162, 170, 177, 178, 247, 255
star, 180
startswith, 69
statistics, 372
mean, 372
StopAsyncIteration, 194
StopIteration, 194, 217, 219, 234, 241
str, 30, 68, 69, 73, 79, 81, 150, 222, 234, 278, 279, 301, 317, 323
+, 68
[...], 68
[], 68
case-sensitive, 69, 366
casefold, 127, 356
concatenation, 68
doc, 119, 356
function, 163
empty, 68
escaping, 77–79, 149
f, 73, 79, 85
find, 159, 195
function, 73
index, 195, 196
interpolation, 73
join, 150, 256, 301, 317
len, 68
length, 68
literal, 68
lowercase, 69
maketrans, 317
slicing, 68

split, 236, 245
translate, 317
unicode, 80
uppercase, 69
strip, 69
sum, 231, 234, 235, 246, 247, 263–265
swap, 99, 100, 356
symmetric_difference, 127
SyntaxError, 194
SystemError, 194
SystemExit, 194

TabError, 194
takewhile, 244, 245
tan, 39, 40, 51
timeit, 223, 224, 361
repeat, 224
TimeoutError, 194
translate, 317
True, 30, 45–47, 63, 64, 66, 69, 73, 77, 81, 133, 135–137, 157, 235
trunc, 40, 41, 51, 324
try, 195–198, 200–204, 206–208, 215
tuple, 112, 120–122, 126, 144, 148, 152, 153, 165, 180, 216, 217, 222, 226, 231, 237, 245, 356
in, 122
index, 122
iterate over, 148
not in, 122
type hint, 121
..., 122
unpacking, 123, 149, 165
type, 64, 82, 100, 231, 255, 256
TypeError, 124, 186, 188, 191–194, 203, 210–212, 254, 295, 325
typing, 183, 206, 217, 231, 254
Final, 254, 358
Literal, 318, 359
Self, 317, 359
typing.Iterable, 217, 270
typing.Iterator, 217

UnboundLocalError, 194
UnicodeDecodeError, 194
UnicodeEncodeError, 194
UnicodeError, 194
UnicodeTranslateError, 194
union, 127
unpacking, 116, 123, 149, 165
update, 125, 129
upper, 69
UTF-8, 206, 207

ValueError, 188, 189, 194, 196, 247, 254, 268, 270, 276
values, 129, 218
venv, 381

while, 156–160, 165, 241
wildcard, 180
with, 207–209, 211–213, 215, 316–318, 323, 326
write, 206, 209, 318, 322

yield, 240–244, 247

ZeroDivisionError, 173, 194, 196–198, 202, 203, 290
zip, 246, 247

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. "TensorFlow: A System for Large-Scale Machine Learning". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'2016)*. Nov. 2–4, 2016, Savannah, GA, USA. Ed. by Kimberly Keeton and Timothy Roscoe. Berkeley, CA, USA: USENIX Association, 2016, pp. 265–283. ISBN: 978-1-931971-33-1. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi (visited on 2024-06-26) (cit. on pp. 3, 378).

[2] Milton Abramowitz and Irene A. Stegun, eds. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. Tenth Printing, with corrections. Vol. 55 of National Bureau of Standards Applied Mathematics Series. Gaithersburg, MD, USA: United States Department of Commerce, National Bureau of Standards and Washington, D.C., USA: United States Government Printing Office, June 1964–Dec. 1972. ISSN: 0083-1786. ISBN: 978-0-16-000202-1. URL: https://personal.math.ubc.ca/~cbm/aands (visited on 2025-09-05) (cit. on pp. 177, 184, 375).

[3] Joseph Adams, Egon Balas, and Daniel Zawack. "The Shifting Bottleneck Procedure for Job Shop Scheduling". *Management Science* 34(3):391–401, Mar. 1988. Catonsville, MD, USA: The Institute for Operations Research and the Management Sciences (INFORMS). ISSN: 0025-1909. doi:10.1287/mnsc.34.3.391 (cit. on p. 370).

[4] Olatunde Adedeji. *Full-Stack Flask and React*. Birmingham, England, UK: Packt Publishing Ltd, Nov. 2023. ISBN: 978-1-80324-844-8 (cit. on pp. 4, 368).

[5] David J. Agans. *Debugging*. New York, NY, USA: AMACOM, Sept. 2002. ISBN: 978-0-8144-2678-4 (cit. on pp. 303, 362, 367).

[6] Dowon "akahard2dj", Ilhan "ilayn" Polat, Pauli "pv" Virtanen, Daniel "dpinol" Pinyol, Lucas "lucascolley" Colley, Matt "mdhaber" Haberland, and Lars "lagru" Grüter. *Issue #9038: Applying Type Hint(PEP 484) for SciPy*. San Francisco, CA, USA: GitHub Inc, July 16, 2018–June 16, 2024. URL: https://github.com/scipy/scipy/issues/9038 (visited on 2025-02-02) (cit. on p. 108).

[7] *American Standard Code for Information Interchange (ASCII 1963)*. Tech. rep. ASA X3.4-1963. New York, NY, USA: American Standards AssociationIncorporated, June 17, 1963. URL: https://www.sensitiveresearch.com/Archive/CharCodeHist/Files/CODES%20standards%20documents%20ASCII%20Sean%20Leonard%20Oct%202015/ASCII%2063,%20X3.4-1963.pdf (visited on 2024-07-26) (cit. on p. 79).

[8] David Amos. "How to Round Numbers in Python". In: *Real Python Tutorials*. Vancouver, BC, Canada: DevCademy Media Inc., Dec. 7, 2024. URL: https://realpython.com/python-rounding (visited on 2025-04-27) (cit. on pp. 53, 54).

[9] AndrewBadr. "TimeComplexity". In: *The Python Wiki*. Beaverton, OR, USA: Python Software Foundation (PSF), Jan. 19, 2023. URL: https://wiki.python.org/moin/TimeComplexity (visited on 2024-08-27) (cit. on pp. 124, 128, 285).

[10] Michael Andrews. *XML, Latin, and the Demise or Endurance of Languages*. Story Needle, Nov. 23, 2020. URL: https://storyneedle.com/xml-latin-and-the-demise-or-endurance-of-languages (visited on 2024-12-15) (cit. on p. 316).

[11] Miguel F. Anjos. *The QAPLIB Challenge*. Edinburgh, Scotland, UK: Miguel F. Anjos @ University of Edinburgh, Nov. 19, 2023. URL: https://www.miguelanjos.com/projects/2999 (visited on 2026-09-28). The document lists the 28 instances of QAPLIB [89, 91] that had not yet been solved to optimality by the time of its writing. (Cit. on p. 376).

[12] "Annotating `Callable` Objects". In: *Python 3 Documentation*. *The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/library/typing.html#annotating-callables (visited on 2024-10-09) (cit. on p. 183).

[13] Denis Antipov and Benjamin Doerr. "Precise Runtime Analysis for Plateaus". In: *15th International Conference on Parallel Problem Solving from Nature (PPSN XV)*. Vol. 2. Sept. 8–12, 2018, Coimbra, Portugal. Ed. by Anne Auger, Carlos M. Fonseca, Nuno Lourenço, Penousal Machado, Luís Paquete, and L. Darrell Whitley. Vol. 11102 of Lecture Notes in Computer Science (LNCS). Cham, Switzerland: Springer, 2018, pp. 117–128. ISSN: 0302-9743. ISBN: 978-3-319-99258-7. doi:10.1007/978-3-319-99259-4_10 (cit. on p. 375).

[14] Tom Mike Apostol. "Primitive Functions and the Second Fundamental Theorem of Calculus". In: *One-Variable Calculus, with an Introduction to Linear Algebra*. 2nd ed. Vol. 1 of Calculus. Chichester, West Sussex, England, UK: John Wiley and Sons Ltd., Jan. 1991. Chap. 5.3, pp. 205–207. ISBN: 978-0-471-00005-1 (cit. on p. 181).

[15] David Lee Applegate, Robert E. Bixby, Vašek Chvátal, and William John Cook. *The Traveling Salesman Problem: A Computational Study*. 2nd ed. Vol. 17 of Princeton Series in Applied Mathematics. Princeton, NJ, USA: Princeton University Press, 2007. ISBN: 978-0-691-12993-8. This book also discusses the Concorde TSP solver. (Cit. on pp. 366, 379).

[16] David Lee Applegate and William John Cook. "A Computational Study of the Job-Shop Scheduling Problem". *ORSA Journal on Computing* 3(2):149–156, May 1991. Catonsville, MD, USA: The Institute for Operations Research and the Management Sciences (INFORMS). ISSN: 0899-1499. doi:10.1287/ijoc.3.2.149. The JSSP instances used were generated in Bonn in 1986. (Cit. on p. 370).

[17] David Ascher, ed. *Python Cookbook*. 1st ed. Sebastopol, CA, USA: O'Reilly Media, Inc., July 2002. ISBN: 978-0-596-00167-4 (cit. on p. 5).

[18] Adam Aspin and Karine Aspin. *Query Answers with MariaDB – Volume I: Introduction to SQL Queries*. Tetras Publishing, Oct. 2018. ISBN: 978-1-9996172-4-0. See also [19] (cit. on pp. 372, 390).

[19] Adam Aspin and Karine Aspin. *Query Answers with MariaDB – Volume II: In-Depth Querying*. Tetras Publishing, Oct. 2018. ISBN: 978-1-9996172-5-7. See also [18] (cit. on pp. 372, 390).

[20] Kendall Eugene Atkinson. *An Introduction to Numerical Analysis*. 2nd ed. Chichester, West Sussex, England, UK: John Wiley and Sons Ltd., 1991. ISBN: 978-0-471-62489-9 (cit. on p. 181).

[21] Anne Auger, Carlos M. Fonseca, Nuno Lourenço, Penousal Machado, Luís Paquete, and L. Darrell Whitley, eds. *15th International Conference on Parallel Problem Solving from Nature (PPSN XV)*. Vol. 2. Sept. 8–12, 2018, Coimbra, Portugal. Vol. 11102 of Lecture Notes in Computer Science (LNCS). Cham, Switzerland: Springer, 2018. ISSN: 0302-9743. ISBN: 978-3-319-99258-7. doi:10.1007/978-3-319-99259-4.

[22] Ivo Babuška. "Numerical Stability in Mathematical Analysis". In: *World Congress on Information Processing (IFIP'1968)*. Vol. 1: Mathematics, Software. Aug. 5, 1966–Aug. 10, 1968, Edinburgh, Scotland, UK. Ed. by A.J.H. Morrell. Laxenburg, Austria: International Federation for Information Processing (IFIP). Amsterdam, The Netherlands: North-Holland Publishing Co., 1969, pp. 11–23. ISBN: 978-0-7204-2032-6 (cit. on pp. 259–262, 270, 412).

[23] Paul Gustav Heinrich Bachmann. *Die Analytische Zahlentheorie / Dargestellt von Paul Bachmann*. Vol. Zweiter Theil of Zahlentheorie: Versuch einer Gesamtdarstellung dieser Wissenschaft in ihren Haupttheilen. Leipzig, Sachsen, Germany: B. G. Teubner, 1894. ISBN: 978-1-4181-6963-3. URL: http://gallica.bnf.fr/ark:/12148/bpt6k994750 (visited on 2023-12-13) (cit. on p. 374).

[24] Thomas Bäck, David B. Fogel, and Zbigniew "Zbyszek" Michalewicz, eds. *Handbook of Evolutionary Computation*. Bristol, England, UK: IOP Publishing Ltd and Oxford, Oxfordshire, England, UK: Oxford University Press, 1997. ISBN: 978-0-7503-0392-7 (cit. on pp. 367, 368, 377).

[25] "Banker's Rounding". In: *Alipay$^+$ Documentation*. Singapore: Alipay Connect Pte. Lte., 2022. URL: https://docs.alipayplus.com/alipayplus/alipayplus/reconcile_mpp/bank_rounding (visited on 2025-07-18) (cit. on pp. 40, 365).

[26] Christopher Barker. *A Function for Testing Approximate Equality*. Python Enhancement Proposal (PEP) 485. Beaverton, OR, USA: Python Software Foundation (PSF), Jan. 20, 2015. URL: https://peps.python.org/pep-0485 (visited on 2024-09-02) (cit. on pp. 157, 357).

[27] Daniel J. Barrett. *Efficient Linux at the Command Line*. Sebastopol, CA, USA: O'Reilly Media, Inc., Feb. 2022. ISBN: 978-1-0981-1340-7 (cit. on pp. 9, 371, 378).

[28] Daniel J. Barrett, Richard E. Silverman, and Robert G. Byrnes. *SSH, The Secure Shell: The Definitive Guide*. 2nd ed. Sebastopol, CA, USA: O'Reilly Media, Inc., May 2005. ISBN: 978-0-596-00895-6 (cit. on p. 378).

[29] Paul Barry. *Head First Python*. 3rd ed. Sebastopol, CA, USA: O'Reilly Media, Inc., Aug. 2023. ISBN: 978-1-4920-5129-9 (cit. on p. 4).

[30] Daniel Bartholomew. *Learning the MariaDB Ecosystem: Enterprise-level Features for Scalability and Availability*. New York, NY, USA: Apress Media, LLC, Oct. 2019. ISBN: 978-1-4842-5514-8 (cit. on p. 372).

[31] "Basic Customizations: `object.__hash__(self)`". In: *Python 3 Documentation*. *The Python Language Reference*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. Chap. 3.3.1. URL: https://docs.python.org/3/reference/datamodel.html#object.__hash__ (visited on 2024-12-09) (cit. on pp. 285, 286, 359).

[32] "Basic Customizations: `object.__repr__`". In: *Python 3 Documentation*. *The Python Language Reference*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. Chap. 3.3.1. URL: https://docs.python.org/3/reference/datamodel.html#object.__repr__ (visited on 2024-09-25) (cit. on p. 278).

[33] "Basic Customizations: `object.__str__`". In: *Python 3 Documentation*. *The Python Language Reference*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. Chap. 3.3.1. URL: https://docs.python.org/3/reference/datamodel.html#object.__str__ (visited on 2024-09-25) (cit. on p. 278).

[34] Roberto Battiti and Giampietro Tecchiolli. "Local Search with Memory: Benchmarking RTS". *OR Spectrum: Quantitative Approaches in Management. Formerly* Operations-Research-Spektrum 17:67–86, June 1995. London, England, UK: Springer Nature Limited. ISSN: 0171-6468. doi:10.1007/BF01719249 (cit. on p. 376).

[35] Roberto Battiti and Giampietro Tecchiolli. "Simulated Annealing and Tabu Search in the Long Run: A Comparison on QAP Tasks". *Computers & Mathematics with Applications* 28(6):1–8, Sept. 1994. Amsterdam, The Netherlands: Elsevier B.V. ISSN: 0898-1221. doi:10.1016/0898-1221(94)00147-2 (cit. on pp. 376, 379).

[36] Roberto Battiti and Giampietro Tecchiolli. "The Reactive Tabu Search". 6(2):126–140, May 1994. doi:10.1287/ijoc.6.2.126. Applications of Reactive Tabu Search (RTS) to the 0-1 Knapsack and QAP. (Cit. on pp. 376, 379).

[37] Roberto Battiti and Giampietro Tecchiolli. "Training Neural Nets with the Reactive Tabu Search". *IEEE Transactions on Neural Networks* 6(5):1185–1200, Sept. 1995. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE). ISSN: 1045-9227. doi:10.1109/72.410361. See also "Training Neural Nets with the Reactive Tabu Search, Technical Report UTM 421, Trento, Alto Adige/Südtirol, Italy: University of Trento, 11 1993" (cit. on p. 376).

[38] Gary Baumgartner, Danny Heap, and Richard Krueger. "Numerical Systems". In: *Course Notes for CSC165H: Mathematical Expression and Reasoning for Computer Science*. Toronto, ON, Canada: Department of Computer Science, University of Toronto, Aut. 2006. Chap. 7. URL: https://www.cs.toronto.edu/~krueger/csc165h/f06/lectures/ch7.pdf (visited on 2024-07-27) (cit. on pp. 40, 355).

[39] James C. Bean. "Genetic Algorithms and Random Keys for Sequencing and Optimization". *ORSA Journal on Computing* 6(2):107–220, Spr. 1994. Catonsville, MD, USA: The Institute for Operations Research and the Management Sciences (INFORMS). ISSN: 0899-1499. doi:10.1287/ijoc.6.2.154. Applies the Random Keys encoding to scheduling and the QAP. See [40]. (Cit. on pp. 366, 376, 392).

[40] James C. Bean. *Genetics and Random Keys for Sequencing and Optimization*. Tech. rep. 92-43. Ann Arbor, MI, USA: University of Michigan, Department of Industrial & Operations Engineering, June 30, 1992. URL: https://www.math.wm.edu/~rrkinc/homework/RdmKeyGA.pdf (visited on 2026-10-01). See [39]. (Cit. on pp. 366, 376, 391).

[41] John Edward Beasley. "OR-Library: Distributing Test Problems by Electronic Mail". *The Journal of the Operational Research Society (JORS)* 41:1069–1072, Nov. 1990. London, England, UK: Taylor and Francis Ltd. and London, England, UK: Springer Nature Limited. ISSN: 0160-5682. doi:10.1057/jors.1990.166. See the website [42] (cit. on pp. 370, 374, 392).

[42] John Edward Beasley. *Welcome to OR-Library*. Uxbridge, Middlesex, England, UK: Brunel University of London, June 1990–Feb. 2018. URL: https://people.brunel.ac.uk/~mastjjb/jeb/info.html (visited on 2026-09-30). See the article [41]. Contains instances of the JSSP, 1D- and 2D-Packing, Knapsack, periodic TSPs, amongst others. (Cit. on pp. 370, 374, 392).

[43] David M. Beazley. "Data Processing with Pandas". *;login: Usenix Magazin* 37(6), Dec. 2012. Berkeley, CA, USA: USENIX Association. ISSN: 1044-6397. URL: https://www.usenix.org/publications/login/december-2012-volume-37-number-6/data-processing-pandas (visited on 2024-06-25) (cit. on pp. 3, 375).

[44] Kent L. Beck. *JUnit Pocket Guide*. Sebastopol, CA, USA: O'Reilly Media, Inc., Sept. 2004. ISBN: 978-0-596-00743-0 (cit. on pp. 169, 380).

[45] Kent L. Beck and Erich Gamma. "Test-Infected: Programmers Love Writing Tests". In: *More Java Gems*. Ed. by Dwight Deugo. New York, NY, USA: Cambridge University Press (CUP), Jan. 2000, pp. 357–376. ISBN: 978-0-521-77477-2. doi:10.1017/CBO9780511550881.029. URL: http://members.pingnet.ch/gamma/junit.htm (visited on 2025-09-05) (cit. on p. 169).

[46] Andreas Beham, Michael Affenzeller, and Erik Pitzer. "Metaheuristic Algorithms for the Quadratic Assignment Problem: Performance and Comparison". In: *Innovative Technologies in Management and Science*. Ed. by Ryszard Klempous and Jan Nikodem. Vol. 10 of Topics in Intelligent Engineering and Informatics (TIEI). Cham, Switzerland: Springer, 2015, pp. 171–190. ISSN: 2193-9411. ISBN: 978-3-319-12651-7. doi:10.1007/978-3-319-12652-4_10. Algorithms like SLS/iterated local search (ILS), SA, TS, VNS, EAs, MAs are compared and, with the exception of SA and EA, all perform best on some instance groups of QAPLIB. (Cit. on p. 376).

[47] Richard Beigel. "Irrationality without Number Theory". *The American Mathematical Monthly* 98(4):332–335, Apr. 1991. London, England, UK: Taylor and Francis Ltd. ISSN: 1930-0972. doi:10.1080/00029890.1991.12000762. URL: https://cis.temple.edu/~beigel/papers/irrationality-amm.PS.gz (visited on 2026-11-04) (cit. on pp. 38, 370).

[48] Ben Beitler. *Hands-On Microsoft Access 2019*. Birmingham, England, UK: Packt Publishing Ltd, Mar. 2020. ISBN: 978-1-83898-747-3 (cit. on pp. 371, 373).

[49] Richard Ernest Bellman. *Dynamic Programming*. Princeton Landmarks in Mathematics and Physics. Princeton, NJ, USA: Princeton University Press, 1957–2003. ISBN: 978-0-691-14668-3 (cit. on p. 373).

[50] Bengt-Erik Bengtsson. "Packing Rectangular Pieces— A Heuristic Approach". *The Computer Journal* 25(3):353–357, Aug. 1982. Oxford, Oxfordshire, England, UK: Oxford University Press. ISSN: 0010-4620. doi:10.1093/comjnl/25.3.353 (cit. on p. 364).

[51] Jon Bentley. *Programming Pearls*. 2nd ed. Reading, MA, USA: Addison-Wesley Professional, Oct. 7, 1999. ISBN: 978-0-201-65788-3 (cit. on pp. 158, 159).

[52] Tim Berners-Lee. *Re: Qualifiers on Hypertext links...* Geneva, Switzerland: World Wide Web project, European Organization for Nuclear Research (CERN) and Newsgroups: alt.hypertext, Aug. 6, 1991. URL: https://www.w3.org/People/Berners-Lee/1991/08/art-6484.txt (visited on 2025-02-05) (cit. on pp. 369, 381).

[53] Tim Berners-Lee, Roy T. Fielding, and Henrik Frystyk Nielsen. *Hypertext Transfer Protocol -- HTTP/1.0*. Request for Comments (RFC) 1945. Wilmington, DE, USA: Internet Engineering Task Force (IETF), May 1996. URL: https://www.ietf.org/rfc/rfc1945.txt (visited on 2025-02-05) (cit. on p. 369).

[54] Tim Berners-Lee, Larry Masinter, and Mark P. McCahill. *Uniform Resource Locators (URL)*. Request for Comments (RFC) 1738. Wilmington, DE, USA: Internet Engineering Task Force (IETF), Dec. 1994. URL: https://www.ietf.org/rfc/rfc1738.txt (visited on 2025-02-05) (cit. on p. 380).

[55] Alex Berson. *Client/Server Architecture*. 2nd ed. Computer Communications Series. New York, NY, USA: McGraw-Hill, Mar. 29, 1996. ISBN: 978-0-07-005664-0 (cit. on p. 366).

[56] Fabian Beuke. *GitHut 2.0: GitHub Language Statistics*. San Francisco, CA, USA: GitHub Inc, 2023. URL: https://madnight.github.io/githut (visited on 2024-06-24) (cit. on p. 3).

[57] Muhammed Beyaz, Tansel Dökeroğlu, and Ahmet Cosar. "Robust Hyper-Heuristic Algorithms for the Offline Oriented/Non-Oriented 2D Bin Packing Problems". *Applied Soft Computing (ASOC)* 36:236–245, Nov. 2015. Amsterdam, The Netherlands: Elsevier B.V. ISSN: 1568-4946. doi:10.1016/j.asoc.2015.06.063 (cit. on p. 364).

[58] Christian Bierwirth. "A Generalized Permutation Approach to Job Shop Scheduling with Genetic Algorithms". *OR Spectrum: Quantitative Approaches in Management. Formerly* Operations-Research-Spektrum 17(2–3):87–92, June 1995. London, England, UK: Springer Nature Limited. ISSN: 0171-6468. doi:10.1007/BF01719250. URL: https://www.researchgate.net/publication/240263036 (visited on 2026-09-30). Proposes a permutations-with-repetitions based encoding for the JSSP, see also [59]. (Cit. on pp. 368, 370, 393, 406).

[59] Christian Bierwirth, Dirk C. Mattfeld, and Herbert Kopfer. "On Permutation Representations for Scheduling Problems". In: *4th International Conference on Parallel Problem Solving from Nature (PPSN IV)*. Sept. 22–24, 1996, Berlin, Germany. Ed. by Hans-Michael Voigt, Werner Ebeling, Ingo Rechenberg, and Hans-Paul Schwefel. Vol. 1141 of Lecture Notes in Computer Science (LNCS). Berlin/Heidelberg, Germany: Springer-Verlag GmbH Germany, 1996, pp. 310–318. ISSN: 0302-9743. ISBN: 978-3-540-61723-5. doi:10.1007/3-540-61723-X_995. URL: https://www.researchgate.net/publication/2753293 (visited on 2026-09-30). Proposes a permutations-with-repetitions based encoding for the JSSP, see also [58]. (Cit. on pp. 368, 370, 393, 406).

[60] Jacek Błażewicz, Wolfgang Domschke, and Erwin Pesch. "The Job Shop Scheduling Problem: Conventional and New Solution Techniques". *European Journal of Operational Research* 93(1):1–33, Aug. 1996. Amsterdam, The Netherlands: Elsevier B.V. ISSN: 0377-2217. doi:10.1016/0377-2217(95)00362-2 (cit. on p. 370).

[61] Joshua Bloch. *Effective Java*. Reading, MA, USA: Addison-Wesley Professional, May 2008. ISBN: 978-0-321-35668-0 (cit. on pp. 257, 370).

[62] Christian Blum and Verena Schmid. "Solving the 2D Bin Packing Problem by Means of a Hybrid Evolutionary Algorithm". In: *International Conference on Computational Science (ICCS'2013)*. June 5–7, 2013, Barcelona, Spain. Ed. by Vassil Alexandrov, Michael Lees, Valeria V. Krzhizhanovskaya, Jack J. Dongarra, and Peter M. A. Sloot. Vol. 18 of Procedia Computer Science. Amsterdam, The Netherlands: Elsevier B.V., 2013, pp. 899–908. ISSN: 1877-0509. doi:10.1016/j.procs.2013.05.255. An EA for the 2BP|O|F using the heuristic by Wong and Lee [640]. (Cit. on p. 364).

[63] Bernard Obeng Boateng. *Data Modeling with Microsoft Excel*. Birmingham, England, UK: Packt Publishing Ltd, Nov. 2023. ISBN: 978-1-80324-028-2 (cit. on p. 373).

[64] Paul Boddie, Skip Montanaro, Michael Foord, and Alex Martelli. "Why is Python a dynamic language and also a strongly typed language". In: *The Python Wiki*. Beaverton, OR, USA: Python Software Foundation (PSF), Aug. 12, 2008–Feb. 24, 2012. URL: https://wiki.python.org/moin/Why%20is%20Python%20a%20dynamic%20language%20and%20also%20a%20strongly%20typed%20language (visited on 2025-08-08) (cit. on p. 101).

[65] Corrado Böhm. "On a Family of Turing Machines and the Related Programming Language". *ICC Bulletin* 3(3):185–194, July 1964. Rome, Italy: International Computation Centre (ICC). URL: https://www.cs.tufts.edu/comp/150FP/archive/corrado-bohm/turing-machine-pl.pdf (visited on 2025-09-01) (cit. on p. 143).

[66] Corrado Böhm and Guiseppe Jacopini. "Flow Diagrams, Turing Machines and Languages with only Two Formation Rules". *Communications of the ACM (CACM)* 9(5):366–371, May 1966. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/355592.365646. URL: https://cs.unibo.it/~martini/PP/bohm-jac.pdf (visited on 2025-09-01) (cit. on p. 143).

[67] Ethan Bommarito and Michael Bommarito. *An Empirical Analysis of the Python Package Index (PyPI)*. arXiv.org: Computing Research Repository (CoRR) abs/1907.11073. Ithaca, NY, USA: Cornell Universiy Library, July 26, 2019. doi:10.48550/arXiv.1907.11073. URL: https://arxiv.org/abs/1907.11073 (visited on 2024-08-17). arXiv:1907.11073v2 [cs.SE] 26 Jul 2019 (cit. on p. 375).

[68] Borko Bošković and Janez Brest. "Computational Search of Long Skew-Symmetric Binary Sequences with High Merit Factors". *MENDEL – Soft Computing Journal* 28(2):17–24, Dec. 2022. Brno, Czech Republic: Brno University of Technology (Vysoké učení technické v Brně). ISSN: 1803-3814. doi:10.13164/mendel.2022.2.017 (cit. on p. 371).

[69] Borko Bošković, Franc Brglez, and Janez Brest. "Low-Autocorrelation Binary Sequences: On Improved Merit Factors and Runtime Predictions to Achieve Them". *Applied Soft Computing (ASOC)* 56:262–285, July 2017. Amsterdam, The Netherlands: Elsevier B.V. ISSN: 1568-4946. doi:10.1016/J.ASOC.2017.02.024 (cit. on p. 371).

[70] Silvia Botros and Jeremy Tinley. *High Performance MySQL*. 4th ed. Sebastopol, CA, USA: O'Reilly Media, Inc., Nov. 2021. ISBN: 978-1-4920-8051-0 (cit. on p. 373).

[71] Ed Bott. *Windows 11 Inside Out*. Hoboken, NJ, USA: Microsoft Press, Pearson Education, Inc., Feb. 2023. ISBN: 978-0-13-769132-6 (cit. on pp. 9, 373).

[72] Georg Brandl and Serhiy Storchaka. *Underscores in Numeric Literals*. Python Enhancement Proposal (PEP) 515. Beaverton, OR, USA: Python Software Foundation (PSF), Feb. 10, 2016. URL: https://peps.python.org/pep-0515 (visited on 2024-09-23) (cit. on pp. 44, 355).

[73] Ron Brash and Ganesh Naik. *Bash Cookbook*. Birmingham, England, UK: Packt Publishing Ltd, July 2018. ISBN: 978-1-78862-936-2 (cit. on pp. 365, 439).

[74] Tim Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. Request for Comments (RFC) 8259. Wilmington, DE, USA: Internet Engineering Task Force (IETF), Dec. 2017. URL: https://www.ietf.org/rfc/rfc8259.txt (visited on 2025-02-05) (cit. on pp. 316, 370).

[75] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler, eds. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. W3C Recommendation. Wakefield, MA, USA: World Wide Web Consortium (W3C), Nov. 26, 2008–Feb. 7, 2013. URL: http://www.w3.org/TR/2008/REC-xml-20081126 (visited on 2024-12-15) (cit. on pp. 316–318, 382).

[76] Richard P. Brent. *Further Analysis of the Binary Euclidean Algorithm*. arXiv.org: Computing Research Repository (CoRR) abs/1303.2772. Ithaca, NY, USA: Cornell Universiy Library, Nov. 1999–Mar. 12, 2013. doi:10.48550/arXiv.1303.2772. URL: https://arxiv.org/abs/1303.2772 (visited on 2024-09-28). arXiv:1303.2772v1 [cs.DS] 12 Mar 2013. Report number PRG TR-7-99 of Oxford, Oxfordshire, England, UK: Oxford University Computing Laboratory, 11 1999, see https://maths-people.anu.edu.au/~brent/pd/rpb183tr.pdf (cit. on pp. 164, 165).

[77] Janez Brest and Borko Bošković. "A Heuristic Algorithm for a Low Autocorrelation Binary Sequence Problem with Odd Length and High Merit Factor". *IEEE Access* 6:4127–4134, Jan. 2018. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE). ISSN: 2169-3536. doi:10.1109/ACCESS.2018.2789916 (cit. on p. 371).

[78] Oleg Broytman and Jim J. Jewett. *`str(container)` should call `str(item)`, not `repr(item)` [Rejected]*. Python Enhancement Proposal (PEP) 3140. Beaverton, OR, USA: Python Software Foundation (PSF), May 27–28, 2008. URL: https://peps.python.org/pep-3140 (visited on 2024-12-08) (cit. on p. 279).

[79] Florian Bruhin. *Python f-Strings*. Winterthur, Switzerland: Bruhin Software, May 31, 2023. URL: https://fstring.help (visited on 2024-07-25) (cit. on pp. 6, 73, 368).

[80] Brandt Bucher. *Add Optional Length-Checking To `zip`*. Python Enhancement Proposal (PEP) 618. Beaverton, OR, USA: Python Software Foundation (PSF), May 1, 2020. URL: https://peps.python.org/pep-0618 (visited on 2024-11-09) (cit. on p. 247).

[81] "Built-in Constants". In: *Python 3 Documentation. The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/library/constants.html (visited on 2024-12-09) (cit. on p. 281).

[82] "Built-in `Exceptions`". In: *Python 3 Documentation*. *The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/library/exceptions.html (visited on 2024-10-29) (cit. on pp. 193, 194, 285).

[83] "Built-in Functions". In: *Python 3 Documentation*. *The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/library/functions.html (visited on 2024-12-09) (cit. on pp. 225, 231, 245, 246, 287).

[84] "Built-in Functions: class bool". In: *Python 3 Documentation*. *The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/library/functions.html#bool (visited on 2025-10-10) (cit. on p. 77).

[85] "Built-in Functions: class object". In: *Python 3 Documentation*. *The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/library/functions.html#object (visited on 2025-09-25) (cit. on p. 269).

[86] "Built-in Functions: `round(number, ndigits=None)`". In: *Python 3 Documentation*. *The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/builtins/functions.html#round (visited on 2026-10-25) (cit. on pp. 40, 365).

[87] "Built-in Types". In: *Python 3 Documentation*. *The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/library/stdtypes.html (visited on 2024-08-22) (cit. on p. 112).

[88] "Built-in Types: `Iterator` Types". In: *Python 3 Documentation*. *The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/library/stdtypes.html#iterator-types (visited on 2025-09-16) (cit. on pp. 216, 217).

[89] Rainer E. Burkard, Eranda Çela, Stefan E. Karisch, Franz Rendl, Miguel F. Anjos, and Peter M. Hahn. *QAPLIB – A Quadratic Assignment Problem Library – Problem Instances and Solutions*. Edinburgh, Scotland, UK: University of Edinburgh, Apr. 20, 2012–Mar. 31, 2022. doi:10.7488/ds/3428. This is the 2012 status of the QAPLIB dataset from [91] (cit. on pp. 376, 390, 395).

[90] Rainer E. Burkard, Eranda Çela, Panos Miltiades Pardalos, and Leonidas S. Pitsoulis. "The Quadratic Assignment Problem". In: *Handbook of Combinatorial Optimization*. Ed. by Panos Miltiades Pardalos, Ding-Zhu Du, and Ronald Lewis Graham. 1st ed. Boston, MA, USA: Springer, 1998, pp. 1713–1809. ISBN: 978-1-4613-7987-4. doi:10.1007/978-1-4613-0303-9_27 (cit. on p. 376).

[91] Rainer E. Burkard, Stefan E. Karisch, and Franz Rendl. "QAPLIB – A Quadratic Assignment Problem Library". *Journal of Global Optimization* 10(4):391–403, June 1997. London, England, UK: Springer Nature Limited. ISSN: 0925-5001. doi:10.1023/A:1008293323270. This article introduces QAPLIB, a set of benchmark problems for the QAP, now hosted at [89]. (Cit. on pp. 376, 390, 395).

[92] Rainer E. Burkard and Josef Offermann. "Entwurf von Schreibmaschinentastaturen mittels quadratischer Zuordnungsprobleme". *Zeitschrift für Operations Research (ZOR). now* Mathematical Methods of Operations Research (MMOR) 21:B121–B132, Aug. 1977. Heidelberg, Baden-Württemberg, Germany: Physica-Verlag; now London, England, UK: Springer Nature Limited. ISSN: 0340-9422. doi:10.1007/BF01918175. Keyboard layout modeled as QAPs. (Cit. on p. 376).

[93] Brett Cannon, Jiwon Seo, Yury Selivanov, and Larry Hastings. *Function Signature Object*. Python Enhancement Proposal (PEP) 362. Beaverton, OR, USA: Python Software Foundation (PSF), Aug. 21, 2006–June 4, 2012. URL: https://peps.python.org/pep-0362 (visited on 2024-12-12) (cit. on p. 377).

[94] Jason Cannon. *High Availability for the LAMP Stack*. Shelter Island, NY, USA: Manning Publications, June 2022 (cit. on pp. 371, 377).

[95] Jacques Carlier and Éric Pinson. "An Algorithm for Solving the Job-Shop Problem". *Management Science* 35(2):164–176, 1989. Catonsville, MD, USA: The Institute for Operations Research and the Management Sciences (INFORMS). ISSN: 0025-1909. doi:10.1287/mnsc.35.2.164. A branch and bound (B&B) method for the JSSP. (Cit. on p. 365).

[96] Stephan C. Carlson and The Editors of Encyclopaedia Britannica. *Golden Ratio*. Ed. by The Editors of Encyclopaedia Britannica. Chicago, IL, USA: Encyclopædia Britannica, Inc., Oct. 21, 2024. URL: https://www.britannica.com/science/golden-ratio (visited on 2024-12-14) (cit. on pp. 312, 363).

[97] Frances Carney Gies, Aakanksha Gaur, Erik Gregersen, Gloria Lotha, Emily Rodriguez, Veenu Setia, Gaurav Shukla, and Grace Young. "Fibonacci". In: *Encyclopaedia Britannica*. Ed. by The Editors of Encyclopaedia Britannica. Chicago, IL, USA: Encyclopædia Britannica, Inc., July 26, 2025. URL: https://www.britannica.com/biography/Fibonacci (visited on 2025-09-18) (cit. on p. 241).

[98] Eduardo Carvalho Pinto and Carola Doerr. "A Simple Proof for the Usefulness of Crossover in Black-Box Optimization". In: *15th International Conference on Parallel Problem Solving from Nature (PPSN XV)*. Vol. 2. Sept. 8–12, 2018, Coimbra, Portugal. Ed. by Anne Auger, Carlos M. Fonseca, Nuno Lourenço, Penousal Machado, Luís Paquete, and L. Darrell Whitley. Vol. 11102 of Lecture Notes in Computer Science (LNCS). Cham, Switzerland: Springer, 2018, pp. 29–41. ISSN: 0302-9743. ISBN: 978-3-319-99258-7. doi:10.1007/978-3-319-99259-4_3 (cit. on p. 369).

[99] Eduardo Carvalho Pinto and Carola Doerr. *Towards a More Practice-Aware Runtime Analysis of Evolutionary Algorithms*. arXiv.org: Computing Research Repository (CoRR) abs/1812.00493. Ithaca, NY, USA: Cornell Universiy Library, Dec. 3, 2018. doi:10.48550/arXiv.1812.00493. URL: https://arxiv.org/abs/1812.00493 (visited on 2025-08-08). arXiv:1812.00493v1 [cs.NE] 3 Dec 2018 (cit. on pp. 363, 369, 377).

[100] Oscar Castro, Pierrick Bruneau, Jean-Sébastien Sottet, and Dario Torregrossa. "Landscape of High-Performance Python to Develop Data Science and Machine Learning Applications". *ACM Computing Surveys (CSUR)* 56(3):65:1–65:30, 2024. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0360-0300. doi:10.1145/3617588 (cit. on p. 3).

[101] Santiago "bryant1410" Castro, Thomas J. "thomasjpfan" Fan, Joel "jnothman" Nothman, Roman "rth" Yurchak, Guillaume "glemaitre" Lemaitre, and Nicolas "NicolasHug" Hug. *Issue #16705: Support Typing*. San Francisco, CA, USA: GitHub Inc, Mar. 17–Aug. 31, 2020. URL: https://github.com/scikit-learn/scikit-learn/issues/16705 (visited on 2025-02-02) (cit. on p. 108).

[102] Antonio Cavacini. "Is the CE/BCE notation becoming a standard in scholarly literature?" *Scientometrics* 102(2):1661–1668, July 2015. London, England, UK: Springer Nature Limited. ISSN: 0138-9130. doi:10.1007/s11192-014-1352-1 (cit. on pp. 365, 366).

[103] Centennial, CO, USA: ACI Learning and Don Pezet. *Working with SSH*. Birmingham, England, UK: Packt Publishing Ltd, Mar. 2024. ISBN: 978-1-83588-600-7 (cit. on p. 378).

[104] Josh Centers. *Take Control of iOS 18 and iPadOS 18*. San Diego, CA, USA: Take Control Books, Dec. 2024. ISBN: 978-1-990783-55-5 (cit. on p. 370).

[105] Vladimír Černý. "Thermodynamical Approach to the Traveling Salesman Problem: An Efficient Simulation Algorithm". *Journal of Optimization Theory and Applications* 45(1):41–51, Jan. 1985. New York, NY, USA: Springer Science+Business Media, LLC. ISSN: 0022-3239. doi:10.1007/BF00940812 (cit. on p. 377).

[106] Noureddine Chabini and Rachid Beguenane. "FPGA-Based Designs of the Factorial Function". In: *IEEE Canadian Conference on Electrical and Computer Engineering (CCECE'2022)*. Sept. 18–20, 2022, Halifax, NS, Canada. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE), 2022, pp. 16–20. ISBN: 978-1-6654-8432-9. doi:10.1109/CCECE49351.2022.9918302 (cit. on pp. 163, 363).

[107] Donald D. Chamberlin. "50 Years of Queries". *Communications of the ACM (CACM)* 67(8):110–121, Aug. 2024. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/3649887. URL: https://cacm.acm.org/research/50-years-of-queries (visited on 2025-01-09) (cit. on pp. 177, 377, 378).

[108] Tony F. Chan, Gene H. Golub, and Randall J. LeVeque. "Algorithms for Computing the Sample Variance: Analysis and Recommendations". *The American Statistician* 37(3):242–247, Aug. 1983. London, England, UK: Taylor and Francis Ltd. ISSN: 0003-1305. doi:10.2307/2683386. URL: https://engineering.yale.edu/application/files/3817/3714/7395/tr222.pdf (visited on 2026-10-30) (cit. on pp. 380, 434).

[109] Eric Chapman. *GitHub Actions*. Birmingham, England, UK: Packt Publishing Ltd, Mar. 2024. ISBN: 978-1-80512-862-5 (cit. on p. 225).

[110] Bo Chen, Chris N. Potts, and Gerhard J. Woeginger. "A Review of Machine Scheduling: Complexity, Algorithms and Approximability". In: *Handbook of Combinatorial Optimization*. Ed. by Panos Miltiades Pardalos, Ding-Zhu Du, and Ronald Lewis Graham. 1st ed. Boston, MA, USA: Springer, 1998, pp. 1493–1641. ISBN: 978-1-4613-7987-4. doi:10.1007/978-1-4613-0303-9_25. See also pages 21–169 in volume 3/3 by Norwell, MA, USA: Kluwer Academic Publishers. Discusses, amongst others, that the JSSP is $\mathcal{NP}$-hard. (Cit. on pp. 370, 374).

[111] Stephen Chen, James Montgomery, and Antonio Bolufé-Röhler. "Measuring the Curse of Dimensionality and its Effects on Particle Swarm Optimization and Differential Evolution". *Applied Intelligence – The International Journal of Research on Intelligent Systems for Real Life Complex Problems* 42:514–526, Apr. 2015. London, England, UK: Springer Nature Limited. ISSN: 0924-669X. doi:10.1007/s10489-014-0613-2 (cit. on p. 373).

[112] Jiayang Chen (陈嘉阳), Zhize Wu (吴志泽), Sarah Louise Thomson, and Thomas Weise (汤卫思). "Frequency Fitness Assignment: Optimization Without Bias for Good Solution Outperforms Randomized Local Search on the Quadratic Assignment Problem". In: *16th International Joint Conference on Computational Intelligence (IJCCI'24)*. Nov. 20–22, 2024, Porto, Portugal. Ed. by Francesco Marcelloni, Kurosh Madani, Niki van Stein, and Joaquim Filipe. Porto, Portugal: SciTePress: Science and Technology Publications, Lda, 2024, pp. 27–37. ISSN: 2184-3236. ISBN: 978-989-758-721-4. doi:10.5220/0012888600003837 (cit. on pp. 368, 373, 376).

[113] T. C. Edwin Cheng, Bo Peng, and Zhipeng Lü. "A Hybrid Evolutionary Algorithm to Solve the Job Shop Scheduling Problem". *Annals of Operations Research* 242(2):223–237, July 2016. London, England, UK: Springer Nature Limited. ISSN: 0254-5330. doi:10.1007/s10479-013-1332-5. A MA utlizing TS (cit. on p. 372).

[114] Aakash Choudhury, Arjav Choudhury, Umashankar Subramanium, and S. Balamurugan. "HealthSaver: A Neural Network based Hospital Recommendation System Framework on Flask Webapplication with Realtime Database and RFID based Attendance System". *Journal of Ambient Intelligence and Humanized Computing* 13(10):4953–4966, Oct. 2022. London, England, UK: Springer Nature Limited. ISSN: 1868-5137. doi:10.1007/S12652-021-03232-7 (cit. on p. 368).

[115] Philippe Chrétienne, Edward G. Coffman, Jan Karel Lenstra, and Zhen Liu, eds. *Scheduling Theory and Its Applications*. Chichester, West Sussex, England, UK: John Wiley and Sons Ltd., Sept. 1995. ISBN: 978-0-471-94059-3 (cit. on p. 370).

[116] Christmas, FL, USA: Simon Sez IT. *Microsoft Access 2021 – Beginner to Advanced*. Birmingham, England, UK: Packt Publishing Ltd, Aug. 2023. ISBN: 978-1-83546-911-8 (cit. on pp. 371, 373).

[117] Philip Chrysopoulos. "The Ancient Greek Who Invented the World's First Steam Turbine". In: *Greek Reporter*. Cheyenne, WY, USA: Greekreporter COM LLC, Dec. 13, 2023. URL: https://greekreporter.com/2023/12/13/ancient-greek-world-first-steam-turbine (visited on 2025-09-02) (cit. on p. 155).

[118] Nestor M. Cid-Garcia and Yasmin A. Rios-Solis. "Positions and Covering: A Two-Stage Methodology to Obtain Optimal Solutions for the 2D-Bin Packing Problem". *PLoS ONE* 15(4):e0229358, Apr. 2020. San Francisco, CA, USA: Public Library of Science (PLOS). ISSN: 1932-6203. doi:10.1371/journal.pone.0229358. An exact method for the 2BP|R|F and the 2BP|O|F. (Cit. on p. 364).

[119] "Class Double". In: *Java® Platform, Standard Edition & Java Development Kit Version 22 API Specification*. Redwood Shores, CA, USA: Oracle Corporation, Apr. 9, 2024. URL: https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/lang/Double.html (visited on 2024-07-07) (cit. on pp. 44, 45).

[120] "Classes". In: *Python 3 Documentation*. *The Python Tutorial*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. Chap. 9. URL: https://docs.python.org/3/tutorial/classes.html (visited on 2025-09-19) (cit. on pp. 249, 250).

[121] David Clinton and Christopher Negus. *Ubuntu Linux Bible*. 10th ed. Bible Series. Chichester, West Sussex, England, UK: John Wiley and Sons Ltd., Nov. 10, 2020. ISBN: 978-1-119-72233-5 (cit. on pp. 9, 378, 380).

[122] "Cloning a Repository". In: *Repositories Documentation*. San Francisco, CA, USA: GitHub Inc, 2025. URL: https://docs.github.com/en/repositories/creating-and-managing-repositories/cloning-a-repository (visited on 2025-02-05) (cit. on p. 344).

[123] Edgar Frank "Ted" Codd. "A Relational Model of Data for Large Shared Data Banks". *Communications of the ACM (CACM)* 13(6):377–387, June 1970. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/362384.362685. URL: https://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf (visited on 2025-01-05) (cit. on p. 376).

[124] Coding Gears and Train Your Brain. *YAML Fundamentals for DevOps, Cloud and IaC Engineers*. Birmingham, England, UK: Packt Publishing Ltd, Mar. 2022. ISBN: 978-1-80324-243-9 (cit. on pp. 316, 382).

[125] Timothy W. Cole and Myung-Ja K. Han. *XML for Catalogers and Metadata Librarians (Third Millennium Cataloging)*. 1st ed. Dublin, OH, USA: Libraries Unlimited, May 23, 2013. ISBN: 978-1-59884-519-8 (cit. on pp. 316, 382).

[126] "`collections.abc` – Abstract Base Classes for Containers". In: *Python 3 Documentation*. *The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/library/collections.abc.html (visited on 2024-08-22) (cit. on p. 112).

[127] *Connecting to GitHub with SSH*. URL: https://docs.github.com/en/authentication/connecting-to-github-with-ssh (visited on 2025-02-05) (cit. on p. 346).

[128] "`contextlib` – Utilities for `with`-Statement Contexts". In: *Python 3 Documentation*. *The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/library/contextlib.html (visited on 2024-11-01) (cit. on pp. 208, 209).

[129] Stephen Arthur Cook. "The Complexity of Theorem-Proving Procedures". In: *Third Annual ACM Symposium on Theory of Computing (STOC'1971)*. May 3–5, 1971, Shaker Heights, OH, USA. Ed. by Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman. New York, NY, USA: Association for Computing Machinery (ACM), 1971, pp. 151–158. ISBN: 978-1-4503-7464-4. doi:10.1145/800157.805047 (cit. on pp. 372, 374).

[130] William John Cook. *Concorde TSP Solver*. Waterloo, ON, Canada: University of Waterloo, 1997. URL: https://math.uwaterloo.ca/tsp/concorde (visited on 2026-09-26) (cit. on p. 366).

[131] William John Cook. *In Pursuit of the Traveling Salesman: Mathematics at the Limits of Computation*. Princeton, NJ, USA: Princeton University Press, 2012–Nov. 9, 2014. ISBN: 978-0-691-16352-9 (cit. on pp. 366, 372).

[132] William John Cook. *World TSP*. Waterloo, ON, Canada: University of Waterloo, 2013–Oct. 25, 2025. URL: http://www.math.uwaterloo.ca/tsp/world (visited on 2026-01-03) (cit. on pp. 372, 379).

[133] Thomas H. Cormen, Charles E. Leiserson, Ronald Linn Rivest, and Clifford Stein. "Asymptotic Notation". In: *Introduction to Algorithms*. 3rd ed. Cambridge, MA, USA: MIT Press, 2009. Chap. 3.1, pp. 43–53. ISBN: 978-0-262-03384-8. Discusses $\boldsymbol{\Theta}$, $\mathcal{O}$, $\boldsymbol{\Omega}$, $\mathbf{o}$, and ω (cit. on pp. 374, 381).

[134] Thomas H. Cormen, Charles E. Leiserson, Ronald Linn Rivest, and Clifford Stein. *Introduction to Algorithms*. 3rd ed. Cambridge, MA, USA: MIT Press, 2009. ISBN: 978-0-262-03384-8 (cit. on pp. 124, 128, 285).

[135] Richard Crandall and Carl Pomerance. *Prime Numbers: A Computational Perspective*. 2nd ed. New York, NY, USA: Springer New York, Aug. 4, 2005. ISBN: 978-0-387-25282-7. doi:10.1007/0-387-28979-8 (cit. on pp. 147, 229, 241, 243).

[136] Scott A. Craver. *Results of the 2015 Underhanded C Contest: An Overview of NaN Poisoning Attacks*. Feb. 3, 2016. URL: http://underhanded-c.org/#nan (visited on 2025-07-24) (cit. on p. 47).

[137] Ignacio Samuel Crespo-Martínez, Adrián Campazas Vega, Ángel Manuel Guerrero-Higueras, Virginia Riego-Del Castillo, Claudia Álvarez-Aparicio, and Camino Fernández Llamas. "SQL Injection Attack Detection in Network Flow Data". *Computers & Security* 127:103093, Apr. 2023. Amsterdam, The Netherlands: Elsevier B.V. ISSN: 0167-4048. doi:10.1016/J.COSE.2023.103093 (cit. on p. 377).

[138] "`csv` – CSV File Reading and Writing". In: *Python 3 Documentation*. *The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/library/csv.html (visited on 2024-11-14) (cit. on p. 367).

[139] Cemre Çubukçuoğlu, Pirouz Nourian, M. Fatih Tasgetiren, I. Sevil Sariyildiz, and Shervin Azadi. "Hospital Layout Design Renovation as a Quadratic Assignment Problem with Geodesic Distances". *Journal of Building Engineering* 44:102952, Dec. 2021. Amsterdam, The Netherlands: Elsevier B.V. ISSN: 2352-7102. doi:10.1016/j.jobe.2021.102952 (cit. on p. 376).

[140] Christopher Cullen. "Learning from Liu Hui? A Different Way to Do Mathematics". *Notices of the American Mathematical Society* 49(7):783–790, Aug. 2002. Providence, RI, USA: American Mathematical Society (AMS). ISSN: 1088-9477. URL: https://www.ams.org/notices/200207/comm-cullen.pdf (visited on 2024-08-09) (cit. on pp. 88, 166, 363).

[141] Erik Dahlström, Patrick Dengler, Anthony Grasso, Chris Lilley, Cameron McCormack, Doug Schepers, Jonathan Watt, Jon Ferraiolo, Jun Fujisawa, and Dean Jackson, eds. *Scalable Vector Graphics (SVG) 1.1 (Second Edition)*. W3C Recommendation. Wakefield, MA, USA: World Wide Web Consortium (W3C), Aug. 16, 2011. URL: http://www.w3.org/TR/2011/REC-SVG11-20110816 (visited on 2024-12-17) (cit. on pp. 277, 316, 378).

[142] "Data Model". In: *Python 3 Documentation*. *The Python Language Reference*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. Chap. 3. URL: https://docs.python.org/3/reference/datamodel.html (visited on 2024-08-22) (cit. on pp. 112, 181).

[143] *Database Language SQL*. Tech. rep. ANSI X3.135-1986. Washington, D.C., USA: American National Standards Institute (ANSI), 1986 (cit. on p. 377).

[144] Joseph W. Dauben. "Archimedes and Liu Hui on Circles and Spheres". *Ontology Studies (Cuadernos de Ontología)* 10:21–38, 2010. Leioa, Bizkaia, Spain: Universidad del País Vasco / Euskal Herriko Unibertsitatea. ISSN: 1576-2270. URL: https://ddd.uab.cat/pub/ontstu/ontstu_a2010n10/ontstu_a2010n10p21.pdf (visited on 2026-11-03). Also discusses LIU Hui's (刘徽) method of approximating π (cit. on pp. 88, 166, 363).

[145] Matt David and Blake Barnhill. *How to Teach People SQL*. San Francisco, CA, USA: The Data School, Chart.io, Inc., Dec. 10, 2019–Apr. 10, 2023. URL: https://dataschool.com/how-to-teach-people-sql (visited on 2025-02-27) (cit. on p. 377).

[146] *Database Language SQL*. International Standard ISO 9075-1987. Geneva, Switzerland: International Organization for Standardization (ISO), 1987 (cit. on p. 377).

[147] Kenneth Alan De Jong. *Evolutionary Computation: A Unified Approach*. Vol. 4 of Complex Adaptive Systems. Cambridge, MA, USA: MIT Press, 2006. ISBN: 978-0-262-04194-2. URL: https://www.researchgate.net/publication/220740669 (visited on 2025-08-08) (cit. on pp. 368, 377).

[148] *Definition of Operations Research*. University of Western Ontario, London, ON, Canada: International Federation of Operational Research Societies (IFORS), 2020. URL: https://www.ifors.org/what-is-or (visited on 2026-01-01) (cit. on p. 374).

[149] Paul Deitel, Harvey Deitel, and Abbey Deitel. *Internet & World Wide Web: How to Program*. 5th ed. Hoboken, NJ, USA: Pearson Education, Inc., Nov. 2011. ISBN: 978-0-13-299045-5 (cit. on p. 381).

[150] Maxence Delorme, Manuel Iori, and Silvano Martello. "Bin Packing and Cutting Stock Problems: Mathematical Models and Exact Algorithms". *European Journal of Operational Research* 255(1):1–20, Nov. 2016. Amsterdam, The Netherlands: Elsevier B.V. ISSN: 0377-2217. doi:10.1016/j.ejor.2016.04.030 (cit. on p. 366).

[151] Ebru Demirkol, Sanjay V. Mehta, and Reha Uzsoy. "Benchmarks for Shop Scheduling Problems". *European Journal of Operational Research* 109(1):137–141, Aug. 1998. Amsterdam, The Netherlands: Elsevier B.V. ISSN: 0377-2217. doi:10.1016/S0377-2217(97)00019-2 (cit. on p. 370).

[152] Justin Dennison, Cherokee Boose, and Peter van Rysdam. *Intro to NumPy*. Centennial, CO, USA: ACI Learning. Birmingham, England, UK: Packt Publishing Ltd, June 2024. ISBN: 978-1-83620-863-1 (cit. on pp. 3, 330, 374).

[153] *Developer Survey 2019: Open Source Runtime Pains*. Vancouver, BC, Canada: ActiveState Software Inc., Apr. 30, 2019. URL: https://cdn.activestate.com/wp-content/uploads/2019/05/ActiveState-Developer-Survey-2019-Open-Source-Runtime-Pains.pdf (visited on 2024-12-29) (cit. on p. 2).

[154] Alfredo Deza and Noah Gift. *Testing In Python*. San Francisco, CA, USA: Pragmatic AI Labs, Feb. 2020. ISBN: 979-8-6169-6064-1 (cit. on pp. 5, 170, 375).

[155] Benjamin Dicken. *PyFlo – The Beginners Guide to Becoming a Python Programmer*. 2023. URL: https://pyflo.net (visited on 2025-08-28) (cit. on p. 6).

[156] "Dictionaries". In: *Python 3 Documentation. The Python Language Reference*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. Chap. 3.2.7.1. URL: https://docs.python.org/3/reference/datamodel.html#dictionaries (visited on 2024-08-27) (cit. on p. 129).

[157] Miroslav Dimitrov. *New Classes of Binary Sequences with High Merit Factor*. arXiv.org: Computing Research Repository (CoRR). Ithaca, NY, USA: Cornell Universiy Library, June 24, 2022. doi:10.48550/arXiv.2206.12070. URL: https://arxiv.org/pdf/2206.12070 (visited on 2026-09-21). arXiv:2206.12070v1 [cs.IT] 24 Jun 2022 (cit. on p. 371).

[158] Miroslav Dimitrov, Tsonka Baitcheva, and Nikolay Nikolov. "Efficient Generation of Low Autocorrelation Binary Sequences". *IEEE Signal Processing Letters* 27:341–345, 2020. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE). ISSN: 1558-2361. doi:10.1109/LSP.2020.2972127 (cit. on p. 371).

[159] Slobodan Dmitrović. *Modern C for Absolute Beginners: A Friendly Introduction to the C Programming Language*. New York, NY, USA: Apress Media, LLC, Mar. 2024. ISBN: 979-8-8688-0224-9 (cit. on p. 366).

[160] "Doctest – Test Interactive Python Examples". In: *Python 3 Documentation. The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/library/doctest.html (visited on 2024-11-07) (cit. on pp. 226, 227, 362, 367).

[161] Benjamin Doerr and Carola Doerr. "Optimal Static and Self-Adjusting Parameter Choices for the $(1+(\lambda,\lambda))$ Genetic Algorithm". *Algorithmica* 80(5):1658–1709, May 2018. London, England, UK: Springer Nature Limited. ISSN: 0178-4617. doi:10.1007/s00453-017-0354-9 (cit. on p. 377).

[162] Benjamin Doerr, Carola Doerr, and Franziska Ebel. "From Black-Box Complexity to Designing New Genetic Algorithms". *Theoretical Computer Science* 567:87–104, Feb. 2015. Amsterdam, The Netherlands: Elsevier B.V. ISSN: 0304-3975. doi:10.1016/J.TCS.2014.11.028 (cit. on p. 374).

[163] Carola Doerr, Furong Ye, Naama Horesh, Hao Wang, and Ofer M. ShirThomas Bäck. "Benchmarking Discrete Optimization Heuristics with IOHprofiler". In: *21st Genetic and Evolutionary Computation Conference Companion (GECCO'2019)*. July 13–17, 2019, Prague, Czech Republic. Ed. by Manuel López-Ibáñez, Anne Auger, and Thomas Stützle. New York, NY, USA: Association for Computing Machinery (ACM), 2019, pp. 1798–1806. ISBN: 978-1-4503-6748-6. doi:10.1145/3319619.3326810. See also [164, 165] (cit. on p. 400).

[164] Carola Doerr, Furong Ye, Naama Horesh, Hao Wang, and Ofer M. ShirThomas Bäck. "Benchmarking Discrete Optimization Heuristics with IOHprofiler". *Applied Soft Computing (ASOC)* 88:106027, Mar. 2020. Amsterdam, The Netherlands: Elsevier B.V. ISSN: 1568-4946. doi:10.1016/j.asoc.2019.106027. See also [163, 165] (cit. on pp. 364, 370, 374, 400).

[165] Carola Doerr, Furong Ye, Naama Horesh, Hao Wang, and Ofer M. ShirThomas Bäck. "Benchmarking Discrete Optimization Heuristics with IOHprofiler". (abs/1912.09237), Jan. 31, 2021. doi:10.48550/arXiv.1912.09237. URL: https://arxiv.org/abs/1912.09237 (visited on 2024-06-27). arXiv:1912.09237v2 [cs.NE] 31 Jan 2021, see also [163, 164] (cit. on p. 400).

[166] Pooyan Doozandeh and Frank E. Ritter. "Some Tips for Academic Writing and Using Microsoft Word". *XRDS: Crossroads, The ACM Magazine for Students* 26(1):10–11, Aut. 2019. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 1528-4972. doi:10.1145/3351470 (cit. on pp. 249, 316, 373).

[167] Marco Dorigo, Mauro Birattari, and Thomas Stützle. "Ant Colony Optimization". *IEEE Computational Intelligence Magazine (CIM)* 1(4):28–39, Nov. 2006. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE). ISSN: 1556-603X. doi:10.1109/MCI.2006.329691. URL: https://www.researchgate.net/publication/308953674 (visited on 2025-07-25) (cit. on p. 364).

[168] Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni. "Ant System: Optimization by a Colony of Cooperating Agents". *IEEE Transactions on Systems, Man, and Cybernetics – Part B: Cybernetics* 26(1):29–41, Feb. 1996. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE). ISSN: 1083-4419. doi:10.1109/3477.484436. URL: https://www.researchgate.net/publication/5589170 (visited on 2025-07-26) (cit. on p. 364).

[169] Marco Dorigo and Thomas Stützle. *Ant Colony Optimization*. Cambridge, MA, USA: MIT Press, 2004. ISBN: 978-0-262-04219-2 (cit. on p. 364).

[170] Ingy döt Net, Tina Müller, Pantelis Antoniou, Eemeli Aro, Thomas Smith, Oren Ben-Kiki, and Clark C. Evans. *YAML Ain't Markup Language (YAML™) version 1.2*. Revision 1.2.2. Seattle, WA, USA: YAML Language Development Team, Oct. 1, 2021. URL: https://yaml.org/spec/1.2.2 (visited on 2025-01-05) (cit. on pp. 316, 382).

[171] Johann Dréo, Alain Pétrowski, Patrick Siarry, and Éric D. Taillard. *Metaheuristics for Hard Optimization – Methods and Case Studies*. Berlin/Heidelberg, Germany: Springer-Verlag GmbH Germany, 2006. ISBN: 978-3-540-23022-9. doi:10.1007/3-540-30966-7. Discusses SA, TS, EAs, and Ant Colony Optimization (ACO). (Cit. on p. 376).

[172] Stefan Droste, Thomas Jansen, and Ingo Wegener. "On the Analysis of the $(1+1)$ Evolutionary Algorithm". *Theoretical Computer Science* 276(1-2):51–81, Apr. 2002. Amsterdam, The Netherlands: Elsevier B.V. ISSN: 0304-3975. doi:10.1016/S0304-3975(01)00182-7 (cit. on pp. 363, 364, 371, 379).

[173] Stefan Droste, Thomas Jansen, and Ingo Wegener. "Upper and Lower Bounds for Randomized Search Heuristics in Black-Box Optimization". *Theoretical Computer Science* 39(4):525–544, July 2006. Amsterdam, The Netherlands: Elsevier B.V. ISSN: 0304-3975. doi:10.1007/S00224-004-1177-Z (cit. on p. 374).

[174] Paul Duplys and Roland Schmitz. *TLS Cryptography In-Depth*. Birmingham, England, UK: Packt Publishing Ltd, Jan. 2024. ISBN: 978-1-80461-195-1 (cit. on p. 379).

[175] Jacques Dutka. "The Early History of the Factorial Function". *Archive for History of Exact Sciences* 43(3):225–249, Sept. 1991. Berlin/Heidelberg, Germany: Springer-Verlag GmbH Germany. ISSN: 0003-9519. doi:10.1007/BF00389433. Communicated by Umberto Bottazzini (cit. on pp. 163, 363).

[176] Harald Dyckhoff and Ute Finke. *Cutting and Packing in Production and Distribution: A Typology and Bibliography*. Contributions to Management Science. Heidelberg, Baden-Württemberg, Germany: Physica-Verlag, 1992. ISSN: 1431-1941. ISBN: 978-3-642-63487-1. doi:10.1007/978-3-642-58165-6 (cit. on p. 366).

[177] Russell J.T. Dyer. *Learning MySQL and MariaDB*. Sebastopol, CA, USA: O'Reilly Media, Inc., Mar. 2015. ISBN: 978-1-4493-6290-4 (cit. on pp. 372, 373).

[178] Kelly Easton, George L. Nemhauser, and Michael A. Trick. "The Traveling Tournament Problem Description and Benchmarks". In: *7th International Conference on Principles and Practice of Constraint Programming (CP'01)*. Nov. 26–Dec. 1, 2001, Paphos, Cyprus. Ed. by Toby Walsh. Vol. 2239 of Lecture Notes in Computer Science (LNCS). Berlin/Heidelberg, Germany: Springer-Verlag GmbH Germany, 2001, pp. 580–584. ISSN: 0302-9743. ISBN: 978-3-540-42863-3. doi:10.1007/3-540-45578-7_43 (cit. on p. 379).

[179] Phillip J. Eby. *Python Web Server Gateway Interface v1.0.1*. Python Enhancement Proposal (PEP) 3333. Beaverton, OR, USA: Python Software Foundation (PSF), Sept. 26–Oct. 4, 2010. URL: https://peps.python.org/pep-3333 (visited on 2025-03-04) (cit. on p. 368).

[180] *ECMAScript Language Specification*. Standard ECMA-262, 3rd Edition. Geneva, Switzerland: Ecma International, Dec. 1999. URL: https://ecma-international.org/wp-content/uploads/ECMA-262_3rd_edition_december_1999.pdf (visited on 2024-12-15) (cit. on p. 370).

[181] "infinitesimal". In: *Merriam-Webster: America's Most Trusted Dictionary*. Ed. by Editors of Merriam-Webster. Sept. 4, 2025. URL: https://www.merriam-webster.com/dictionary/infinitesimal (visited on 2025-09-08) (cit. on pp. 181, 370).

[182] Alwalid N. Elshafei. "Hospital Layout as a Quadratic Assignment Problem". *The Journal of the Operational Research Society (JORS)* 28(1):167–179, Apr. 1977. London, England, UK: Taylor and Francis Ltd. and London, England, UK: Springer Nature Limited. ISSN: 0160-5682. doi:10.1057/jors.1977.29 (cit. on p. 376).

[183] James F. Epperson. *An Introduction to Numerical Methods and Analysis*. 2nd ed. John Wiley and Sons Ltd., Oct. 2013. ISBN: 978-1-118-36759-9 (cit. on pp. 181, 185).

[184] "Escape Sequences". In: *Python 3 Documentation*. *The Python Language Reference*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. Chap. 2.4.1.1. URL: https://docs.python.org/3/reference/lexical_analysis.html#escape-sequences (visited on 2025-08-05) (cit. on pp. 77, 367).

[185] Bernhard Eschermann and Hans-Joachim Wunderlich. "Optimized Synthesis of Self-Testable Finite State Machines". In: *20th International Symposium Fault-Tolerant Computing (FTCS'1990): Digest of Papers*. June 26–28, 1990, Newcastle Upon Tyne, England, UK. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE), 1990. ISBN: 978-0-8186-2051-5. doi:10.1109/FTCS.1990.89393 (cit. on p. 376).

[186] Euclid of Alexandria (Εὐκλείδης). *Euclid's Elements of Geometry (Στοιχεῖα). The Greek Text of J.L. Heiberg (1883-1885) from Euclidis Elementa, Edidit et Latine Interpretatus est I.L. Heiberg in Aedibus B. G. Teubneri, 1883-1885. Edited, and provided with a modern English translation, by Richard Fitzpatrick*. Vol. 7: *Elementary Number Theory*. Ed. by Richard Fitzpatrick. Trans. by Johan Ludvig Heiberg. revised and corrected. Austin, TX, USA: The University of Texas at Austin, 2008. ISBN: 978-0-615-17984-1. URL: https://farside.ph.utexas.edu/Books/Euclid/Elements.pdf (visited on 2024-09-30) (cit. on p. 164).

[187] Euclid of Alexandria (Εὐκλείδης). *Euclid's Elements of Geometry (Στοιχεῖα). The Greek Text of J.L. Heiberg (1883-1885) from Euclidis Elementa, Edidit et Latine Interpretatus est I.L. Heiberg in Aedibus B. G. Teubneri, 1883-1885. Edited, and provided with a modern English translation, by Richard Fitzpatrick*. Ed. by Richard Fitzpatrick. Trans. by Johan Ludvig Heiberg. revised and corrected. Austin, TX, USA: The University of Texas at Austin, 2008. ISBN: 978-0-615-17984-1. URL: https://farside.ph.utexas.edu/Books/Euclid/Elements.pdf (visited on 2024-09-30) (cit. on pp. 312, 363).

[188] Leonhard Euler. "An Essay on Continued Fractions". Trans. by Myra F. Wyman and Bostwick F. Wyman. *Mathematical Systems Theory* 18(1):295–328, Dec. 1985. New York, NY, USA: Springer Science+Business Media, LLC. ISSN: 1432-4350. doi:10.1007/BF01699475. URL: https://www.researchgate.net/publication/301720080 (visited on 2024-09-24). Translation of [189]. (Cit. on p. 402).

[189] Leonhard Euler. "De Fractionibus Continuis Dissertation". *Commentarii Academiae Scientiarum Petropolitanae* 9:98–137, 1737–1744. Petropolis (St. Petersburg), Russia: Typis Academiae. URL: https://scholarlycommons.pacific.edu/cgi/viewcontent.cgi?article=1070 (visited on 2024-09-24). See [188] for a translation. (Cit. on pp. 367, 402).

[190] Merran Evans, Nicholas Hastings, and Brian Peacock. *Statistical Distributions*. 3rd ed. Chichester, West Sussex, England, UK: Wiley Interscience, June 2000. ISBN: 978-0-471-37124-3 (cit. on pp. 177, 184, 375).

[191] Steve Fanning, Vasudev Narayanan, "flywire", Olivier Hallot, Jean Hollis Weber, Jenna Sargent, Pulkit Krishna, Dan Lewis, Peter Schofield, Jochen Schiffers, Robert Großkopf, Jost Lange, Martin Fox, Hazel Russman, Steve Schwettman, Alain Romedenne, Andrew Pitonyak, Jean-Pierre Ledure, Drew Jensen, and Randolph Gam. *Base Guide 7.3. Revision 1. Based on LibreOffice 7.3 Community*. Berlin, Germany: The Document Foundation, Aug. 2022. URL: https://books.libreoffice.org/en/BG73/BG73-BaseGuide.pdf (visited on 2025-01-13) (cit. on p. 371).

[192] Greg Fee. *The Golden Ratio: (1+sqrt(5))/2 to 20000 Places*. Salt Lake City, UT, USA: Project Gutenberg Literary Archive Foundation, Aug. 1, 1996. URL: https://www.gutenberg.org/ebooks/633 (visited on 2024-12-14) (cit. on p. 312).

[193] Thomas A. Feo and Mauricio G. C. Resende. "Greedy Randomized Adaptive Search Procedures". *Journal of Global Optimization* 6(2):109–133, Mar. 1995. London, England, UK: Springer Nature Limited. ISSN: 0925-5001. doi:10.1007/BF01096763. Introduces Greedy Randomized Adaptive Search Procedures (GRASPs) and applies them to the set covering problem, among others, and lists many problem domains. (Cit. on p. 369).

[194] Luca Ferrari and Enrico Pirozzi. *Learn PostgreSQL*. 2nd ed. Birmingham, England, UK: Packt Publishing Ltd, Oct. 2023. ISBN: 978-1-83763-564-1 (cit. on p. 375).

[195] Paola Festa and Mauricio G. C. Resende. "An Annotated Bibliography of GRASP—Part I: Algorithms". *International Transactions on Operational Research (ITOR)* 16(1):1–24, Jan. 2009. Chichester, West Sussex, England, UK: John Wiley and Sons Ltd. ISSN: 0969-6016. doi:10.1111/j.1475-3995.2009.00663.x. See also [196] (cit. on pp. 369, 403).

[196] Paola Festa and Mauricio G. C. Resende. "An Annotated Bibliography of GRASP—Part II: Applications". *International Transactions on Operational Research (ITOR)* 16(2):131–172, Mar. 2009. Chichester, West Sussex, England, UK: John Wiley and Sons Ltd. ISSN: 0969-6016. doi:10.1111/j.1475-3995.2009.00664.x. See also [195] (cit. on pp. 369, 403).

[197] Roy T. Fielding, Jim Gettys, Jeffrey C. Mogul, Henrik Frystyk Nielsen, and Tim Berners-Lee. *Hypertext Transfer Protocol -- HTTP/1.1*. Request for Comments (RFC) 2068. Wilmington, DE, USA: Internet Engineering Task Force (IETF), Jan. 1997. URL: https://www.ietf.org/rfc/rfc2068.txt (visited on 2025-02-05) (cit. on p. 369).

[198] Roy T. Fielding, Mark Nottingham, and Julian F. Reschke. *HTTP Semantics*. Request for Comments (RFC) 9110. Wilmington, DE, USA: Internet Engineering Task Force (IETF), June 2022. URL: https://www.ietf.org/rfc/rfc9110.txt (visited on 2025-02-05) (cit. on p. 369).

[199] Michael Filaseta. "The Transcendence of e and π". In: *Math 785: Transcendental Number Theory*. Columbia, SC, USA: University of South Carolina, Spr. 2011. Chap. 6. URL: https://people.math.sc.edu/filaseta/gradcourses/Math785/Math785Notes6.pdf (visited on 2024-07-05) (cit. on pp. 363, 367).

[200] Simon Fischer. "A Polynomial Upper Bound for a Mutation-Based Algorithm on the Two-Dimensional Ising Model". In: *6th Genetic and Evolutionary Computation Conference (GECCO'2004), Part I*. June 26–30, 2004, Seattle, WA, USA. Ed. by Kalyanmoy Deb, Riccardo Poli, Wolfgang Banzhaf, Hans-Georg Beyer, Edmund K. Burke, Paul J. Darwen, Dipankar Dasgupta, Dario Floreano, James A. Foster, Mark Harman, Owen Holland, Pier Luca Lanzi, Lee Spector, Andrea Tettamanzi, Dirk Thierens, and Andrew M. Tyrrell. Vol. 3102 of Lecture Notes in Computer Science (LNCS). Berlin/Heidelberg, Germany: Springer-Verlag GmbH Germany, 2004, pp. 1100–1112. ISSN: 0302-9743. ISBN: 978-3-540-22344-3. doi:10.1007/978-3-540-24854-5_108 (cit. on p. 370).

[201] Simon Fischer and Ingo Wegener. "The One-Dimensional Ising Model: Mutation versus Recombination". *Theoretical Computer Science* 344(2-3):208–225, Nov. 2005. Amsterdam, The Netherlands: Elsevier B.V. ISSN: 0304-3975. doi:10.1016/J.TCS.2005.04.002 (cit. on p. 370).

[202] Henry Fisher and Gerald L. Thompson. "Probabilistic Learning Combinations of Local Job-Shop Scheduling Rules". In: *Industrial Scheduling*. Ed. by John F. Muth and Gerald L. Thompson. Englewood Cliffs, NJ, USA: Prentice-Hall, 1963. Chap. 3.2, pp. 225–251 (cit. on p. 370).

[203] Charles Fleurent and Jacques A. Ferland. "Genetic Hybrids for the Quadratic Assignment Problem". In: *Quadratic Assignment and Related Problems, Proceedings of a DIMACS Workshop*. May 20–21, 1993, New Brunswick, NJ, USA: Rutgers University. Ed. by Panos Miltiades Pardalos and Henry Wolkowicz. Providence, RI, USA: American Mathematical Society (AMS), 1994, pp. 173–187. ISBN: 978-0-8218-6607-8. A MA using the TS from [549] as local search performs well. (Cit. on pp. 372, 376).

[204] "Floating-Point Arithmetic: Issues and Limitations". In: *Python 3 Documentation. The Python Tutorial*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. Chap. 15. URL: https://docs.python.org/3/tutorial/floatingpoint.html (visited on 2024-12-08) (cit. on pp. 38, 259).

[205] *PDF 32000-1:2008 – Document Management – Portable Document Format – Part 1: PDF 1.7*. 1st ed. San Jose, CA, USA: Adobe Systems Incorporated, July 1, 2008. URL: https://pdf-lib.js.org/assets/with_large_page_count.pdf (visited on 2024-12-12) (cit. on pp. 249, 277, 375).

[206] David Fowler and Eleanor Robson. "Square Root Approximations in Old Babylonian Mathematics: YBC 7289 in Context". *Historia Mathematica* 25(4):366–378, Nov. 1998. Amsterdam, The Netherlands: Elsevier B.V. ISSN: 0315-0860. doi:10.1006/hmat.1998.2209. URL: https://www.ux1.eiu.edu/~cfcid/Classes/4900/Class%20Notes/Babylonian%20Approximations.pdf (visited on 2024-09-25). Article NO. HM982209 (cit. on p. 155).

[207] "`fractions` – Rational Numbers". In: *Python 3 Documentation*. *The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/library/fractions.html (visited on 2024-12-12) (cit. on p. 290).

[208] Tobias Friedrich, Pietro Simone Oliveto, Dirk Sudholt, and Carsten Witt. "Analysis of Diversity-Preserving Mechanisms for Global Exploration". *Evolutionary Computation* 17(4):455–476, Win. 2009. Cambridge, MA, USA: MIT Press. ISSN: 1063-6560. doi:10.1162/evco.2009.17.4.17401 (cit. on p. 380).

[209] Tobias Friedrich, Francesco Quinzan, and Markus Wagner. "Escaping Large Deceptive Basins of Attraction with Heavy-Tailed Mutation Operators". In: *20th Genetic and Evolutionary Computation Conference (GECCO'2018)*. July 15–19, 2018, Kyōto, Kyōto-fu, Japan. Ed. by Hernán E. Aguirre and Keiki Takadama. New York, NY, USA: Association for Computing Machinery (ACM), 2018, pp. 293–300. ISBN: 978-1-4503-5618-3. doi:10.1145/3205455.3205515 (cit. on pp. 371, 380).

[210] "Formatted String Literals". In: *Python 3 Documentation*. *The Python Tutorial*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. Chap. 7.1.1. URL: https://docs.python.org/3/tutorial/inputoutput.html#formatted-string-literals (visited on 2024-07-25) (cit. on pp. 73, 368).

[211] Toru Fujita, Koji Nakano, and Yasuaki Ito. "Bulk Execution of Euclidean Algorithms on the CUDA-Enabled GPU". *International Journal of Networking and Computing (IJNC)* 6(1):42–63, Jan. 2016. Higashi-Hiroshima, Hiroshima-ken, Japan: Department of Information Engineering, Hiroshima University. ISSN: 2185-2839. URL: http://www.ijnc.org (visited on 2024-09-28) (cit. on p. 164).

[212] Kevin P. Gaffney, Martin Prammer, Laurence C. Brasfield, D. Richard Hipp, Dan R. Kennedy, and Jignesh M. Patel. "SQLite: Past, Present, and Future". *Proceedings of the VLDB Endowment (PVLDB)* 15(12):3535–3547, Aug. 2022. Irvine, CA, USA: Very Large Data Bases Endowment Inc. ISSN: 2150-8097. doi:10.14778/3554821.3554842. URL: https://www.vldb.org/pvldb/vol15/p3535-gaffney.pdf (visited on 2025-01-12). All papers in this issue were presented at the *48th International Conference on Very Large Data Bases (VLDB 2022)*, 9 5-9, 2022, hybrid/Sydney, NSW, Australia (cit. on pp. 177, 378).

[213] Pablo Galindo and Brett Cannon. *No More Bare Excepts*. Python Enhancement Proposal (PEP) 760. Beaverton, OR, USA: Python Software Foundation (PSF), Oct. 2–9, 2024. URL: https://peps.python.org/pep-0760 (visited on 2025-09-10). Status: Withdrawn, due to backward compatibility breakage, see https://discuss.python.org/t/pep-760-no-more-bare-excepts visited on 2024-09-10. But it has many good references on why we should not just `except` any possible `Exceptions`. (Cit. on pp. 197, 358).

[214] José E. Gallardo, Carlos Cotta, and Antonio J. Fernández. "A Memetic Algorithm for the Low Autocorrelation Binary Sequence Problem". In: *9th Genetic and Evolutionary Computation Conference (GECCO'2007)*. July 7–11, 2007, London, England, UK. Ed. by Hod Lipson. New York, NY, USA: Association for Computing Machinery (ACM), 2007, pp. 1226–1233. ISBN: 978-1-59593-697-4. doi:10.1145/1276958.1277195 (cit. on p. 371).

[215] Jonas Gamalielsson and Björn Lundell. "Long-Term Sustainability of Open Source Software Communities beyond a Fork: A Case Study of LibreOffice". In: *8th IFIP WG 2.13 International Conference on Open Source Systems: Long-Term Sustainability OSS'2012*. Sept. 10–13, 2012, Hammamet, Tunisia. Ed. by Imed Hammouda, Björn Lundell, Tommi Mikkonen, and Walt Scacchi. Vol. 378. Vol. 378 of IFIP Advances in Information and Communication Technology (IFIPAICT). Berlin/Heidelberg, Germany: Springer-Verlag GmbH Germany, 2012, pp. 29–

47. ISSN: 1868-4238. ISBN: 978-3-642-33441-2. doi:10.1007/978-3-642-33442-9_3 (cit. on pp. 249, 316, 371).

[216] Alessandro F. Garcia, andCecília M. F. Rubira, Alexander B. Romanovsky, and Jie Xu. "A Comparative Study of Exception Handling Mechanisms for Building Dependable Object-Oriented Software". *Journal of Systems and Software* 59(2):197–222, Nov. 15, 2001. Amsterdam, The Netherlands: Elsevier B.V. ISSN: 0164-1212. doi:10.1016/S0164-1212(01)00062-0 (cit. on p. 186).

[217] Michael R. Garey and David Stifler Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman and Company, 1979. ISBN: 978-0-7167-1045-5 (cit. on p. 379).

[218] Bhavesh Gawade. "Mastering F-Strings in Python: Efficient String Handling in Python Using Smart F-Strings". In: *C O D E B*. Mumbai, Maharashtra, India: Code B Solutions Pvt Ltd, Apr. 25–June 3, 2025. URL: https://code-b.dev/blog/f-strings-in-python (visited on 2025-08-04) (cit. on pp. 76, 368).

[219] Mitsuo Gen, Yasuhiro Tsujimura, and Erika Kubota. "Solving Job-Shop Scheduling Problems by Genetic Algorithm". In: *Humans, Information and Technology: Proceedings of the 1994 IEEE International Conference on Systems, Man and Cybernetics (SMC'1994)*. Vol. 2. Oct. 2–5, 1994, San Antonio, TX, USA. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE), 1994, pp. 1577–1582. ISBN: 978-0-7803-2129-8. doi:10.1109/ICSMC.1994.400072. Proposes a permutations-with-repetitions based encoding for the JSSP. (Cit. on pp. 368, 370, 406).

[220] GitHub Staff. *Octoverse: AI leads Python to top language as the number of global developers surges*. San Francisco, CA, USA: GitHub Inc, Oct. 29, 2024. URL: https://github.blog/news-insights/octoverse/octoverse-2024 (visited on 2025-01-07) (cit. on p. 3).

[221] *Python 3 Documentation*. *Glossary*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/glossary.html (visited on 2025-09-16).

[222] Fred Glover. "Tabu Search – Part I". *ORSA Journal on Computing* 1(3):190–206, Sum. 1989. Catonsville, MD, USA: The Institute for Operations Research and the Management Sciences (INFORMS). ISSN: 0899-1499. doi:10.1287/ijoc.1.3.190 (cit. on p. 379).

[223] Fred Glover. "Tabu Search – Part II". *ORSA Journal on Computing* 2(1):4–32, Win. 1990. Catonsville, MD, USA: The Institute for Operations Research and the Management Sciences (INFORMS). ISSN: 0899-1499. doi:10.1287/ijoc.2.1.4 (cit. on p. 379).

[224] Fred Glover and Gary A. Kochenberger, eds. *Handbook of Metaheuristics*. Vol. 57 of International Series in Operations Research & Management Science (ISOR). Cham, Switzerland: Springer, 2003. ISSN: 0884-8289. ISBN: 978-0-306-48056-0. doi:10.1007/b101874 (cit. on p. 372).

[225] Fred Glover, Éric D. Taillard, and Dominique de Werra. "A User's Guide to Tabu Search". *Annals of Operations Research* 41(1):3–28, Mar. 1993. London, England, UK: Springer Nature Limited. ISSN: 0254-5330. doi:10.1007/BF02078647. Special issue on TS. (Cit. on p. 379).

[226] Adam Gold. *Python Hash Tables Under the Hood*. San Francisco, CA, USA: GitHub Inc, June 30, 2020. URL: https://adamgold.github.io/posts/python-hash-tables-under-the-hood (visited on 2024-12-09) (cit. on pp. 285, 286).

[227] David Goldberg. "What Every Computer Scientist Should Know About Floating-Point Arithmetic". *ACM Computing Surveys (CSUR)* 23(1):5–48, Mar. 1991. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0360-0300. doi:10.1145/103162.103163. URL: https://pages.cs.wisc.edu/~david/courses/cs552/S12/handouts/goldberg-floating-point.pdf (visited on 2025-09-03) (cit. on pp. 37, 38, 46, 157, 175, 259, 357, 413).

[228] David Edward Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1989. ISBN: 978-0-201-15767-3 (cit. on pp. 368, 377).

[229] José Fernando Gonçalves and Mauricio G. C. Resende. "A Biased Random Key Genetic Algorithm for 2BP and 3D Bin Packing Problems". *International Journal of Production Economics (IJPE)* 145(2):500–510, Oct. 2013. Amsterdam, The Netherlands: Elsevier B.V. ISSN: 0925-5273. doi:10.1016/j.ijpe.2013.04.019. Applies the Biased Random Key Genetic Algorithm (BRKGA) to the 2BP|R|F and 2BP|O|F as well as 3D packing. Continuous vectors are translated to permutations and orientation data which is then the input of a placement heuristic. (Cit. on pp. 364, 366).

[230] José Fernando Gonçalves and Mauricio G. C. Resende. "An Extended Akers Graphical Method with a Biased Random-Key Genetic Algorithm for Job-Shop Scheduling". *International Transactions on Operational Research (ITOR)* 21(2):215–246, Mar. 2014. Chichester, West Sussex, England, UK: John Wiley and Sons Ltd. ISSN: 0969-6016. doi:10.1111/itor.12044. URL: http://mauricio.resende.info/doc/brkga-jss2011.pdf (visited on 2026-09-30). The BRKGA basically is a MA using a continuous encoding of permutations that are decoded to schedules using the method from [58, 59, 219, 508] and then refined via a TS. (Cit. on pp. 366, 370).

[231] José Fernando Gonçalves and Mauricio G. C. Resende. "Biased Random-Key Genetic Algorithms for Combinatorial Optimization". *Journal of Heuristics* 17:487–525, Oct. 2010. London, England, UK: Springer Nature Limited. ISSN: 1381-1231. doi:10.1007/s10732-010-9143-1 (cit. on p. 366).

[232] David Goodger and Guido van Rossum. *Docstring Conventions*. Python Enhancement Proposal (PEP) 257. Beaverton, OR, USA: Python Software Foundation (PSF), May 29–June 13, 2001. URL: https://peps.python.org/pep-0257 (visited on 2024-07-27) (cit. on pp. 68, 79, 119, 355, 356, 367).

[233] Dan Goodin. "AI-generated code could be a disaster for the software supply chain. Here's why." In: *Ars Technica*. Ed. by Ken Fisher and Jon Stokes. New York, NY, USA: Condé Nast, Apr. 29, 2025. URL: https://arstechnica.com/security/2025/04/ai-generated-code-could-be-a-disaster-for-the-software-supply-chain-heres-why (visited on 2025-05-08) (cit. on p. 60).

[234] Michael T. Goodrich. *A Gentle Introduction to NP-Completeness*. Irvine, CA, USA: University of California, Irvine, Apr. 2022. URL: https://ics.uci.edu/~goodrich/teach/cs165/notes/NPComplete.pdf (visited on 2025-08-01) (cit. on p. 374).

[235] Michael Goodwin. *reStructuredText Docstring Format*. Tech. rep. PEP287. Mar. 25–Apr. 2, 2002. URL: https://peps.python.org/pep-0287 (visited on 2024-12-12) (cit. on pp. 254, 358).

[236] Michael Goodwin. *What is an API?* Armonk, NY, USA: International Business Machines Corporation (IBM), Apr. 9, 2024. URL: https://www.ibm.com/topics/api (visited on 2024-12-12) (cit. on p. 365).

[237] Olaf Górski. "Why f-strings are awesome: Performance of different string concatenation methods in Python". In: *DEV Community*. Sacramento, CA, USA: DEV Community Inc., Nov. 8, 2022. URL: https://dev.to/grski/performance-of-different-string-concatenation-methods-in-python-why-f-strings-are-awesome-2e97 (visited on 2025-08-04) (cit. on pp. 76, 368).

[238] Ronald Lewis Graham, Eugene Leighton Lawler, Jan Karel Lenstra, and Alexander Hendrik George Rinnooy Kan. "Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey". Annals of Discrete Mathematics (5):287–326. Ed. by Peter Ladislaw Hammer, Ellis Lane Johnson, and Bernhard H. Korte, 1979. ISSN: 0167-5060. doi:10.1016/S0167-5060(08)70356-X. An early review on JSSPs and FSSPs. (Cit. on pp. 368, 370).

[239] Linda Grandell, Mia Peltomäki, Ralph-Johan Back, and Tapio Salakoski. "Why complicate things? Introducing Programming in High School using Python". In: *8th Australasian Conference on Computing Education (ACE'2006)*. Jan. 16–19, 2006, Hobart, TAS, Australia. Ed. by Denise Tolhurst and Samuel Mann. Vol. 52. New York, NY, USA: Association for Computing Machinery (ACM), 2006, pp. 71–80. ISBN: 978-1-920682-34-7. doi:10.5555/1151869.1151880 (cit. on p. 4).

[240] Dawn Griffiths. *Excel Cookbook – Receipts for Mastering Microsoft Excel*. Sebastopol, CA, USA: O'Reilly Media, Inc., May 2024. ISBN: 978-1-0981-4332-9 (cit. on p. 373).

[241] Ilya Grigorik. *HTTP Protocols*. Sebastopol, CA, USA: O'Reilly Media, Inc., Dec. 2017. ISBN: 978-1-4920-3046-1 (cit. on p. 369).

[242] Martin Grötschel, Michael Jünger, and Gerhard Reinelt. "Optimal Control of Plotting and Drilling Machines: A Case Study". *Zeitschrift für Operations Research (ZOR). now* Mathematical Methods of Operations Research (MMOR) 35(1):61–84, Jan. 1991. Heidelberg, Baden-Württemberg, Germany: Physica-Verlag; now London, England, UK: Springer Nature Limited. ISSN: 0340-9422. doi:10.1007/BF01415960. This paper tackles circuit board plotting and drilling problems by modelling (parts of) them as STSPs. (Cit. on p. 379).

[243] Joel Grus. *Data Science from Scratch: First Principles with Python*. 2nd ed. Sebastopol, CA, USA: O'Reilly Media, Inc., May 2019. ISBN: 978-1-4920-4113-9 (cit. on pp. 3, 5, 367).

[244] Sławomir Gryś. *Computer Arithmetic in Practice*. Boca Raton, FL, USA: CRC Press, Inc., Sept. 2023. ISBN: 978-1-000-93492-2 (cit. on p. 47).

[245] Gregory Z. Gutin and Abraham P. Punnen, eds. *The Traveling Salesman Problem and its Variations*. Vol. 12 of Combinatorial Optimization (COOP). New York, NY, USA: Springer New York, May 2002. ISSN: 1388-3011. doi:10.1007/b101971 (cit. on p. 379).

[246] Terry Halpin and Tony Morgan. *Information Modeling and Relational Databases*. 3rd ed. Burlington, MA, USA/San Mateo, CA, USA: Morgan Kaufmann Publishers, July 2024. ISBN: 978-0-443-23791-1 (cit. on p. 376).

[247] Mohammad Hammoud. "Binary Search". In: *15-122: Principles of Imperative Computation*. Doha, Qatar: Carnegie-Mellon University Qatar, Spr. 2024. Chap. Lecture 06. URL: https://web2.qatar.cmu.edu/~mhhammou/15122-s24/lectures/06-binsearch/slides (visited on 2024-09-26) (cit. on pp. 158, 159).

[248] Hangzhou, Zhejiang, China (中国浙江省杭州市): DeepSeek-AI et al. "DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning". (abs/2501.12948), Jan. 22, 2025. doi:10.48550/ARXIV.2501.12948. URL: https://arxiv.org/abs/2501.12948 (visited on 2025-04-28). arXiv:2501.12948v1 [cs.CL] 22 Jan 2025 (cit. on pp. 57, 372).

[249] Pierre Hansen, Nenad Mladenović, and José Andrés Moreno Pérez. "Variable Neighbourhood Search: Methods and Applications. Invited Survey". *4OR* 6(4):319–360, Dec. 2008. ISSN: 1619-4500. doi:10.1007/s10288-008-0089-1. URL: https://www.researchgate.net/publication/225155945 (visited on 2026-10-02) (cit. on p. 381).

[250] David Harel. "On Folk Theorems". *Communications of the ACM (CACM)* 23(7):379–389, July 1980. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/358886.358892 (cit. on p. 143).

[251] Jan L. Harrington. *Relational Database Design and Implementation*. 4th ed. Burlington, MA, USA/San Mateo, CA, USA: Morgan Kaufmann Publishers, Apr. 2016. ISBN: 978-0-12-849902-3 (cit. on p. 376).

[252] Charles R. Harris, K. Jarrod Millman, Stéfan van der Walt, Ralf Gommers, Pauli "pv" Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. "Array programming with NumPy". *Nature* 585:357–362, 2020. London, England, UK: Springer Nature Limited. ISSN: 0028-0836. doi:10.1038/S41586-020-2649-2 (cit. on pp. 3, 330, 374).

[253] Michael Hausenblas. *Learning Modern Linux*. Sebastopol, CA, USA: O'Reilly Media, Inc., Apr. 2022. ISBN: 978-1-0981-0894-6 (cit. on pp. 9, 365, 371).

[254] Christian Heimes. "`defusedxml` 0.7.1: XML Bomb Protection for Python stdlib Modules". In: Mar. 8, 2021. URL: https://pypi.org/project/defusedxml (visited on 2024-12-15) (cit. on p. 382).

[255] Matthew Helmke. *Ubuntu Linux Unleashed 2021 Edition*. 14th ed. Reading, MA, USA: Addison-Wesley Professional, Aug. 2020. ISBN: 978-0-13-668539-5 (cit. on pp. 9, 371, 380).

[256] Keld Helsgaun. "General k-opt Submoves for the Lin–Kernighan TSP Heuristic". *Mathematical Programming Computation* 1(2-3):119–163, July 2009. London, England, UK: Springer Nature Limited. ISSN: 1867-2949. doi:10.1007/s12532-009-0004-6. URL: https://www2.seas.gwu.edu/~simhaweb/champalg/tsp/papers/Helsgaun2.pdf (visited on 2026-09-26) (cit. on p. 372).

[257] Keld Helsgaun. "An Effective Implementation of the Lin-Kernighan Traveling Salesman Heuristic". 126(1):106–130, Oct. 2000. doi:10.1016/S0377-2217(99)00284-2. URL: http://akira.ruc.dk/~keld/research/LKH/LKH-2.0/DOC/LKH_REPORT.pdf (visited on 2026-09-26) (cit. on p. 372).

[258] André Henning. "Praktische Job-Shop Scheduling-Probleme". PhD thesis. Jena, Thüringen, Germany: Friedrich-Schiller-Universität Jena, Aug. 2002. URL: http://www.db-thueringen.de/servlets/DocumentServlet?id=873 (visited on 2026-09-30). Alternate url: https://nbn-resolving.org/urn:nbn:de:gbv:27-20060809-115700-4. Applies MA-like algorithms to the JSSP. (Cit. on pp. 370, 372).

[259] Raymond Hettinger. *Binary Floating Point Summation Accurate to Full Precision (Python Recipe)*. Vancouver, BC, Canada: ActiveState Software Inc., Mar. 28, 2005. URL: http://code.activestate.com/recipes/393090 (visited on 2024-11-19) (cit. on p. 265).

[260] Raymond Hettinger. `Generator` *Expressions*. Python Enhancement Proposal (PEP) 274. Beaverton, OR, USA: Python Software Foundation (PSF), Jan. 30, 2002. URL: https://peps.python.org/pep-0289 (visited on 2024-11-08) (cit. on p. 231).

[261] Raymond Hettinger. *The* `enumerate()` *Built-In Function*. Python Enhancement Proposal (PEP) 279. Beaverton, OR, USA: Python Software Foundation (PSF), Jan. 30, 2002. URL: https://peps.python.org/pep-0279 (visited on 2025-09-02) (cit. on p. 153).

[262] Raymond Hettinger. "What's New In Python 3.8: f-String Support `=` for Self-Documenting Expressions and Debugging". In: *Python 3 Documentation*. *What's New in Python*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/whatsnew/3.8.html#bpo-36817-whatsnew (visited on 2024-12-01) (cit. on p. 75).

[263] Ian Hickson, Robin Berjon, Steve Faulkner, Travis Leithead, Erika Doyle Navara, Theresa O'Connor, and Silvia Pfeiffer, eds. *HTML5: A Vocabulary and Associated APIs for HTML and XHTML*. W3C Recommendation. Wakefield, MA, USA: World Wide Web Consortium (W3C), Oct. 28, 2014. URL: http://www.w3.org/TR/2014/REC-html5-20141028 (visited on 2024-12-17) (cit. on pp. 316, 369).

[264] D. Richard Hipp et al. "How SQLite Is Tested". In: *SQLite*. Charlotte, NC, USA: Hipp, Wyrick & Company, Inc. (Hwaci), Mar. 13, 2024. URL: https://sqlite.org/testing.html (visited on 2025-01-12) (cit. on p. 177).

[265] D. Richard Hipp et al. "Well-Known Users of SQLite". In: *SQLite*. Charlotte, NC, USA: Hipp, Wyrick & Company, Inc. (Hwaci), Jan. 2, 2023. URL: https://www.sqlite.org/famous.html (visited on 2025-01-12) (cit. on pp. 177, 378).

[266] Manuel Hoffmann, Frank Nagle, and Yanuo Zhou. *The Value of Open Source Software*. Working Paper 24-038. Boston, MA, USA: Harvard Business School, Jan. 1, 2024. URL: https://www.hbs.edu/ris/Publication%20Files/24-038_51f8444f-502c-4139-8bf2-56eb4b65c58a.pdf (visited on 2025-06-04) (cit. on p. 375).

[267] Steve Hollasch. "IEEE Standard 754 Floating Point Numbers". In: *CSE401: Introduction to Compiler Construction*. Seattle, WA, USA: University of Washington, Jan. 8, 1997. URL: https://courses.cs.washington.edu/courses/cse401/01au/details/fp.html (visited on 2024-07-05) (cit. on pp. 37, 44, 45, 259, 366, 368, 377).

[268] Holger H. Hoos and Thomas Stützle. *Stochastic Local Search: Foundations & Applications*. The Morgan Kaufmann Series in Artificial Intelligence. Amsterdam, The Netherlands: Elsevier B.V., 2005. ISBN: 978-1-55860-872-6. doi:10.1016/B978-1-55860-872-6.X5016-1 (cit. on pp. 372, 377).

[269] Holger H. Hoos and Thomas Stützle. *SATLIB – The Satisfiability Library*. 1.4.4. Vancouver, BC, Canada: The University of British Columbia (UBC), May 11, 2001. URL: https://www.cs.ubc.ca/~hoos/SATLIB/index-ubc.html (visited on 2026-09-20). See paper [270] (cit. on pp. 377, 408).

[270] Holger H. Hoos and Thomas Stützle. "SATLIB: An Online Resource for Research on SAT". In: *SAT2000 – Highlights of Satisfiability Research in the Year 2000*. Ed. by Ian Philip Gent, Toby Walsh, and Hans van Maaren. Vol. 63 of Frontiers in Artificial Intelligence and Applications. Amsterdam, The Netherlands: IOS Press BV, Jan. 2000, pp. 283–292. ISSN: 0922-6389. ISBN: 978-1-58603-061-2. See the website [269] (cit. on pp. 377, 408).

[271] "Why does Python 3 round half to even?" In: *Stack Overflow*. Ed. by wjandrea. New York, NY, USA: Stack Exchange Inc., May 31, 2012–June 13, 2025. URL: https://stackoverflow.com/questions/10825926 (visited on 2025-07-18). Closely related to banker's rounding (cit. on pp. 40, 365).

[272] "How to implement division with round-towards-infinity in Python". In: ed. by Kbob. Aug. 24–26, 2011. URL: https://stackoverflow.com/questions/7181757 (visited on 2025-07-28) (cit. on pp. 55, 56).

[273] Trey Hunner. "Every Dunder Method in Python". In: *Python Morsels*. Reykjavík, Iceland: Python Morsels, Mar. 19, 2024. URL: https://www.pythonmorsels.com/every-dunder-method (visited on 2024-12-18) (cit. on pp. 323, 326).

[274] Trey Hunner. "Everything is an Object". In: *Python Morsels*. Reykjavík, Iceland: Python Morsels, Feb. 25, 2021. URL: https://www.pythonmorsels.com/everything-is-an-object (visited on 2025-09-08) (cit. on p. 181).

[275] Trey Hunner. "Mutable Default Arguments". In: *Python Morsels*. Reykjavík, Iceland: Python Morsels, Apr. 7, 2025. URL: https://www.pythonmorsels.com/mutable-default-arguments (visited on 2025-09-06) (cit. on pp. 179, 357).

[276] Trey Hunner. "Python Big $\mathcal{O}$: The Time Complexities of Different Data Structures in Python; Python 3.8-3.12". In: *Python Morsels*. Reykjavík, Iceland: Python Morsels, Apr. 16, 2024. URL: https://www.pythonmorsels.com/time-complexities (visited on 2024-08-27) (cit. on pp. 124, 285).

[277] Trey Hunner. *Python Morsels*. Reykjavík, Iceland: Python Morsels, 2025. URL: https://www.pythonmorsels.com (visited on 2025-04-17) (cit. on p. 6).

[278] John Hunt. *A Beginners Guide to Python 3 Programming*. 2nd ed. Undergraduate Topics in Computer Science (UTICS). Cham, Switzerland: Springer, 2023. ISSN: 1863-7310. ISBN: 978-3-031-35121-1. doi:10.1007/978-3-031-35122-8 (cit. on pp. 4, 375).

[279] John D. Hunter. "Matplotlib: A 2D Graphics Environment". *Computing in Science & Engineering* 9(3):90–95, May–June 2007. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE). ISSN: 1521-9615. doi:10.1109/MCSE.2007.55 (cit. on pp. 3, 372).

[280] John D. Hunter, Darren Dale, Eric Firing, Michael Droettboom, and The Matplotlib Development Team. "Coding Guidelines". In: *Matplotlib: Visualization with Python*. Austin, TX, USA: NumFOCUS, Inc., 2012–2025. URL: https://matplotlib.org/devdocs/devel/coding_guide.html (visited on 2025-02-02) (cit. on p. 108).

[281] John D. Hunter, Darren Dale, Eric Firing, Michael Droettboom, and The Matplotlib Development Team. *Matplotlib: Visualization with Python*. Austin, TX, USA: NumFOCUS, Inc., 2012–2025. URL: https://matplotlib.org (visited on 2025-02-02) (cit. on pp. 3, 372).

[282] *IEEE Standard for Floating-Point Arithmetic*. IEEE Std 754™-2019 (Revision of IEEE Std 754-2008). New York, NY, USA: Institute of Electrical and Electronics Engineers (IEEE), June 13, 2019 (cit. on pp. 37, 38, 40, 44, 45, 259, 281, 365, 366, 368, 377).

[283] *IEEE Standard for Information Technology--Portable Operating System Interfaces (POSIX(TM))--Part 2: Shell and Utilities*. IEEE Std 1003.2-1992. New York, NY, USA: Institute of Electrical and Electronics Engineers (IEEE), June 23, 1993. URL: https://mirror.math.princeton.edu/pub/oldlinux/Linux.old/Ref-docs/POSIX/all.pdf (visited on 2025-03-27). Board Approved: 1992-09-17, ANSI Approved: 1993-04-05. See unapproved draft IEEE P1003.2 Draft 11.2 of 9 1991 at the url (cit. on pp. 211, 376).

[284] *Information Technology – Database Languages – SQL – Part 1: Framework (SQL/Framework), Part 1*. International Standard ISO/IEC 9075-1:2023(E), Sixth Edition, (ANSI X3.135). Geneva, Switzerland: International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC), June 2023. URL: https://standards.iso.org/ittf/PubliclyAvailableStandards/ISO_IEC_9075-1_2023_ed_6_-_id_76583_Publication_PDF_(en).zip (visited on 2025-01-08). Consists of several parts, see https://modern-sql.com/standard for information where to obtain them. (Cit. on p. 377).

[285] *Information Technology – Universal Coded Character Set (UCS)*. International Standard ISO/IEC 10646:2020. Geneva, Switzerland: International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC), Dec. 2020 (cit. on pp. 80, 206, 380, 436).

[286] *Python 3 Documentation*. *Installing Python Modules*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/installing (visited on 2024-08-17) (cit. on pp. 168, 329, 375).

[287] Manuel Iori, Vinícius Loti de Lima, Silvano Martello, Flávio K. Miyazaw, and Michele Monaci. "Exact Solution Techniques for Two-Dimensional Cutting and Packing". *European Journal of Operational Research* 289(2):399–415, Mar. 2021. Amsterdam, The Netherlands: Elsevier B.V. ISSN: 0377-2217. doi:10.1016/j.ejor.2020.06.050. See also arXiv:2004.12619v1 [math.OC] 27 Apr 2020 at doi:10.1016/j.ejor.2020.06.050. (Cit. on p. 364).

[288] "`Iterable`". In: *Python 3 Documentation*. *Glossary*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/glossary.html#term-iterable (visited on 2025-09-16) (cit. on p. 217).

[289] "`Iterator`". In: *Python 3 Documentation*. *Glossary*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/glossary.html#term-iterator (visited on 2025-09-16) (cit. on pp. 216, 217).

[290] "`itertools` – Functions Creating `Iterators` for Efficient Looping". In: *Python 3 Documentation*. *The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/library/itertools.html (visited on 2024-11-09) (cit. on pp. 244, 247).

[291] Christoph Jabs, Dieter Vandesande, Jeremias Berg, Johannes Fichte, George Katsirelos, Ruben Martins, André Schidler, and Dominik Schreiber. *MaxSAT Evaluation: Complete Benchmark Collection*. 2026th ed. Geneva, Switzerland: zenodo (hosted by European Organization for Nuclear Research (CERN)), July 9, 2026. doi:10.5281/zenodo.21283148. All versions of this dataset can be reached directly via doi:10.5281/zenodo.21283147. See also https://github.com/maxsat-evaluations/data and https://maxsat-evaluations.github.io. (Cit. on p. 373).

[292] Dean Jacobs, Jan Prins, Peter Siegel, and Kenneth Wilson. "Monte Carlo Techniques in Code Optimization". In: *15th Annual Workshop on Microprogramming (MICRO 15)*. Oct. 5–7, 1982. Ed. by Joseph Allen Fisher, William J. Tracz, and Bill Hopkins. Palo Alto, CA, USA: Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE) and New York, NY, USA: Association for Computing Machinery (ACM), Oct. 1982, pp. 143–148. doi:10.5555/800036.800944. See [293] (cit. on p. 410).

[293] Dean Jacobs, Jan Prins, Peter Siegel, and Kenneth Wilson. "Monte Carlo Techniques in Code Optimization". *ACM SIGMICRO Newsletter* 13(4):143–148, Dec. 1982. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 1050-916X. doi:10.1145/1014194.800944. See [292] (cit. on pp. 377, 410).

[294] Stefan Jakobs. "On Genetic Algorithms for the Packing of Polygons". *European Journal of Operational Research* 88(1):165–181, Jan. 1996. Amsterdam, The Netherlands: Elsevier B.V. ISSN: 0377-2217. doi:10.1016/0377-2217(94)00166-9 (cit. on p. 369).

[295] Mike James. *Programmer's Python: Everything is an Object – Something Completely Different*. 2nd ed. I/O Press, June 25, 2022 (cit. on pp. 4, 278).

[296] Paul Jansen. *TIOBE Index for January 2025 – January Headline: Python is TIOBE's programming language of the year 2024!* Eindhoven, The Netherlands: TIOBE Software BV, Jan. 2025. URL: https://www.tiobe.com/tiobe-index (visited on 2025-01-07) (cit. on pp. 3, 107).

[297] *Java® Platform, Standard Edition & Java Development Kit Version 22 API Specification*. Redwood Shores, CA, USA: Oracle Corporation, Apr. 9, 2024. URL: https://docs.oracle.com/en/java/javase/22/docs/api (visited on 2024-07-07) (cit. on pp. 40, 365).

[298] Wassim Jaziri, ed. *Tabu Search*. London, England, UK: IntechOpen Limited, Aug. 1, 2008. ISBN: 978-3-902613-34-9. doi:10.5772/65. URL: https://www.intechopen.com/books/6114 (visited on 2026-09-30) (cit. on p. 379).

[299] Robert Johansson. *Numerical Python: Scientific Computing and Data Science Applications with NumPy, SciPy and Matplotlib*. New York, NY, USA: Apress Media, LLC, Dec. 2018. ISBN: 978-1-4842-4246-9 (cit. on pp. 3, 5, 330, 372, 374, 377).

[300] David Stifler Johnson, Gregory Z. Gutin, Lyle Andrew McGeoch, Anders Yeo, Weixiong Zhang, and Alexei Zverovitch. *Experimental Analysis of Heuristics for the ATSP*. Ed. by Gregory Z. Gutin and Abraham P. Punnen. New York, NY, USA, May 2002. doi:10.1007/0-306-48213-4_10. URL: https://archive.dimacs.rutgers.edu/Challenges/TSP/papers/atspchap.pdf (visited on 2026-09-25) (cit. on p. 379).

[301] David Stifler Johnson and Lyle Andrew McGeoch. *Experimental Analysis of Heuristics for the STSP*. Ed. by Gregory Z. Gutin and Abraham P. Punnen. New York, NY, USA, May 2002. doi:10.1007/0-306-48213-4_9. URL: https://archive.dimacs.rutgers.edu/Challenges/TSP/papers/stspchap.pdf (visited on 2026-09-25) (cit. on p. 379).

[302] David Stifler Johnson, Lyle Andrew McGeoch, Fred Glover, and Cesar Rego. *8th DIMACS Implementation Challenge: The Traveling Salesman Problem*. June 1, 2000–Nov. 19, 2013. URL: https://archive.dimacs.rutgers.edu/Challenges/TSP (visited on 2026-09-25) (cit. on p. 379).

[303] Selmer Martin Johnson. "Optimal Two- and Three-Stage Production Schedules with Setup Times Included". *Naval Research Logistics Quarterly* I(1):61–68, Mar. 1954. Chichester, West Sussex, England, UK: John Wiley and Sons Ltd. ISSN: 0894-069X. doi:10.1002/nav.3800010110. URL: https://www.rand.org/content/dam/rand/pubs/papers/2008/P402.pdf (visited on 2023-12-06) (cit. on p. 368).

[304] Stephen Curtis Johnson. *Lint, a C Program Checker*. Computing Science Technical Report 78-1273. New York, NY, USA: Bell Telephone Laboratories, Incorporated, Oct. 25, 1978. URL: https://wolfram.schneider.org/bsd/7thEdManVol2/lint/lint.pdf (visited on 2024-08-23) (cit. on pp. 117, 371).

[305] Steven G. Johnson. "Numerical Integration and the Redemption of the Trapezoidal Rule". In: *18.335 Introduction to Numerical Methods*. Cambridge, MA, USA: Massachusetts Institute of Technology (MIT), Spr. 2011–May 8, 2020. URL: https://math.mit.edu/%7Estevenj/trap-iap-2011.pdf (visited on 2025-09-07). Past lectures are available at https://github.com/mitmath/18335 (visited on 2025-09-07) (cit. on p. 181).

[306] Arthur Jones, Kenneth R. Pearson, and Sidney A. Morris. "Transcendence of e and π". In: *Abstract Algebra and Famous Impossibilities*. Universitext (UTX). New York, NY, USA: Springer New York, 1991. Chap. 9, pp. 115–161. ISSN: 0172-5939. ISBN: 978-1-4419-8552-1. doi:10.1007/978-1-4419-8552-1_8 (cit. on pp. 363, 367).

[307] "`exit` – Terminate a Process". In: *POSIX.1-2024: The Open Group Base Specifications Issue 8, IEEE Std 1003.1™-2024 Edition*. Ed. by Andrew Josey. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE) and San Francisco, CA, USA: The Open Group, Aug. 8, 2024. URL: https://pubs.opengroup.org/onlinepubs/9799919799/functions/exit.html (visited on 2024-10-30) (cit. on p. 367).

[308] Andrew Josey, ed. *POSIX.1-2024: The Open Group Base Specifications Issue 8, IEEE Std 1003.1™-2024 Edition*. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE) and San Francisco, CA, USA: The Open Group, Aug. 8, 2024. URL: https://pubs.opengroup.org/onlinepubs/9799919799 (visited on 2024-10-30).

[309] "stderr, stdin, stdout – Standard I/O Streams". In: *POSIX.1-2024: The Open Group Base Specifications Issue 8, IEEE Std 1003.1™-2024 Edition*. Ed. by Andrew Josey. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE) and San Francisco, CA, USA: The Open Group, Aug. 8, 2024. URL: https://pubs.opengroup.org/onlinepubs/9799919799/functions/stdin.html (visited on 2024-10-30) (cit. on p. 378).

[310] William Kahan. "Pracniques: Further Remarks on Reducing Truncation Errors". *Communications of the ACM (CACM)* 8(1):40, Jan. 1965. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/363707.363723. URL: https://www.convexoptimization.com/TOOLS/Kahan.pdf (visited on 2024-11-18) (cit. on pp. 259–263, 265, 270, 412).

[311] Shen Kangshen, John Newsome Crossley, and Anthony W.-C. Lun. *The Nine Chapters on the Mathematical Art: Companion and Commentary*. Oxford, Oxfordshire, England, UK: Oxford University Press, Oct. 7, 1999. ISBN: 978-0-19-853936-0. doi:10.1093/oso/9780198539360.001.0001. Translation of the historical Chinese mathematics book *Jiu Zhang Suanshu* (九章算术), including the commentaries of LIU Hui (刘徽). (Cit. on pp. 88, 166, 363).

[312] Mikhail G. Katz and David Sherry. "Leibniz's Infinitesimals: Their Fictionality, Their Modern Implementations, and Their Foes from Berkeley to Russell and Beyond". *Erkenntnis: An International Journal of Scientific Philosophy* 78(3):571–625, June 2013. London, England, UK: Springer Nature Limited. ISSN: 0165-0106. doi:10.1007/s10670-012-9370-y (cit. on pp. 181, 370).

[313] Hideto Kazawa, Hisami Suzuki, and Taku Kudo, eds. *Workshop on Advances in Text Input Methods (WTIM@IJCNLP), part of the International Joint Conference on Natural Language Processing (IJCNLP'2011)*. Nov. 13, 2011, Chiang Mai, Thailand. https://www.afnlp.org: Asian Federation of Natural Language Processing (AFNLP), 2011. URL: https://aclanthology.org/W11-35.pdf (visited on 2025-05-08) (cit. on p. 370).

[314] Faizan Khan, Boqi Chen, Dániel Varró, and Shane McIntosh. "An Empirical Study of Type-Related Defects in Python Projects". *IEEE Transactions on Software Engineering* 48(8):3145–3158, Aug. 1, 2022. Los Alamitos, CA, USA: IEEE Computer Society. ISSN: 0098-5589. doi:10.1109/TSE.2021.3082068. URL: https://www.researchgate.net/publication/351729684 (visited on 2024-08-16) (cit. on p. 102).

[315] Scott Kirkpatrick, C. Daniel Gelatt, Jr., and Mario P. Vecchi. "Optimization by Simulated Annealing". *Science Magazine* 220(4598):671–680, May 13, 1983. Washington, D.C., USA: American Association for the Advancement of Science (AAAS). ISSN: 0036-8075. doi:10.1126/science.220.4598.671. URL: https://www.researchgate.net/publication/6026283 (visited on 2025-08-08) (cit. on p. 377).

[316] Andreas Klein. "A Generalized Kahan-Babuška-Summation-Algorithm". *Computing* 76(3-4):279–293, Jan. 2006. Berlin/Heidelberg, Germany: Springer-Verlag GmbH Germany. ISSN: 0010-485X. doi:10.1007/s00607-005-0139-x. Based on [22, 310, 409] (cit. on p. 260).

[317] Bernd Klein. *Einführung in Python 3 – Für Ein- und Umsteiger*. 3., überarbeitete. München, Bayern, Germany: Carl Hanser Verlag GmbH & Co. KG, 2018. ISBN: 978-3-446-45208-4. doi:10.3139/9783446453876 (cit. on p. 5).

[318] Robert Klein. *Scheduling of Resource-Constrained Projects*. Vol. 10 of Operations Research/Computer Science Interfaces Series (ORCS). New York, NY, USA: Springer New York, 2000. ISSN: 1387-666X. ISBN: 978-0-7923-8637-7. doi:10.1007/978-1-4615-4629-0 (cit. on p. 368).

[319] Donald Ervin Knuth. "Big Omicron and Big Omega and Big Theta". *ACM SIGACT News* 8(2):18–24, Apr.–June 1976. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0163-5700. doi:10.1145/1008328.1008329 (cit. on p. 374).

[320] Donald Ervin Knuth. *Fundamental Algorithms*. 3rd ed. Vol. 1 of The Art of Computer Programming. Reading, MA, USA: Addison-Wesley Professional, 1997. ISBN: 978-0-201-89683-1 (cit. on pp. 374, 378).

[321] Donald Ervin Knuth. *Seminumerical Algorithms*. Vol. 2 of The Art of Computer Programming. Reading, MA, USA: Addison-Wesley Professional, Nov. 1997. ISBN: 978-0-201-89684-8 (cit. on pp. 381, 434).

[322] Donald Ervin Knuth. *Sorting and Searching*. Vol. 3 of The Art of Computer Programming. Reading, MA, USA: Addison-Wesley Professional, 1998. ISBN: 978-0-201-89685-5 (cit. on pp. 124, 128, 158, 159, 285, 373).

[323] Katie Kodes. *Intro to XML, JSON, & YAML*. London, England, UK: Payhip, 2019–Sept. 4, 2020 (cit. on pp. 316, 382).

[324] Tobias Kohn and Guido van Rossum. *Structural Pattern Matching: Motivation and Rationale*. Python Enhancement Proposal (PEP) 635. Beaverton, OR, USA: Python Software Foundation (PSF), Sept. 12, 2020–Feb. 8, 2021. URL: https://peps.python.org/pep-0635 (visited on 2024-09-23) (cit. on pp. 145, 357).

[325] Tjalling Charles Koopmans and Martin Beckmann. "Assignment Problems and the Location of Economic Activities". *Econometrica* 25(1):53–76, 1957. New Haven, CT, USA: The Econometric Society and Chichester, West Sussex, England, UK: John Wiley and Sons Ltd. ISSN: 0012-9682. doi:10.2307/1907742 (cit. on p. 376).

[326] Olga Kosheleva. "Babylonian Method of Computing the Square Root: Justifications based on Fuzzy Techniques and on Computational Complexity". In: *Annual Meeting of the North American Fuzzy Information Processing Society (NAFIPS'2009)*. June 14–19, 2009, Cincinnati, OH, USA. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE), 2009. ISBN: 978-1-4244-4575-2. doi:10.1109/NAFIPS.2009.5156463. URL: https://www.cs.utep.edu/vladik/2009/olg09-05a.pdf (visited on 2024-09-25) (cit. on pp. 155, 156).

[327] Jakob Krarup and Peter Mark Pruzan. "Computer-aided Layout Design". In: *Mathematical Programming in Use*. Ed. by Michel Louis Balinski and Claude Lemaréchal. Vol. 9 of. Berlin/Heidelberg, Germany: Springer-Verlag GmbH Germany, 1978. Chap. 6, pp. 75–94. ISSN: 0303-3929. ISBN: 978-3-642-00796-5. doi:10.1007/BFb0120827. Uses QAPs as model for designing architectural layouts. (Cit. on p. 376).

[328] Holger Krekel and pytest-Dev Team. "How to Run Doctests". In: *pytest Documentation*. Release 8.4. Freiburg, Baden-Württemberg, Germany: merlinux GmbH. Chap. 2.8, pp. 65–69. URL: https://docs.pytest.org/en/stable/how-to/doctest.html (visited on 2024-11-07) (cit. on pp. 227, 362, 367).

[329] Holger Krekel and pytest-Dev Team. *pytest Documentation*. Release 8.4. Freiburg, Baden-Württemberg, Germany: merlinux GmbH. URL: https://readthedocs.org/projects/pytest/downloads/pdf/latest (visited on 2024-11-07) (cit. on pp. 170, 171, 361, 375).

[330] Mark W. Krentel. "The Complexity of Optimization Problems". *Journal of Computer and System Sciences* 36(3):490–509, June 1988. Amsterdam, The Netherlands: Elsevier B.V. ISSN: 0022-0000. Explores the hardness (optimization) problems such TSP, Clique, and Coloring, where computing the optimal value is $\mathcal{NP}$-complete. (Cit. on p. 379).

[331] Andrew M. Kuchling. *Python 3 Documentation*. *Regular Expression HOWTO*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/howto/regex.html (visited on 2024-11-01) (cit. on pp. 211, 376).

[332] Animesh Kumar, Sandip Dutta, and Prashant Pranav. "Analysis of SQL Injection Attacks in the Cloud and in WEB Applications". *Security and Privacy* 7(3), May–June 2024. Chichester, West Sussex, England, UK: John Wiley and Sons Ltd. ISSN: 2475-6725. doi:10.1002/SPY2.370 (cit. on p. 377).

[333] Jay LaCroix. *Mastering Ubuntu Server*. 4th ed. Birmingham, England, UK: Packt Publishing Ltd, Sept. 2022. ISBN: 978-1-80323-424-3 (cit. on p. 377).

[334] Vincent Lafage. *Revisiting "What Every Computer Scientist Should Know About Floating-Point Arithmetic"*. arXiv.org: Computing Research Repository (CoRR) abs/2012.02492. Ithaca, NY, USA: Cornell Universiy Library, Dec. 4, 2020. doi:10.48550/arXiv.2012.02492. URL: https://arxiv.org/abs/2012.02492 (visited on 2025-09-03). arXiv:2012.02492v1 [math.NA] 4 Dec 2020, see also [227] (cit. on pp. 155, 157, 259, 357).

[335] "Lambda". In: *Python 3 Documentation*. *The Python Language Reference*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. Chap. 6.14. URL: https://docs.python.org/3/reference/expressions.html#lambda (visited on 2024-10-09) (cit. on p. 183).

[336] Joan Lambert and Curtis Frye. *Microsoft Office Step by Step (Office 2021 and Microsoft 365)*. Hoboken, NJ, USA: Microsoft Press, Pearson Education, Inc., June 2022. ISBN: 978-0-13-754493-6 (cit. on p. 373).

[337] Charles Landau. *TensorFlow Deep Dive: Build, Train, and Deploy Machine Learning Models with TensorFlow*. Sebastopol, CA, USA: O'Reilly Media, Inc., Dec. 2023 (cit. on pp. 3, 378).

[338] Edmund Landau. *Handbuch der Lehre von der Verteilung der Primzahlen*. Leipzig, Sachsen, Germany: B. G. Teubner, 1909. ISBN: 978-0-8218-2650-8 (cit. on p. 374).

[339] Łukasz Langa. *Literature Overview for Type Hints*. Python Enhancement Proposal (PEP) 482. Beaverton, OR, USA: Python Software Foundation (PSF), Jan. 8, 2015. URL: https://peps.python.org/pep-0482 (visited on 2024-10-09) (cit. on p. 380).

[340] Łukasz Langa. *Type Hinting Generics In Standard Collections*. Python Enhancement Proposal (PEP) 585. Beaverton, OR, USA: Python Software Foundation (PSF), Mar. 3, 2019. URL: https://peps.python.org/pep-0585 (visited on 2024-10-09) (cit. on pp. 113, 121, 124, 128).

[341] Eugene Leighton Lawler, Jan Karel Lenstra, and Alexander Hendrik George Rinnooy Kan. "Recent Developments in Deterministic Sequencing and Scheduling: A Survey". In: *Deterministic and Stochastic Scheduling: Proceedings of the NATO Advanced Study and Research Institute on Theoretical Approaches to Scheduling Problems*. June 6–17, 1981, Durham, England, UK. Ed. by Michael Anthony Howe Dempster, Jan Karel Lenstra, and Alexander Hendrik George Rinnooy Kan. Vol. 84 of Nato Science Series C: Mathematical and Physical Science (ASIC). London, England, UK: Springer Nature Limited, Apr. 1982, pp. 35–73. ISBN: 978-90-277-1397-1. doi:10.1007/978-94-009-7801-0_3. The JSSP is $\mathcal{NP}$-hard. (Cit. on p. 370).

[342] Eugene Leighton Lawler, Jan Karel Lenstra, Alexander Hendrik George Rinnooy Kan, and David B. Shmoys. "Sequencing and Scheduling: Algorithms and Complexity". In: *Production Planning and Inventory*. Ed. by Stephen C. Graves, Alexander Hendrik George Rinnooy Kan, and Paul H. Zipkin. Vol. IV of Handbooks of Operations Research and Management Science. Amsterdam, The Netherlands: Elsevier B.V., 1993. Chap. 9, pp. 445–522. ISSN: 0927-0507. ISBN: 978-0-444-87472-6. doi:10.1016/S0927-0507(05)80189-6. URL: http://alexandria.tue.nl/repository/books/339776.pdf (visited on 2023-12-06). Discusses, amongst others, that the JSSP is $\mathcal{NP}$-hard. (Cit. on pp. 370, 374).

[343] Eugene Leighton Lawler, Jan Karel Lenstra, Alexander Hendrik George Rinnooy Kan, and David B. Shmoys. *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. Estimation, Simulation, and Control – Wiley-Interscience Series in Discrete Mathematics and Optimization. Chichester, West Sussex, England, UK: Wiley Interscience, Sept. 1985. ISSN: 0277-2698. ISBN: 978-0-471-90413-7 (cit. on p. 379).

[344] Stephen R. Lawrence. "Resource Constrained Project Scheduling: An Experimental Investigation of Heuristic Scheduling Techniques". PhD thesis. Pittsburgh, PA, USA: Graduate School of Industrial Administration (GSIA), Carnegie-Mellon University, 1984 (cit. on p. 370).

[345] Kent D. Lee and Steve Hubbard. *Data Structures and Algorithms with Python*. Undergraduate Topics in Computer Science (UTICS). Cham, Switzerland: Springer, 2015. ISSN: 1863-7310. ISBN: 978-3-319-13071-2. doi:10.1007/978-3-319-13072-9 (cit. on pp. 5, 375).

[346] Michael Lee, Ivan Levkivskyi, and Jukka Lehtosalo. *Literal Types*. Python Enhancement Proposal (PEP) 586. Beaverton, OR, USA: Python Software Foundation (PSF), Mar. 14, 2019. URL: https://peps.python.org/pep-0586 (visited on 2024-12-17) (cit. on pp. 318, 359, 371).

[347] Jukka Lehtosalo, Ivan Levkivskyi, Jared Hance, Ethan Smith, Guido van Rossum, Jelle "JelleZijlstra" Zijlstra, Michael J. Sullivan, Shantanu Jain, Xuanda Yang, Jingchen Ye, Nikita Sobolev, and Mypy Contributors. *Mypy – Static Typing for Python*. San Francisco, CA, USA: GitHub Inc, 2024. URL: https://github.com/python/mypy (visited on 2024-08-17) (cit. on pp. 103, 104, 361, 373).

[348] Jukka Lehtosalo and Mypy Contributors. *Welcome to Mypy Documentation! (Mypy 1.13.0 documentation)*. Portland, OR, USA: Read the Docs, Inc., Oct. 22, 2024. URL: https://mypy.readthedocs.io (visited on 2024-12-12) (cit. on pp. 254, 358).

[349] Marc-André Lemburg. *Python Database API Specification v2.0*. Python Enhancement Proposal (PEP) 249. Beaverton, OR, USA: Python Software Foundation (PSF), Apr. 12, 1999. URL: https://peps.python.org/pep-0249 (visited on 2025-02-02) (cit. on p. 375).

[350] Moritz Lenz. *Python Continuous Integration and Delivery: A Concise Guide with Examples*. New York, NY, USA: Apress Media, LLC, Dec. 2018. ISBN: 978-1-4842-4281-0 (cit. on pp. 5, 366).

[351] Reuven M. Lerner. *Pandas Workout*. Shelter Island, NY, USA: Manning Publications, June 2024. ISBN: 978-1-61729-972-8 (cit. on pp. 3, 375).

[352] Xueping Li and Kaike Zhang. "Single Batch Processing Machine Scheduling with Two-Dimensional Bin Packing Constraints". *International Journal of Production Economics (IJPE)* 196:113–121, Feb. 2018. Amsterdam, The Netherlands: Elsevier B.V. ISSN: 0925-5273. doi:10.1016/j.ijpe.2017.11.015. Mentions some scheduling tasks of Additive Manufacturing jobs can be viewed as instance of the 2BP|R|F. (Cit. on p. 364).

[353] Tianyu Liang (梁天宇), Zhize Wu (吴志泽), Jörg Lässig, Daan van den Berg, Sarah Louise Thomson, and Thomas Weise (汤卫思). "Addressing the Traveling Salesperson Problem with Frequency Fitness Assignment and Hybrid Algorithms". *Soft Computing* 28(17-18):9495–9508, July 2024. London, England, UK: Springer Nature Limited. ISSN: 1432-7643. doi:10.1007/S00500-024-09718-8 (cit. on pp. 368, 373, 379).

[354] Tianyu Liang (梁天宇), Zhize Wu (吴志泽), Jörg Lässig, Daan van den Berg, and Thomas Weise (汤卫思). "Solving the Traveling Salesperson Problem using Frequency Fitness Assignment". In: *IEEE Symposium Series on Computational Intelligence (SSCI'2022)*. Dec. 4–7, 2022, Singapore. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE), 2022. ISBN: 978-1-6654-8769-6. doi:10.1109/SSCI51031.2022.10022296 (cit. on pp. 368, 373).

[355] Tianyu Liang (梁天宇), Zhize Wu (吴志泽), Matthias Thürer, Markus Wagner, and Thomas Weise (汤卫思). "Generating Small Instances with Interesting Features for the Traveling Salesperson Problem". In: *16th International Joint Conference on Computational Intelligence (IJCCI'24)*. Nov. 20–22, 2024, Porto, Portugal. Ed. by Francesco Marcelloni, Kurosh Madani, Niki van Stein, and Joaquim Filipe. Porto, Portugal: SciTePress: Science and Technology Publications, Lda, 2024, pp. 173–180. ISSN: 2184-3236. ISBN: 978-989-758-721-4. doi:10.5220/0012888800003837 (cit. on p. 373).

[356] *LibreOffice – The Document Foundation*. Berlin, Germany: The Document Foundation, 2024. URL: https://www.libreoffice.org (visited on 2024-12-12) (cit. on pp. 249, 316, 371, 373).

[357] Luan P. Lima, Lincoln S. Rocha, Carla I. M. Bezerra, and Matheus Paixão. "Assessing Exception Handling Testing Practices in Open-Source Libraries". *Empirical Software Engineering: An International Journal* 26(5:85), June–Sept. 2021. London, England, UK: Springer Nature Limited. ISSN: 1382-3256. doi:10.1007/s10664-021-09983-3. URL: https://arxiv.org/abs/2105.00500 (visited on 2024-10-29). See also arXiv:2105.00500v1 [cs.SE] 2 May 2021 (cit. on p. 209).

[358] Shen Lin and Brian W. Kernighan. "An Effective Heuristic Algorithm for the Traveling-Salesman Problem". *Operations Research* 21(2):498–516, Mar.–Apr. 1973. Catonsville, MD, USA: The Institute for Operations Research and the Management Sciences (INFORMS). ISSN: 0030-364X. doi:10.1287/opre.21.2.498. URL: https://www.cs.princeton.edu/~bwk/btl.mirror/tsp.pdf (visited on 2026-09-26) (cit. on p. 372).

[359] John D. C. Little, Katta G. Murty, Dura W. Sweeny, and Caroline Karel. *An Algorithm for the Traveling Salesman Problem*. Sloan Working Papers 07-63. Massachusetts Institute of Technology (MIT): Massachusetts Institute of Technology (MIT), Sloan School of Management, Mar. 1963. URL: http://dspace.mit.edu/bitstream/handle/1721.1/46828/algorithmfortrav00litt.pdf (visited on 2026-01-03) (cit. on p. 365).

[360] Dequan Liu (刘德全) and Hongfei Teng (滕弘飞). "An Improved BL-Algorithm for EAGenetic Algorithm of the Orthogonal Packing of Rectangles". *European Journal of Operational Research* 112(2):413–420, Jan. 1999. Amsterdam, The Netherlands: Elsevier B.V. ISSN: 0377-2217. doi:10.1016/S0377-2217(97)00437-2 (cit. on p. 369).

[361] Andrea Lodi, Silvano Martello, and Michele Monaci. "Two-Dimensional Packing Problems: A Survey". *European Journal of Operational Research* 141(2):241–252, Sept. 2002. Amsterdam, The Netherlands: Elsevier B.V. ISSN: 0377-2217. doi:10.1016/S0377-2217(02)00123-6 (cit. on pp. 364, 366).

[362] Andrea Lodi, Silvano Martello, and Daniele Vigo. "Heuristic and Metaheuristic Approaches for a Class of Two-Dimensional Bin Packing Problems". *INFORMS Journal on Computing* 11(4):345–357, Aut. 1999. Catonsville, MD, USA: The Institute for Operations Research and the Management Sciences (INFORMS). ISSN: 1091-9856. doi:10.1287/ijoc.11.4.345 (cit. on p. 364).

[363] Andrea Lodi, Silvano Martello, and Daniele Vigo. "Recent Advances on Two-Dimensional Bin Packing Problems". *Discrete Applied Mathematics* 123(1–3):379–396, Nov. 2002. Amsterdam, The Netherlands: Elsevier B.V. ISSN: 0166-218X. doi:10.1016/S0166-218X(01)00347-X (cit. on pp. 364, 366, 367).

[364] Eliane Maria Loiola, Nair Maria Maia de Abreu, Paulo Oswaldo Boaventura-Netto, Peter M. Hahn, and Tania Querido. "A Survey for the Quadratic Assignment Problem". *European Journal of Operational Research* 176(2):657–690, 2007. Amsterdam, The Netherlands: Elsevier B.V. ISSN: 0377-2217. doi:10.1016/j.ejor.2005.09.032 (cit. on p. 376).

[365] Mariana A. Londe, Luciana S. Pessoa, Carlos E. Andrade, and Mauricio G. C. Resende. *Biased Random-Key Genetic Algorithms: A Review*. arXiv.org: Computing Research Repository (CoRR) abs/2312.00961v2. Ithaca, NY, USA: Cornell Universiy Library, Dec. 6, 2023. doi:10.48550/arXiv.2312.00961. URL: https://arxiv.org/abs/2312.00961v2 (visited on 2024-06-27). arXiv:2312.00961v2 [cs.NE] 06 Dec 2023; See [366] (cit. on p. 416).

[366] Mariana A. Londe, Luciana S. Pessoa, Carlos E. Andrade, and Mauricio G. C. Resende. "Biased Random-Key Genetic Algorithms: A Review". *European Journal of Operational Research* 321(1):1–22, Feb. 2025. Amsterdam, The Netherlands: Elsevier B.V. ISSN: 0377-2217. doi:10.1016/j.ejor.2024.03.030. See [365] (cit. on pp. 366, 415).

[367] Gloria Lotha, Aakanksha Gaur, Erik Gregersen, Swati Chopra, and William L. Hosch. "Client-Server Architecture". In: *Encyclopaedia Britannica*. Ed. by The Editors of Encyclopaedia Britannica. Chicago, IL, USA: Encyclopædia Britannica, Inc., Jan. 3, 2025. URL: https://www.britannica.com/technology/client-server-architecture (visited on 2025-01-20) (cit. on p. 366).

[368] Marc Loy, Patrick Niemeyer, and Daniel Leuck. *Learning Java*. 5th ed. Sebastopol, CA, USA: O'Reilly Media, Inc., Mar. 2020. ISBN: 978-1-4920-5627-0 (cit. on p. 370).

[369] Laurent Luce. *Python Dictionary Implementation*. Belmont, MA, USA, Aug. 29, 2011–May 30, 2020. URL: https://www.laurentluce.com/posts/python-dictionary-implementation (visited on 2024-12-09) (cit. on pp. 285, 286).

[370] Peter Luschny. *A New Kind of Factorial Function*. Highland Park, NJ, USA: The OEIS Foundation Inc., Oct. 4, 2015. URL: https://oeis.org/A000142/a000142.pdf (visited on 2024-09-29) (cit. on pp. 163, 363).

[371] Mark Lutz. *Learning Python*. 6th ed. Sebastopol, CA, USA: O'Reilly Media, Inc., Mar. 2025. ISBN: 978-1-0981-7130-8 (cit. on pp. 4, 375).

[372] Rita Macedo, Cláudio Alves, and J. M. Valério de Carvalho. "Arc-Flow Model for the Two-Dimensional Guillotine Cutting Stock Problem". *Computers & Operations Research* 37(6):991–1001, June 2010. Amsterdam, The Netherlands: Elsevier B.V. ISSN: 0305-0548. doi:10.1016/j.cor.2009.08.005 (cit. on p. 367).

[373] *Making Open Source Work Better for Developers: Highlights of the 2019 Tidelift Managed Open Source Survey*. Boston, MA, USA: Tidelift, Inc., June 2019. URL: https://tidelift.com/subscription/managed-open-source-survey (visited on 2024-12-29) (cit. on p. 2).

[374] Reinhard Männer and Bernard Manderick, eds. *Parallel Problem Solving from Nature 2 (PPSN-II)*. Sept. 28–30, 1992, Brussels, Belgium. Amsterdam, The Netherlands: Elsevier B.V., 1992.

[375] Francesco Marcelloni, Kurosh Madani, Niki van Stein, and Joaquim Filipe, eds. *16th International Joint Conference on Computational Intelligence (IJCCI'24)*. Nov. 20–22, 2024, Porto, Portugal. Porto, Portugal: SciTePress: Science and Technology Publications, Lda, 2024. ISSN: 2184-3236. ISBN: 978-989-758-721-4. doi:10.5220/0000195000003837.

[376] *MariaDB Server Documentation*. Milpitas, CA, USA: MariaDB, 2025. URL: https://mariadb.com/kb/en/documentation (visited on 2025-04-24) (cit. on p. 372).

[377] Charlie Marsh. "Ruff". In: URL: https://pypi.org/project/ruff (visited on 2025-08-29) (cit. on pp. 118, 376).

[378] Charlie Marsh. *ruff: An Extremely Fast Python Linter and Code Formatter, Written in Rust*. New York, NY, USA: Astral Software Inc., Aug. 28, 2022. URL: https://docs.astral.sh/ruff (visited on 2024-08-23) (cit. on pp. 118, 138, 141, 356, 357, 361, 376).

[379] Silvano Martello and Daniele Vigo. "Exact Solution of the Two-Dimensional Finite Bin Packing Problem". 44(3):388–399, Mar. 1998. doi:10.1287/mnsc.44.3.388 (cit. on pp. 364, 365).

[380] "`math` – Mathematical functions". In: *Python 3 Documentation. The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/library/math.html (visited on 2025-04-27) (cit. on pp. 51–54).

[381] "Mathematical Functions and Operators". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 20, 2025. Chap. 9.3. URL: https://www.postgresql.org/docs/17/functions-math.html (visited on 2025-02-27) (cit. on pp. 363, 367).

[382] Eric Matthes. *Python Crash Course*. 3rd ed. San Francisco, CA, USA: No Starch Press, Jan. 2023. ISBN: 978-1-7185-0270-3 (cit. on p. 4).

[383] Clive Maxfield and Alvin Brown. "Rounding Algorithms 101". In: *DIY Calculator*. Huntsville, AL, USA: DIY Calculator, 2005. URL: https://www.clivemaxfield.com/diycalculator/popup-m-round.shtml (visited on 2025-07-18). Also discusses banker's rounding (cit. on pp. 40, 365).

[384] Aaron Maxwell. *What are f-strings in Python and how can I use them?* Oakville, ON, Canada: Infinite Skills Inc, June 2017. ISBN: 978-1-4919-9486-3 (cit. on pp. 5, 73, 368).

[385] Ron McFadyen and Cindy Miller. *Relational Databases and Microsoft Access*. 3rd ed. Palatine, IL, USA: Harper College, 2014–2019. URL: https://harpercollege.pressbooks.pub/relationaldatabases (visited on 2025-04-11) (cit. on p. 373).

[386] Wes McKinney. *Python for Data Analysis*. 3rd ed. Sebastopol, CA, USA: O'Reilly Media, Inc., Aug. 2022. ISBN: 978-1-0981-0403-0 (cit. on pp. 3, 5).

[387] MDN Contributors. *Signature (Functions)*. San Francisco, CA, USA: Mozilla Corporation, June 8, 2023. URL: https://developer.mozilla.org/en-US/docs/Glossary/Signature/Function (visited on 2024-12-12) (cit. on p. 377).

[388] Lucas Mearian. "What are LLMs, and how are they used in generative AI?" *Computerworld*, Feb. 7, 2024. Framingham, MA, USA: CW Communications, Inc. and Needham, MA, USA: Foundry (formerly IDG Communications, Inc.) ISSN: 0010-4841. URL: https://www.computerworld.com/article/1627101 (visited on 2025-04-27) (cit. on p. 372).

[389] Pedro Mejia Alvarez, Raul E. Gonzalez Torres, and Susana Ortega Cisneros. *Exception Handling – Fundamentals and Programming*. SpringerBriefs in Computer Science. Cham, Switzerland: Springer, Feb. 2024. ISSN: 2191-5768. ISBN: 978-3-031-50680-2. doi:10.1007/978-3-031-50681-9 (cit. on pp. 2, 186).

[390] Jim Melton and Alan R. Simon. *SQL: 1999 – Understanding Relational Language Components*. The Morgan Kaufmann Series in Data Management Systems. Burlington, MA, USA/San Mateo, CA, USA: Morgan Kaufmann Publishers, June 2001. ISSN: 1046-1698. ISBN: 978-1-55860-456-8 (cit. on p. 377).

[391] Mark Mendoza. *Parameter Specification Variables*. Python Enhancement Proposal (PEP) 612. Beaverton, OR, USA: Python Software Foundation (PSF), Dec. 18, 2019–July 13, 2020. URL: https://peps.python.org/pep-0612 (visited on 2024-10-09) (cit. on p. 183).

[392] Stephan Mertens. "Exhaustive Search for Low-Autocorrelation Binary Sequences". *Journal of Physics A: Mathematical and General* 29(18):L473, Sept. 1996. Bristol, England, UK: IOP Publishing Ltd. ISSN: 0305-4470. doi:10.1088/0305-4470/29/18/005 (cit. on p. 371).

[393] Peter Merz and Bernd Freisleben. "A Comparison of MAMemetic Algorithms, TSTabu Search, and ACOAnt Colonies for the QAPQuadratic Assignment Problem". In: *Congress on Evolutionary Computation (CEC'1999)*. July 6–9, 1999, Washington, D.C., USA. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE), 1999, pp. 2063–2070. doi:10.1109/CEC.1999.785529 (cit. on p. 376).

[394] Carl Meyer. *Python Virtual Environments*. Python Enhancement Proposal (PEP) 405. Beaverton, OR, USA: Python Software Foundation (PSF), June 13, 2011–May 24, 2012. URL: https://peps.python.org/pep-0405 (visited on 2024-12-25) (cit. on pp. 330, 381).

[395] Zbigniew "Zbyszek" Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Berlin/Heidelberg, Germany: Springer-Verlag GmbH Germany, 1996. ISBN: 978-3-540-58090-4. doi:10.1007/978-3-662-03315-9 (cit. on pp. 368, 377).

[396] *Microsoft Word*. Redmond, WA, USA: Microsoft Corporation, 2024. URL: https://www.microsoft.com/en-us/microsoft-365/word (visited on 2024-12-12) (cit. on pp. 249, 316, 373).

[397] Burkhard Militzer, Michele Zamparelli, and Dieter Beule. "Evolutionary Search for Low Autocorrelated Binary Sequences". *IEEE Transactions on Evolutionary Computation* 2(1):34–39, Apr. 1998. Los Alamitos, CA, USA: IEEE Computer Society. ISSN: 1089-778X. doi:10.1109/4235.728212 (cit. on p. 371).

[398] Alfonsas Misevičius. "A Tabu Search Algorithm for the Quadratic Assignment Problem". *Computational Optimization and Applications* 30:95–111, Jan. 2005. London, England, UK: Springer Nature Limited. ISSN: 0926-6003. doi:10.1007/s10589-005-4562-x (cit. on p. 379).

[399] Alfonsas Misevičius. "A Modified Simulated Annealing Algorithm for the Quadratic Assignment Problem". *Informatica* 14(4):497–514, Jan. 2003. Vilnius, Lithuania: Vilnius University, Institute of Data Science and Digital Technologies (DMSTI), Lithuanian Academy of Sciences. ISSN: 0868-4952. doi:10.15388/Informatica.2003.037 (cit. on p. 377).

[400] Alfonsas Misevičius. *An Implementation of the Iterated Tabu Search Algorithm for the Quadratic Assignment Problem*. Working Paper. Kaunas, Lithuania: Kaunas University of Technology, 2008. See also [401] (cit. on p. 418).

[401] Alfonsas Misevičius. "An Implementation of the Iterated Tabu Search Algorithm for the Quadratic Assignment Problem". *OR Spectrum: Quantitative Approaches in Management. Formerly* Operations-Research-Spektrum 34(3):665–690, July 2012. London, England, UK: Springer Nature Limited. ISSN: 0171-6468. doi:10.1007/s00291-011-0274-z. See also [400] (cit. on pp. 376, 379, 418).

[402] Melanie Mitchell. *An Introduction to Genetic Algorithms*. Complex Adaptive Systems. Cambridge, MA, USA: MIT Press, Feb. 1998. ISBN: 978-0-262-13316-6. URL: http://boente.eti.br/fuzzy/ebook-fuzzy-mitchell.pdf (visited on 2025-08-08) (cit. on pp. 368, 377).

[403] Nenad Mladenović and Pierre Hansen. "Variable Neighborhood Search". *Computers & Operations Research* 24(11):1097–1100, Nov. 1997. Amsterdam, The Netherlands: Elsevier B.V. ISSN: 0305-0548. doi:10.1016/S0305-0548(97)00031-2 (cit. on p. 381).

[404] "More Control Flow Tools". In: *Python 3 Documentation*. *The Python Tutorial*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. Chap. 4. URL: https://docs.python.org/3/tutorial/controlflow.html (visited on 2025-09-03) (cit. on pp. 133, 135, 137, 143, 156, 158, 161).

[405] Wai Ho Mow, Ke-Lin Du, and Wei Hsiang Wu. "New Evolutionary Search for Long Low Autocorrelation Binary Sequences". *IEEE Transactions on Aerospace and Electronic Systems* 51(1):290–303, Jan. 2015. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE). ISSN: 0018-9251. doi:10.1109/TAES.2014.130518 (cit. on p. 371).

[406] Heinz Mühlenbein. "How Genetic Algorithms Really Work: Mutation and Hillclimbing". In: *Parallel Problem Solving from Nature 2 (PPSN-II)*. Sept. 28–30, 1992, Brussels, Belgium. Ed. by Reinhard Männer and Bernard Manderick. Amsterdam, The Netherlands: Elsevier B.V., 1992, pp. 15–26. URL: https://www.researchgate.net/publication/284805798 (visited on 2025-08-15) (cit. on p. 374).

[407] Zsolt Nagy. *Regex Quick Syntax Reference: Understanding and Using Regular Expressions*. New York, NY, USA: Apress Media, LLC, Aug. 2018. ISBN: 978-1-4842-3876-9 (cit. on pp. 211, 376).

[408] Robert Nemiroff and Jerry Bonnell. *The Square Root of Two to 1 Million Digits*. Hanover, MD, USA: Astrophysics Science Division (ASD), National Aeronautics and Space Administration (NASA), Apr. 2, 1997. URL: https://apod.nasa.gov/htmltest/gifcity/sqrt2.1mil (visited on 2024-12-14) (cit. on p. 312).

[409] Arnold Neumaier. "Rundungsfehleranalyse einiger Verfahren zur Summation endlicher Summen". *ZAMM – Journal of Applied Mathematics and Mechanics / Zeitschrift für Angewandte Mathematik und Mechanik* 54(1):39–51, 1974. Weinheim, Baden-Württemberg, Germany: Wiley-VCH GmbH. ISSN: 0044-2267. doi:10.1002/zamm.19740540106. URL: https://arnold-neumaier.at/scan/01.pdf (visited on 2024-11-18) (cit. on pp. 260–262, 264, 270, 412).

[410] Cameron Newham and Bill Rosenblatt. *Learning the Bash Shell – Unix Shell Programming: Covers Bash 3.0*. 3rd ed. Sebastopol, CA, USA: O'Reilly Media, Inc., 2005. ISBN: 978-0-596-00965-6 (cit. on pp. 365, 439).

[411] Thomas Nield. *An Introduction to Regular Expressions*. Sebastopol, CA, USA: O'Reilly Media, Inc., June 2019. ISBN: 978-1-4920-8255-2 (cit. on pp. 211, 376).

[412] Siegfried Nijssen and Thomas Bäck. "An Analysis of the Behavior of Simplified Evolutionary Algorithms on Trap Functions". *IEEE Transactions on Evolutionary Computation* 7(1):11–22, Feb. 2003. Los Alamitos, CA, USA: IEEE Computer Society. ISSN: 1089-778X. doi:10.1109/TEVC.2002.806169 (cit. on p. 379).

[413] nishkarsh146. *Complexity Cheat Sheet for Python Operations*. Noida, Uttar Pradesh, India: GeeksforGeeks – Sanchhaya Education Private Limited, Aug. 17, 2022. URL: https://www.geeksforgeeks.org/complexity-cheat-sheet-for-python-operations (visited on 2024-08-27) (cit. on pp. 124, 285).

[414] Ivan Niven. "The Transcendence of π". *The American Mathematical Monthly* 46(8):469–471, Oct. 1939. London, England, UK: Taylor and Francis Ltd. ISSN: 1930-0972. doi:10.2307/2302515 (cit. on p. 363).

[415] Beatrice Nolan. "An AI-powered coding tool wiped out a software company's database, then apologized for a 'catastrophic failure on my part'". *Fortune*, July 23, 2025. New York, NY, USA: Fortune Media IP Limited. URL: https://fortune.com/2025/07/23/ai-coding-tool-replit-wiped-database-called-it-a-catastrophic-failure (visited on 2025-07-29) (cit. on p. 61).

[416] Eugeniusz Nowicki and Czesław Smutnicki. "A Fast Taboo Search Algorithm for the Job Shop Problem". *Management Science* 42(6):783–938, June 1996. Catonsville, MD, USA: The Institute for Operations Research and the Management Sciences (INFORMS). ISSN: 0025-1909. doi:10.1287/mnsc.42.6.797 (cit. on p. 379).

[417] "Numeric Types – `int`, `float`, `complex`". In: *Python 3 Documentation*. *The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/library/stdtypes.html#typesnumeric (visited on 2024-07-05) (cit. on p. 37).

[418] NumPy Team. *NumPy*. San Francisco, CA, USA: GitHub Inc and Austin, TX, USA: NumFOCUS, Inc. URL: https://numpy.org (visited on 2025-02-02) (cit. on pp. 3, 374).

[419] NumPy Team. "Typing (`numpy.typing`)". In: *NumPy*. San Francisco, CA, USA: GitHub Inc and Austin, TX, USA: NumFOCUS, Inc. URL: https://numpy.org/devdocs/reference/typing.html (visited on 2025-02-02) (cit. on p. 108).

[420] John J. O'Connor and Edmund F. Robertson. *Liu Hui*. St Andrews, Scotland, UK: University of St Andrews, School of Mathematics and Statistics, Dec. 2003. URL: https://mathshistory.st-andrews.ac.uk/Biographies/Liu_Hui (visited on 2024-08-10). Discusses LIU Hui's (刘徽) method of approximating π (cit. on pp. 88, 166, 363).

[421] Regina O. Obe and Leo S. Hsu. *PostgreSQL: Up and Running*. 3rd ed. Sebastopol, CA, USA: O'Reilly Media, Inc., Oct. 2017. ISBN: 978-1-4919-6336-4 (cit. on p. 375).

[422] "`object`". In: *Python 3 Documentation*. *Glossary*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/glossary.html#term-object (visited on 2025-09-25) (cit. on p. 269).

[423] "Objects, Values and Types". In: *Python 3 Documentation*. *The Python Language Reference*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. Chap. 3.1. URL: https://docs.python.org/3/reference/datamodel.html#objects-values-and-types (visited on 2024-12-07) (cit. on p. 278).

[424] A. Jefferson Offutt. "Unit Testing Versus Integration Testing". In: *Test: Faster, Better, Sooner – IEEE International Test Conference (ITC'1991)*. Oct. 26–30, 1991, Nashville, TN, USA. Los Alamitos, CA, USA: IEEE Computer Society, 1991. Chap. Paper P2.3, pp. 1108–1109. ISSN: 1089-3539. ISBN: 978-0-8186-9156-0. doi:10.1109/TEST.1991.519784 (cit. on pp. 169, 380).

[425] Brian Okken. *Python Testing with pytest*. Flower Mound, TX, USA: Pragmatic Bookshelf by The Pragmatic Programmers, L.L.C., Feb. 2022. ISBN: 978-1-68050-860-4 (cit. on pp. 5, 170, 375).

[426] Michael Olan. "Unit Testing: Test Early, Test Often". *Journal of Computing Sciences in Colleges (JCSC)* 19(2):319–328, Dec. 2003. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 1937-4771. doi:10.5555/948785.948830. URL: https://www.researchgate.net/publication/255673967 (visited on 2025-09-05) (cit. on pp. 169, 380).

[427] Robert Orfali, Dan Harkey, and Jeri Edwards. *Client/Server Survival Guide*. 3rd ed. Chichester, West Sussex, England, UK: John Wiley and Sons Ltd., Jan. 25, 1999. ISBN: 978-0-471-31615-2 (cit. on p. 366).

[428] Tom Packebusch and Stephan Mertens. "Low Autocorrelation Binary Sequences". *Journal of Physics A: Mathematical and Theoretical* 49(16):165001, Mar. 2016. Bristol, England, UK: IOP Publishing Ltd. ISSN: 1751-8121. doi:10.1088/1751-8113/49/16/165001 (cit. on p. 371).

[429] Ashwin Pajankar. *Hands-on Matplotlib: Learn Plotting and Visualizations with Python 3*. New York, NY, USA: Apress Media, LLC, Nov. 2021. ISBN: 978-1-4842-7410-1 (cit. on pp. 3, 5, 372).

[430] Ashwin Pajankar. *Python Unit Test Automation: Automate, Organize, and Execute Unit Tests in Python*. New York, NY, USA: Apress Media, LLC, Dec. 2021. ISBN: 978-1-4842-7854-3 (cit. on pp. 5, 169, 170, 375, 380).

[431] Pandas Developers. *Contributing to the Code Base*. Austin, TX, USA: NumFOCUS, Inc. and Montreal, QC, Canada: OVHcloud. URL: https://pandas.pydata.org/docs/development/contributing_codebase.html (visited on 2025-02-02) (cit. on p. 108).

[432] Pandas Developers. *Pandas*. Austin, TX, USA: NumFOCUS, Inc. and Montreal, QC, Canada: OVHcloud. URL: https://pandas.pydata.org (visited on 2025-02-02) (cit. on pp. 3, 375).

[433] Panos Miltiades Pardalos, Ding-Zhu Du, and Ronald Lewis Graham, eds. *Handbook of Combinatorial Optimization*. 1st ed. Boston, MA, USA: Springer, 1998. ISBN: 978-1-4613-7987-4.

[434] Panos Miltiades Pardalos, Oleg V. Shylo, and Alkis Vazacopoulos. "Solving Job Shop Scheduling Problems Utilizing the Properties of Backbone and "Big Valley"". *Computational Optimization and Applications* 47(1):61–76, Sept. 2010. London, England, UK: Springer Nature Limited. ISSN: 0926-6003. doi:10.1007/s10589-008-9206-5. An algorithm similar to SA. (Cit. on pp. 370, 377).

[435] Francisco Parreño, Ramón Alvarez-Valdés, José Fernando Oliveira, and José Manuel Tamarit. "A Hybrid GRASP/VND Algorithm for Two- and Three-Dimensional Bin Packing". *Annals of Operations Research* 179(1):203–220, Sept. 2010. London, England, UK: Springer Nature Limited. ISSN: 0254-5330. doi:10.1007/s10479-008-0449-4 (cit. on pp. 364, 369, 381).

[436] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems (NeurIPS'2019)*. Dec. 8–14, 2019, Vancouver, BC, Canada. Ed. by Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett. San Diego, CA, USA: The Neural Information Processing Systems Foundation (NeurIPS), 2019, pp. 8024–8035. ISBN: 978-1-7138-0793-3. URL: https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html (visited on 2024-07-18) (cit. on pp. 3, 375).

[437] Alan Paul, Vishal Sharma, and Oluwafemi Olukoya. "SQL Injection Attack: Detection, Prioritization & Prevention". *Journal of Information Security and Applications* 85:103871, Sept. 2024. Amsterdam, The Netherlands: Elsevier B.V. ISSN: 2214-2126. doi:10.1016/J.JISA.2024.103871 (cit. on p. 377).

[438] Fabian Pedregos, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vander-Plas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Edouard Duchesnay. "Scikit-learn: Machine Learning in Python". *Journal of Machine Learning Research (JMLR)* 12:2825–2830, Oct. 2011. Cambridge, MA, USA: MIT Press. ISSN: 1532-4435. doi:10.5555/1953048.2078195 (cit. on pp. 3, 377).

[439] Bo Peng, Zhipeng Lü, and T. C. Edwin Cheng. "A Tabu Search/Path Relinking Algorithm to Solve the Job Shop Scheduling Problem". *Computers & Operations Research* 53:154–164, Jan. 2015. Amsterdam, The Netherlands: Elsevier B.V. ISSN: 0305-0548. doi:10.1016/j.cor.2014.08.006. URL: http://arxiv.org/abs/1402.5613 (visited on 2026-09-30) (cit. on pp. 370, 379).

[440] Yun Peng, Yubing Wang, Yingxu Zhang, Lei Liang, Jun Zhang, Jianwei Zhang, Qin Li, and Lijun Wang. "Hardware Implementation of Improved Banker's Fixed-Point Rounding Algorithm". *IEEE Access* 13:36679–36686, Feb. 19, 2025. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE). ISSN: 2169-3536. doi:10.1109/ACCESS.2025.3543540 (cit. on pp. 40, 365).

[441] Yasset Pérez-Riverol, Laurent Gatto, Rui Wang, Timo Sachsenberg, Julian Uszkoreit, Felipe da Veiga Leprevost, Christian Fufezan, Tobias Ternent, Stephen J. Eglen, Daniel S. Katz, Tom J. Pollard, Alexander Konovalov, Robert M. Flight, Kai Blin, and Juan Antonio Vizcaíno. "Ten Simple Rules for Taking Advantage of Git and GitHub". *PLOS Computational Biology* 12(7), July 14, 2016. San Francisco, CA, USA: Public Library of Science (PLOS). ISSN: 1553-7358. doi:10.1371/JOURNAL.PCBI.1004947 (cit. on pp. 25, 344, 369).

[442] Tim Peters. "Algorithms – Introduction". In: *Python Cookbook*. Ed. by David Ascher. 1st ed. Sebastopol, CA, USA: O'Reilly Media, Inc., July 2002. Chap. 17. ISBN: 978-0-596-00167-4. URL: https://www.oreilly.com/library/view/python-cookbook/0596001673/ch17.html (visited on 2024-11-07) (cit. on pp. 223, 224, 361).

[443] Tim Peters. *The Zen of Python*. Python Enhancement Proposal (PEP) 20. Beaverton, OR, USA: Python Software Foundation (PSF), Aug. 19–22, 2004. URL: https://peps.python.org/pep-0020 (visited on 2025-08-03) (cit. on p. 186).

[444] Ferdinando Pezzella and Emanuela Merelli. "A Tabu Search Method Guided by Shifting Bottleneck for the Job Shop Scheduling Problem". *European Journal of Operational Research* 120(2):297–310, Jan. 2000. Amsterdam, The Netherlands: Elsevier B.V. ISSN: 0377-2217. doi:10.1016/S0377-2217(99)00158-7. URL: https://www2.cs.sfu.ca/CourseCentral/827/havens/papers/topic%5C%2310(JobShop)/Tabu%5C%20With%5C%20Shifting.pdf (visited on 2026-09-30) (cit. on p. 379).

[445] Amit Phalgun, Cory Kissinger, Margaret M. Burnett, Curtis R. Cook, Laura Beckwith, and Joseph R. Ruthruff. "Garbage in, Garbage out? An Empirical Look at Oracle Mistakes by End-User Programmers". In: *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'2005)*. Sept. 21–24, 2005, Dallas, TX, USA. Ed. by Martin Erwig and Andy Schürr. Los Alamitos, CA, USA: IEEE Computer Society, 2005, pp. 45–52. ISSN: 1943-6092. ISBN: 978-0-7695-2443-6. doi:10.1109/VLHCC.2005.40 (cit. on pp. 186, 369).

[446] Martin Pincus. "Letter to the Editor – A Monte Carlo Method for the Approximate Solution of Certain Types of Constrained Optimization Problems". *Operations Research* 18(6):1225–1228, Nov.–Dec. 1970. Catonsville, MD, USA: The Institute for Operations Research and the Management Sciences (INFORMS). ISSN: 0030-364X. doi:10.1287/opre.18.6.1225 (cit. on p. 377).

[447] Marcelo Pinto, Cristóvão Silva, Matthias Thürer, and Samuel Moniz. "Survey in Operations Research and Management Science. Nesting and Scheduling Optimization of Additive Manufacturing Systems: Mapping the Territory". *Computers & Operations Research* 165(106592), Feb. 2024. Amsterdam, The Netherlands: Elsevier B.V. ISSN: 0305-0548. doi:10.1016/j.cor.2024.106592. Mentions some scheduling tasks of Additive Manufacturing jobs can be viewed as instance of the 2BP|R|F. (Cit. on p. 364).

[448] pip Developers. *pip Documentation v24.3.1*. Beaverton, OR, USA: Python Software Foundation (PSF), Oct. 27, 2024. URL: https://pip.pypa.io (visited on 2024-12-25) (cit. on pp. 329, 375).

[449] pip Developers. "Requirements File Format". In: *pip Documentation v24.3.1*. Beaverton, OR, USA: Python Software Foundation (PSF), Oct. 27, 2024. URL: https://pip.pypa.io/en/stable/reference/requirements-file-format (visited on 2024-12-25) (cit. on pp. 330, 362).

[450] "POSIX Regular Expressions". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 20, 2025. Chap. 9.7.3. URL: https://www.postgresql.org/docs/17/functions-matching.html#FUNCTIONS-POSIX-REGEXP (visited on 2025-02-27) (cit. on p. 376).

[451] *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 2025. URL: https://www.postgresql.org/docs/17/index.html (visited on 2025-02-25).

[452] *PostgreSQL Essentials: Leveling Up Your Data Work*. Sebastopol, CA, USA: O'Reilly Media, Inc., Mar. 2024 (cit. on p. 375).

[453] Philippe PRADOS and Maggie Moss. *Allow Writing Union Types as `X | Y`*. Python Enhancement Proposal (PEP) 604. Beaverton, OR, USA: Python Software Foundation (PSF), Aug. 28, 2019. URL: https://peps.python.org/pep-0604 (visited on 2024-10-09) (cit. on pp. 106, 137, 183).

[454] Prague, Czech Republic: JetBrains. *pycharm-community: PyCharm Community Edition*. London, England, UK: Canonical Ltd., Dec. 12, 2024. URL: https://snapcraft.io/pycharm-community (visited on 2025-01-01) (cit. on p. 12).

[455] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. "1.1 Error, Accuracy, and Stability". In: *Numerical Recipes: The Art of Scientific Computing*. 3rd ed. Cambridge, England, UK: Cambridge University Press (CUP), 2007–2011. Chap. 1 Preliminaries, pp. 8–12. ISBN: 978-0-521-88068-8. URL: https://numerical.recipes/book.html (visited on 2024-07-27). Version 3.04 (cit. on pp. 40, 355).

[456] S. D. Prestwich. *Improved Branch-and-Bound for Low Autocorrelation Binary Sequences*. arXiv.org: Computing Research Repository (CoRR) abs/1305.6187. Ithaca, NY, USA: Cornell Universiy Library, Apr. 3, 2024. doi:10.48550/arXiv.1305.6187. URL: https://arxiv.org/abs/1305.6187 (visited on 2013-07-23). arXiv:1305.6187v2 [cs.AI] 23 Jul 2013 (cit. on p. 371).

[457] "Private Name Mangling". In: *Python 3 Documentation*. *The Python Language Reference*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. Chap. 6.2.1.1. URL: https://docs.python.org/3/reference/expressions.html#private-name-mangling (visited on 2025-09-23) (cit. on pp. 264, 359).

[458] "Private Variables". In: *Python 3 Documentation*. *The Python Tutorial*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. Chap. 9.6. URL: https://docs.python.org/3/tutorial/classes.html#private-variables (visited on 2025-09-23) (cit. on pp. 264, 359).

[459] *Programming Languages – C, Working Document of SC22/WG14*. International Standard ISO/3IEC9899:2017 C17 Ballot N2176. Geneva, Switzerland: International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC), Nov. 2017. URL: https://files.lhmouse.com/standards/ISO%20C%20N2176.pdf (visited on 2024-06-29) (cit. on pp. 31, 366).

[460] "programming: Meaning of *programming* in English". In: *Cambridge Dictionary English (UK)*. Cambridge, England, UK: Cambridge University Press & Assessment, June 2024. URL: https://dictionary.cambridge.org/dictionary/english/programming (visited on 2024-06-17) (cit. on p. 1).

[461] Pylint Contributors. *Pylint*. Toulouse, Occitanie, France: Logilab, 2003–2024. URL: https://pylint.readthedocs.io/en/stable (visited on 2024-09-24) (cit. on pp. 152, 361, 375).

[462] *python™*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://python.org (visited on 2025-04-17) (cit. on p. 5).

[463] *Python 3 Documentation*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3 (visited on 2024-07-05) (cit. on pp. 5, 51–55, 62, 361).

[464] *Python 3 Documentation*. *Python HOWTOs*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/howto (visited on 2025-04-27) (cit. on p. 5).

[465] *Python 3 Documentation*. *Python Setup and Usage*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/using (visited on 2024-07-05) (cit. on pp. 5, 9, 62).

[466] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. *Improving Language Understanding with Unsupervised Learning*. Tech. rep. San Francisco, CA, USA: OpenAI Inc., June 11, 2018. URL: https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf (visited on 2025-04-28) (cit. on p. 372).

[467] Sanatan Rai and George Vairaktarakis. "NP-Complete Problems and Proof Methodology". In: *Encyclopedia of Optimization*. Ed. by Christodoulos A. Floudas and Panos Miltiades Pardalos. 2nd ed. Boston, MA, USA: Springer, Sept. 2008, pp. 2675–2682. ISBN: 978-0-387-74758-3. doi:10.1007/978-0-387-74759-0_462 (cit. on p. 374).

[468] Luciano Ramalho. *Fluent Python*. 2nd ed. Sebastopol, CA, USA: O'Reilly Media, Inc., Apr. 2022. ISBN: 978-1-4920-5635-5 (cit. on p. 4).

[469] Soraya B. Rana and L. Darrell Whitley. "Genetic Algorithm Behavior in the MAXSAT Domain". In: *5th International Conference on Parallel Problem Solving from Nature (PPSN V)*. Sept. 27–30, 1998, Amsterdam, The Netherlands. Ed. by Ágoston Endre Eiben, Thomas Bäck, Marc Schoenauer, and Hans-Paul Schwefel. Lecture Notes in Computer Science (LNCS). 1998, pp. 785–794. ISSN: 0302-9743. ISBN: 978-3-540-65078-2. doi:10.1007/BFb0056920 (cit. on p. 372).

[470] Sebastian Raschka, Yuxi Liu, and Vahid Mirjalili. *Machine Learning with PyTorch and Scikit-learn*. Birmingham, England, UK: Packt Publishing Ltd, Feb. 2022. ISBN: 978-1-80181-931-2 (cit. on pp. 3, 375, 377).

[471] Abhishek Ratan, Eric Chou, Pradeeban Kathiravelu, and Dr. M.O. Faruque Sarker. *Python Network Programming*. Birmingham, England, UK: Packt Publishing Ltd, Jan. 2019. ISBN: 978-1-78883-546-6 (cit. on pp. 5, 366).

[472] Federico Razzoli. *Mastering MariaDB*. Birmingham, England, UK: Packt Publishing Ltd, Sept. 2014. ISBN: 978-1-78398-154-0 (cit. on p. 372).

[473] "`re` – Regular Expression Operations". In: *Python 3 Documentation*. *The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/library/re.html#module-re (visited on 2024-11-01) (cit. on pp. 211, 376).

[474] *Real Python Tutorials*. Vancouver, BC, Canada: DevCademy Media Inc., 2021–2025. URL: https://realpython.com (visited on 2025-04-17) (cit. on p. 6).

[475] Mike Reichardt, Michael Gundall, and Hans D. Schotten. "Benchmarking the Operation Times of NoSQL and MySQL Databases for Python Clients". In: *47th Annual Conference of the IEEE Industrial Electronics Society (IECON'2021*. Oct. 13–15, 2021, Toronto, ON, Canada. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE), 2021, pp. 1–8. ISSN: 2577-1647. ISBN: 978-1-6654-3554-3. doi:10.1109/IECON48115.2021.9589382 (cit. on p. 373).

[476] Gerhard Reinelt. "TSPLIB – A Traveling Salesman Problem Library". *ORSA Journal on Computing* 3(4):376–384, Aut. 1991. Catonsville, MD, USA: The Institute for Operations Research and the Management Sciences (INFORMS). ISSN: 0899-1499. doi:10.1287/ijoc.3.4.376 (cit. on p. 379).

[477] Gerhard Reinelt. *TSPLIB95*. Tech. rep. Heidelberg, Baden-Württemberg, Germany: Universität Heidelberg, Institut für Angewandte Mathematik, 1995–Apr. 28, 2016. URL: https://github.com/shredderzwj/TSPLIB (visited on 2026-09-25). The original URL of the repository has become defunct in 2026. Therefore, we provide the repository listed as main URL which contains the data. Additionally, you may also check https://github.com/thomasWeise/tspSuite of our own project *TSPSuite*, which contains all the TSPLib instances as well as Java code to load them. (Cit. on p. 379).

[478] *Repositories Documentation*. San Francisco, CA, USA: GitHub Inc, 2025. URL: https://docs.github.com/en/repositories (visited on 2025-02-05) (cit. on p. 344).

[479] Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. Request for Comments (RFC) 8446. Wilmington, DE, USA: Internet Engineering Task Force (IETF), Aug. 2018. URL: https://www.ietf.org/rfc/rfc8446.txt (visited on 2025-02-05) (cit. on p. 379).

[480] Mark Richards and Neal Ford. *Fundamentals of Software Architecture: An Engineering Approach*. Sebastopol, CA, USA: O'Reilly Media, Inc., Jan. 2020. ISBN: 978-1-4920-4345-4 (cit. on p. 366).

[481] Hans Riesel. *Prime Numbers and Computer Methods for Factorization*. 2nd ed. Vol. 126 of Progress in Mathematics (PM). New York, NY, USA: Springer Science+Business Media, LLC, Oct. 1, 1994–Sept. 30, 2012. ISSN: 0743-1643. ISBN: 978-0-8176-3743-9. doi:10.1007/978-1-4612-0251-6. Boston, MA, USA: Birkhäuser (cit. on pp. 147, 229, 241, 243).

[482] Ori Roth. "Python Type Hints Are Turing Complete (Pearl/Brave New Idea)". In: *37th European Conference on Object-Oriented Programming (ECOOP'2023)*. July 17–21, 2023, Seattle, WA, USA. Ed. by Karim Ali and Guido Salvaneschi. Vol. 263 of Leibniz International Proceedings in Informatics (LIPIcs). Wadern, Saarland, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 44:1–44:15. ISSN: 1868-8969. ISBN: 978-3-95977-281-5. doi:10.4230/LIPICS.ECOOP.2023.44 (cit. on p. 105).

[483] Kristian Rother. *Pro Python Best Practices: Debugging, Testing and Maintenance*. New York, NY, USA: Apress Media, LLC, Mar. 2017. ISBN: 978-1-4842-2241-6 (cit. on pp. 5, 303, 362, 367).

[484] Ernest E. Rothman, Rich Rosen, and Brian Jepson. *Mac OS X for Unix Geeks*. 4th ed. Sebastopol, CA, USA: O'Reilly Media, Inc., Sept. 2008. ISBN: 978-0-596-52062-5 (cit. on p. 372).

[485] Günter Rudolph. *Convergence Properties of Evolutionary Algorithms*. Vol. 35 of Forschungsergebnisse zur Informatik. Hamburg, Germany: Verlag Dr. Kovač, 1997. ISSN: 1435-6260. ISBN: 978-3-86064-554-3 (cit. on p. 371).

[486] Per Runeson. "A Survey of Unit Testing Practices". *IEEE Software* 23(4):22–29, July–Aug. 2006. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE). ISSN: 0740-7459. doi:10.1109/MS.2006.91 (cit. on pp. 169, 170, 380).

[487] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (AIMA)*. 4th ed. Hoboken, NJ, USA: Pearson Education, Inc. ISBN: 978-1-292-40113-3. URL: https://aima.cs.berkeley.edu (visited on 2024-06-27) (cit. on pp. 3, 142, 364).

[488] Yeonhee Ryou, Sangwoo Joh, Joonmo Yang, Sujin Kim, and Youil Kim. "Code Understanding Linter to Detect Variable Misuse". In: *37th IEEE/ACM International Conference on Automated Software Engineering (ASE'2022)*. Oct. 10–14, 2022, Rochester, MI, USA. New York, NY, USA: Association for Computing Machinery (ACM), 2022, 133:1–133:5. ISBN: 978-1-4503-9475-8. doi:10.1145/3551349.3559497 (cit. on pp. 117, 371).

[489] Yoram Sagher. "What Pythagoras Could Have Done". *The American Mathematical Monthly* 95(2):117, Feb. 1988. London, England, UK: Taylor and Francis Ltd. ISSN: 1930-0972. doi:10.1080/00029890.1988.11971978 (cit. on p. 38).

[490] Ahmad Sahar. *iOS 26 Programming for Beginners*. 10th ed. Birmingham, England, UK: Packt Publishing Ltd, Nov. 2025. ISBN: 978-1-80602-393-6 (cit. on p. 381).

[491] Sartaj Sahni and Teofilo Francisco Gonzalez Arce. "P-complete Approximation Problems". *Journal of the ACM (JACM)* 23(3):555–565, July 1976. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0004-5411. doi:10.1145/321958.321975. URL: https://sites.cs.ucsb.edu/~teo/papers/JACM-PCom.pdf (visited on 2026-02-21) (cit. on p. 376).

[492] Stephen R. Schach. *Object-Oriented Software Engineering*. New York, NY, USA: McGraw-Hill, Sept. 2007. ISBN: 978-0-07-352333-0 (cit. on p. 374).

[493] Guntram Scheithauer. *Introduction to Cutting and Packing Optimization: Problems, Modeling Approaches, Solution Methods*. Vol. 263 of International Series in Operations Research & Management Science (ISOR). Cham, Switzerland: Springer, 2018. ISSN: 0884-8289. ISBN: 978-3-319-64402-8. doi:10.1007/978-3-319-64403-5 (cit. on p. 366).

[494] Neil Schemenauer, Tim Peters, and Magnus Lie Hetland. *Simple `Generators`*. Python Enhancement Proposal (PEP) 255. Beaverton, OR, USA: Python Software Foundation (PSF), May 18, 2001. URL: https://peps.python.org/pep-0255 (visited on 2024-11-08) (cit. on pp. 240, 244).

[495] Stephan Scheuermann. "SQL Injection". In: *1st Kassel Student Workshop on Security in Distributed Systems (KaSWoSDS'2008)*. Feb. 13, 2008, Kassel, Hessen, Germany. Ed. by Thomas Weise (汤卫思) and Philipp Andreas Baer. Vol. 2008-1 of Kasseler Informatikschriften (KIS). Kassel, Hessen, Germany: Universität Kassel, Universitätsbibliothek, Apr. 14, 2008. Chap. 3, pp. 35–49. URL: https://kobra.uni-kassel.de/items/a6134d61-d5c3-4cb8-804f-bbb89a3f59a7 (visited on 2025-08-23) (cit. on p. 377).

[496] Daniel R. Schlegel. "The Böhm-Jacopini Theorem and an Introduction to Structured Programming with Python". In: *CSE 111: Great Ideas in Computer Science*. Buffalo, NY, USA: Universityat Buffalo (UB), The State University of New York, Sum. 2011. Chap. 5. URL: https://danielschlegel.org/teaching/111/lecture5.html (visited on 2025-09-01) (cit. on p. 143).

[497] Larkin Ridgway Scott. "Numerical Algorithms". In: *Numerical Analysis*. Princeton, NJ, USA: Princeton University Press, Apr. 18, 2011. Chap. 1. ISBN: 978-1-4008-3896-7. doi:10.1515/9781400838967-002. URL: https://assets.press.princeton.edu/chapters/s9487.pdf (visited on 2024-09-25) (cit. on pp. 155, 156).

[498] Winfried Seimert. *LibreOffice 7.3 – Praxiswissen für Ein- und Umsteiger*. Blaufelden, Schwäbisch Hall, Baden-Württemberg, Germany: mitp Verlags GmbH & Co. KG, Apr. 2022. ISBN: 978-3-7475-0504-5 (cit. on pp. 371, 373).

[499] Syamal K. Sen and Ravi P. Agarwal. "Existence of year zero in astronomical counting is advantageous and preserves compatibility with significance of AD, BC, CE, and BCE". In: *Zero – A Landmark Discovery, the Dreadful Void, and the Ultimate Mind*. Amsterdam, The Netherlands: Elsevier B.V., 2016. Chap. 5.5, pp. 94–95. ISBN: 978-0-08-100774-7. doi:10.1016/C2015-0-02299-7 (cit. on pp. 365, 366).

[500] "Sequences". In: *Python 3 Documentation*. *The Python Language Reference*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. Chap. 3.2.5. URL: https://docs.python.org/3/reference/datamodel.html#sequences (visited on 2024-08-24) (cit. on pp. 112, 116).

[501] "Set Types – `set`, `frozenset`". In: *Python 3 Documentation*. *The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/library/stdtypes.html#set (visited on 2024-08-27) (cit. on pp. 124, 126).

[502] Veenu Setia, Emily Rodriguez, Aakanksha Gaur, Meg Matthias, and Gloria Lotha. "Leap Year". In: *Encyclopaedia Britannica*. Ed. by The Editors of Encyclopaedia Britannica. Chicago, IL, USA: Encyclopædia Britannica, Inc., Aug. 20, 2024. URL: https://www.britannica.com/science/leap-year-calendar (visited on 2024-08-29) (cit. on p. 133).

[503] "Set Types". In: *Python 3 Documentation*. *The Python Language Reference*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. Chap. 3.2.6. URL: https://docs.python.org/3/reference/datamodel.html#set-types (visited on 2024-08-27) (cit. on p. 126).

[504] Yakov Shafranovich. *Common Format and MIME Type for Comma-Separated Values (CSV) Files*. Request for Comments (RFC) 4180. Wilmington, DE, USA: Internet Engineering Task Force (IETF), Oct. 2005. URL: https://www.ietf.org/rfc/rfc4180.txt (visited on 2025-02-05) (cit. on p. 367).

[505] Ali Shahrokni and Robert Feldt. "A Systematic Review of Software Robustness". *Information and Software Technology* 55(1):1–17, Jan. 2013. Amsterdam, The Netherlands: Elsevier B.V. ISSN: 0950-5849. doi:10.1016/J.INFSOF.2012.06.002. URL: http://robertfeldt.net/publications/shahrokni_2013_sysrev_robustness.pdf (visited on 2024-10-29) (cit. on p. 186).

[506] Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge, England, UK: Cambridge University Press (CUP), July 2014. ISBN: 978-1-107-05713-5. URL: http://www.cs.huji.ac.il/~shais/UnderstandingMachineLearning (visited on 2024-06-27) (cit. on pp. 3, 142, 373).

[507] Jonathan Richard Shewchuk. "Adaptive Precision Floating-Point Arithmetic and Fast Robust Geometric Predicates". *Discrete & Computational Geometry* 18(3):305–363, Oct. 1997. London, England, UK: Springer Nature Limited. ISSN: 0179-5376. doi:10.1007/PL00009321. URL: https://people.eecs.berkeley.edu/~jrs/papers/robustr.pdf (visited on 2024-11-19). Discusses LIU Hui's (刘徽) comments on the *Jiu Zhang Suanshu* (九章算术), including an algorithm for computing the square root and one for approximating π (cit. on pp. 155, 265).

[508] Guoyong Shi, Hitoshi Iima, and Nobuo Sannomiya. "New Encoding Scheme for Solving Job Shop Problems by Genetic Algorithm". In: *35th IEEE Conference on Decision and Control (CDC'96)*. Vol. 4. Dec. 11–13, 1996. Kōbe, Hyōgo-ken, Japan: Institute of Electrical and Electronics Engineers (IEEE), 1997, pp. 4395–4400. ISBN: 978-0-7803-3590-5. doi:10.1109/CDC.1996.577484. URL: https://www.researchgate.net/publication/224238934 (visited on 2026-09-30). Proposes a permutations-with-repetitions based encoding for the JSSP. (Cit. on pp. 368, 370, 406).

[509] Oleg V. Shylo. *Job Shop Scheduling (Personal Homepage)*. Knoxville, TN, USA: Oleg V. Shylo, Aug. 2019. URL: http://optimizizer.com/jobshop.php (visited on 2026-09-30). Contains many JSSP benchmark instances. (Cit. on pp. 370, 426).

[510] Oleg V. Shylo and Hesam Shams. *Boosting Binary Optimization via Binary Classification: A Case Study of Job Shop Scheduling*. arXiv.org: Computing Research Repository (CoRR) abs/1808.10813. Proposes a TS using regression to adapt the tabu tenures for each variable of the solution vector of a JSSP separately. Many results are available in the GitHub repository https://github.com/quasiquasar/gta-jobshop-data; other results are published Shylo's website http://optimizizer.com/jobshop.php. Ithaca, NY, USA: Cornell Universiy Library, Aug. 31, 2018. doi:10.48550/arXiv.1808.10813. URL: https://arxiv.org/abs/1808.10813 (visited on 2026-09-30) (cit. on pp. 370, 379).

[511] Ellen Siever, Stephen Figgins, Robert Love, and Arnold Robbins. *Linux in a Nutshell*. 6th ed. Sebastopol, CA, USA: O'Reilly Media, Inc., Sept. 2009. ISBN: 978-0-596-15448-6 (cit. on pp. 365, 371).

[512] Laurence E. Sigler. *Fibonacci's Liber Abaci: A Translation into Modern English of Leonardo Pisano's Book of Calculation*. Sources and Studies in the History of Mathematics and Physical Sciences. New York, NY, USA: Springer New York, Sept. 10, 2002. ISSN: 2196-8810. ISBN: 978-0-387-95419-6. doi:10.1007/978-1-4613-0079-3 (cit. on pp. 241, 242).

[513] Abraham "Avi" Silberschatz, Henry F. "Hank" Korth, and S. Sudarshan. *Database System Concepts*. 7th ed. New York, NY, USA: McGraw-Hill, Mar. 2019. ISBN: 978-0-07-802215-9 (cit. on pp. 124, 128, 285).

[514] Bryan Sills, Brian Gardner, Kristin Marsicano, and Chris Stewart. *Android Programming: The Big Nerd Ranch Guide*. 5th ed. Reading, MA, USA: Addison-Wesley Professional, May 2022. ISBN: 978-0-13-764579-4 (cit. on p. 364).

[515] Anna Skoulikari. *Learning Git*. Sebastopol, CA, USA: O'Reilly Media, Inc., May 2023. ISBN: 978-1-0981-3391-7 (cit. on pp. 25, 344, 369).

[516] Brett Slatkin. *Effective Python: 125 Specific Ways to Write Better Python*. 3rd ed. Reading, MA, USA: Addison-Wesley Professional, Nov. 2024. ISBN: 978-0-13-817239-8 (cit. on p. 4).

[517] Neil James Alexander Sloane. *Decimal Expansion of Golden Ratio phi (or tau) = (1 + sqrt(5))/2*. Ed. by John Horton Conway. Vol. A001622 of The On-Line Encyclopedia of Integer Sequences. Highland Park, NJ, USA: The OEIS Foundation Inc., Dec. 13, 2024. URL: https://oeis.org/A001622 (visited on 2024-12-14) (cit. on pp. 312, 363).

[518] Neil James Alexander Sloane. *Decimal Expansion of Square Root of 2*. Ed. by John Horton Conway. Vol. A002193 of The On-Line Encyclopedia of Integer Sequences. Highland Park, NJ, USA: The OEIS Foundation Inc., Dec. 13, 2024. URL: https://oeis.org/A002193 (visited on 2024-12-14) (cit. on p. 312).

[519] Drew Smith. *Modern Apple Platform Administration – macOS and iOS Essentials (2025)*. Birmingham, England, UK: Packt Publishing Ltd, Feb. 2025. ISBN: 978-1-80580-309-6 (cit. on pp. 370, 372).

[520] Eric V. "ericvsmith" Smith. "Improve f-string implementation: `FORMAT_VALUE` opcode #69669". In: *python/cpython Issues*. San Francisco, CA, USA: GitHub Inc, Oct. 26–Nov. 2, 2015. URL: https://github.com/python/cpython/issues/69669 (visited on 2025-08-04) (cit. on p. 76).

[521] Eric V. "ericvsmith" Smith. *Literal String Interpolation*. Python Enhancement Proposal (PEP) 498. Beaverton, OR, USA: Python Software Foundation (PSF), Nov. 6, 2016–Sept. 9, 2023. URL: https://peps.python.org/pep-0498 (visited on 2024-07-25) (cit. on pp. 73, 368).

[522] John Miles Smith and Philip Yen-Tang Chang. "Optimizing the Performance of a Relational Algebra Database Interface". *Communications of the ACM (CACM)* 18(10):568–579, Oct. 1975. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/361020.361025 (cit. on p. 376).

[523] *Snap Documentation*. London, England, UK: Canonical Ltd., 2025. URL: https://snapcraft.io/docs (visited on 2025-01-01) (cit. on p. 12).

[524] Hossein M. Soroush. "Scheduling in Stochastic Bicriteria Single Machine Systems with Set-up Times". *International Journal of Planning and Scheduling* 1(1–2):109–145, Jan. 2011. Geneva, Switzerland: Inderscience Publishers. ISSN: 2044-494X. doi:10.1504/IJPS.2011.044605. A scheduling problem is phrased as QAP. (Cit. on p. 376).

[525] Sphinx Developers. "Doc Comments and Docstrings". In: `sphinx.ext.autodoc` – *Include Documentation from Docstrings*. Oct. 13, 2024. URL: https://www.sphinx-doc.org/en/master/usage/extensions/autodoc.html#doc-comments-and-docstrings (visited on 2024-12-12) (cit. on pp. 251, 254, 292, 358, 359).

[526] *SQLite*. Charlotte, NC, USA: Hipp, Wyrick & Company, Inc. (Hwaci), 2025. URL: https://sqlite.org (visited on 2025-04-24) (cit. on p. 378).

[527] "SQL Commands". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 20, 2025. Chap. Part VI. Reference. URL: https://www.postgresql.org/docs/17/sql-commands.html (visited on 2025-02-25) (cit. on p. 377).

[528] Pradeep Kumar Srinivasan and Graham Bleaney. *Arbitrary Literal String Type*. Python Enhancement Proposal (PEP) 675. Beaverton, OR, USA: Python Software Foundation (PSF), Nov. 30, 2021–Feb. 7, 2022. URL: https://peps.python.org/pep-0675 (visited on 2025-03-04) (cit. on pp. 108, 378).

[529] Pradeep Kumar Srinivasan and James Hilton-Balfe. `Self` *Type*. Python Enhancement Proposal (PEP) 673. Beaverton, OR, USA: Python Software Foundation (PSF), Nov. 10–17, 2021. URL: https://peps.python.org/pep-0673 (visited on 2024-12-17) (cit. on pp. 317, 359).

[530] *Stack Overflow*. New York, NY, USA: Stack Exchange Inc. URL: https://stackoverflow.com (visited on 2025-02-27) (cit. on pp. 55, 56, 63).

[531] "Stack Overflow 2024 Developer Survey". In: *Stack Overflow*. New York, NY, USA: Stack Exchange Inc., May–June 2024. URL: https://survey.stackoverflow.co/2024 (visited on 2025-06-01) (cit. on pp. 3, 4).

[532] Bogdan Stashchuk. *Git and GitHub Crash Course*. Birmingham, England, UK: Packt Publishing Ltd, June 2021. ISBN: 978-1-80181-370-9 (cit. on p. 344).

[533] Bogdan Stashchuk. *SSL Complete Guide 2021: HTTP to HTTPS*. Birmingham, England, UK: Packt Publishing Ltd, May 2021. ISBN: 978-1-83921-150-8 (cit. on pp. 369, 379).

[534] Leon Steinberg. "The Backboard Wiring Problem: A Placement Algorithm". *SIAM Review* 3(1):37–50, Jan. 1961. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics Publications (SIAM). ISSN: 0036-1445. doi:10.1137/1003003. The problem is modeled as a QAP (cit. on p. 376).

[535] Ryan K. Stephens and Ronald R. Plew. *Sams Teach Yourself SQL in 21 Days*. 4th ed. Sams Teach Yourself. Indianapolis, IN, USA: SAMS Technical Publishing and Hoboken, NJ, USA: Pearson Education, Inc., Oct. 2002. ISBN: 978-0-672-32451-2 (cit. on pp. 377, 427).

[536] Ryan K. Stephens, Ronald R. Plew, Bryan Morgan, and Jeff Perkins. *SQL in 21 Tagen. Die Datenbank-Abfragesprache SQL vollständig erklärt (in 14/21 Tagen)*. 6th ed. Burgthann, Bayern, Germany: Markt+Technik Verlag GmbH, Feb. 1998. ISBN: 978-3-8272-2020-2. Translation of [535] (cit. on p. 377).

[537] Marc Stöckel. "Ärger mit Replit: Coding-KI löscht Produktivdatenbank und verweigert Rollback". *golem.de – IT-News für Profis*, July 22, 2025. Berlin, Germany: Golem Media GmbH. URL: https://glm.io/198349 (visited on 2025-07-29) (cit. on p. 61).

[538] Marc Stöckel. "Google Antigravity: Vibe-Coding-Tool löscht unerwartet ganzen Datenträger". *golem.de – IT-News für Profis*, Dec. 2, 2025. Berlin, Germany: Golem Media GmbH. URL: https://glm.io/202805 (visited on 2025-12-02) (cit. on pp. 60, 61).

[539] Robert H. Storer, S. David Wu, and Renzo Vaccari. "New Search Spaces for Sequencing Problems with Application to Job Shop Scheduling". *Management Science* 38(10):1495–1509, Oct. 1992. Catonsville, MD, USA: The Institute for Operations Research and the Management Sciences (INFORMS). ISSN: 0025-1909. doi:10.1287/mnsc.38.10.1495 (cit. on p. 370).

[540] Philip D. Straffin Jr. "Liu Hui and the First Golden Age of Chinese Mathematics". *Mathematics Magazine* 71(3):163–181, June 1998. London, England, UK: Taylor and Francis Ltd. ISSN: 0025-570X. doi:10.2307/2691200. URL: https://www.researchgate.net/publication/237334342 (visited on 2024-08-10) (cit. on pp. 88, 166, 363).

[541] Thomas Strecker and Leonhard Hennig. "Automatic Layouting of Personalized Newspaper Pages". In: *Operations Research Proceedings 2008: Selected Papers of the Annual International Conference of the German Operations Research Society (GOR)*. Sept. 3–5, 2008, Augsburg, Bayern, Germany: University of Augsburg. Ed. by Bernhard Fleischmann, Karl-Heinz Borgwardt, Robert Klein, and Axel Tuma. Vol. 2008 of Operations Research Proceedings (ORP), Gesellschaft für Operations Research e.V., (GOR). Berlin/Heidelberg, Germany: Springer-Verlag GmbH Germany, 2009, pp. 469–474. ISSN: 0721-5924. doi:10.1007/978-3-642-00142-0_76 (cit. on p. 364).

[542] "String Constants". In: chap. 4.1.2.1. URL: https://www.postgresql.org/docs/17/sql-syntax-lexical.html#SQL-SYNTAX-STRINGS (visited on 2025-08-23) (cit. on p. 367).

[543] Thomas Stützle. "Iterated Local Search for the Quadratic Assignment Problem". *European Journal of Operational Research* 174(3):1519–1539, Nov. 2006. Amsterdam, The Netherlands: Elsevier B.V. ISSN: 0377-2217. doi:10.1016/j.ejor.2005.01.066 (cit. on pp. 369, 376).

[544] Dirk Sudholt. "Crossover Speeds up Building-Block Assembly". In: *14th Genetic and Evolutionary Computation Conference (GECCO'2012)*. July 7–11, 2012, Philadelphia, PA, USA. Ed. by Terence Soule and Jason Hall Moore. New York, NY, USA: Association for Computing Machinery (ACM), 2012, pp. 689–702. ISBN: 978-1-4503-1177-9. doi:10.1145/2330163.2330260. This paper is on the Greedy $(2+1)$ GA, see also [545] (cit. on pp. 369, 428).

[545] Dirk Sudholt. "How Crossover Speeds up Building Block Assembly in Genetic Algorithms". *Evolutionary Computation* 25(2):237–274, 2017. Cambridge, MA, USA: MIT Press. ISSN: 1063-6560. doi:10.1162/EVCO_a_00171. This paper is on the Greedy $(2+1)$ GA, see also [544] (cit. on pp. 369, 428).

[546] Michael J. Sullivan and Ivan Levkivskyi. *Adding a `Final` Qualifier to `typing`*. Python Enhancement Proposal (PEP) 591. Beaverton, OR, USA: Python Software Foundation (PSF), Mar. 15, 2019. URL: https://peps.python.org/pep-0591 (visited on 2024-11-19) (cit. on pp. 254, 256, 290, 299, 358, 359).

[547] Éric D. Taillard. "Benchmarks for Basic Scheduling Problems". *European Journal of Operational Research* 64(2):278–285, Jan. 1993. Amsterdam, The Netherlands: Elsevier B.V. ISSN: 0377-2217. doi:10.1016/0377-2217(93)90182-M. URL: https://arodes.hes-so.ch/record/8224/files/Preprint.pdf (visited on 2026-02-20) (cit. on p. 370).

[548] Éric D. Taillard. "Comparison of Iterative Searches for the Quadratic Assignment Problem". *Location Science. Now incorporated into* Computers & Operations Research 3(2):87–105, Aug. 1995. Amsterdam, The Netherlands: Elsevier B.V. doi:10.1016/0966-8349(95)00008-6 (cit. on p. 379).

[549] Éric D. Taillard. "Robust Taboo Search for the Quadratic Assignment Problem". *Parallel Computing* 17(4–5):443–455, July 1991. Amsterdam, The Netherlands: Elsevier B.V. ISSN: 0167-8191. doi:10.1016/S0167-8191(05)80147-4 (cit. on pp. 376, 379, 403, 428).

[550] Éric D. Taillard and Luca M. Gambardella. *Adaptive Memories for the Quadratic Assignment Problem*. Tech. rep. IDSIA-87-97. Lugano, Switzerland: Istituto Dalle Molle Di Studi Sull Intelligenza Artificiale (IDSIA), Sept. 10, 1997. URL: https://mistic.iict-heig-vd.ch/taillard/articles.dir/TaillardGambardella1997.pdf (visited on 2026-09-28). A MA using TS from [549] as well as a ACO-based approach refining its solutions with a very simple SLS are presented. (Cit. on p. 376).

[551] Allen Taylor. *Introducing SQL and Relational Databases*. New York, NY, USA: Apress Media, LLC, Sept. 2018. ISBN: 978-1-4842-3841-7 (cit. on pp. 376, 377).

[552] "Text Sequence Type – `str`". In: *Python 3 Documentation*. *The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/library/stdtypes.html#textseq (visited on 2024-07-25) (cit. on p. 70).

[553] Alkin Tezuysal and Ibrar Ahmed. *Database Design and Modeling with PostgreSQL and MySQL*. Birmingham, England, UK: Packt Publishing Ltd, July 2024. ISBN: 978-1-80323-347-5 (cit. on pp. 373, 375).

[554] The Editors of Encyclopaedia Britannica, ed. *Encyclopaedia Britannica*. Chicago, IL, USA: Encyclopædia Britannica, Inc.

[555] "The Import System". In: *Python 3 Documentation*. *The Python Language Reference*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. Chap. 5. URL: https://docs.python.org/3/reference/import.html (visited on 2024-10-01) (cit. on p. 167).

[556] *The JSON Data Interchange Syntax*. Standard ECMA-404, 2nd Edition. Geneva, Switzerland: Ecma International, Dec. 2017. URL: https://ecma-international.org/publications-and-standards/standards/ecma-404 (visited on 2024-12-15) (cit. on pp. 316, 370).

[557] The PEP Editors. *Index of Python Enhancement Proposals (PEPs)*. Python Enhancement Proposal (PEP) 0. Beaverton, OR, USA: Python Software Foundation (PSF), July 13, 2000. URL: https://peps.python.org (visited on 2025-04-17) (cit. on pp. 6, 62).

[558] *Python 3 Documentation*. *The Python Language Reference*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/reference (visited on 2025-04-27).

[559] *The Python Package Index (PyPI)*. Beaverton, OR, USA: Python Software Foundation (PSF), 2024. URL: https://pypi.org (visited on 2024-08-17) (cit. on pp. 329, 375).

[560] *Python 3 Documentation*. *The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/library (visited on 2025-04-27) (cit. on pp. 5, 62).

[561] *Python 3 Documentation*. *The Python Tutorial*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/tutorial (visited on 2025-04-26) (cit. on pp. 5, 62).

[562] "Literals". In: *Static Typing with Python*. Ed. by The Python Typing Team. Beaverton, OR, USA: Python Software Foundation (PSF), 2021. URL: https://typing.python.org/en/latest/spec/literal.html (visited on 2025-08-29) (cit. on pp. 318, 359, 371).

[563] The Python Typing Team, ed. *Static Typing with Python*. Beaverton, OR, USA: Python Software Foundation (PSF), 2021. URL: https://typing.python.org (visited on 2025-08-28) (cit. on p. 105).

[564] *The Python Wiki*. Beaverton, OR, USA: Python Software Foundation (PSF), 2005–2025. URL: https://wiki.python.org/moin/FrontPage (visited on 2025-08-28).

[565] *The Unicode Standard, Version 15.1: Archived Code Charts*. South San Francisco, CA, USA: The Unicode Consortium, Aug. 25, 2023. URL: https://www.unicode.org/Public/15.1.0/charts/CodeCharts.pdf (visited on 2024-07-26) (cit. on pp. 80, 380).

[566] "The `with` Statement". In: *Python 3 Documentation*. *The Python Language Reference*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. Chap. 8.5. URL: https://docs.python.org/3/reference/compound_stmts.html#with (visited on 2024-12-15) (cit. on p. 208).

[567] Dirk Thierens, David Edward Goldberg, and Angela Guimarães Pereira. "Domino Convergence, Drift, and the Temporal-Salience Structure of Problems". In: pp. 535–540. doi:10.1109/ICEC.1998.700085 (cit. on p. 363).

[568] George K. Thiruvathukal, Konstantin Läufer, and Benjamin Gonzalez. "Unit Testing Considered Useful". *Computing in Science & Engineering* 8(6):76–87, Nov.–Dec. 2006. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE). ISSN: 1521-9615. doi:10.1109/MCSE.2006.124. URL: https://www.researchgate.net/publication/220094077 (visited on 2024-10-01) (cit. on pp. 169, 170, 380).

[569] Sarah Louise Thomson, Gabriela Ochoa, Daan van den Berg, Tianyu Liang (梁天宇), and Thomas Weise (汤卫思). "Entropy, Search Trajectories, and Explainability for Frequency Fitness Assignment". In: *Parallel Problem Solving from Nature (PPSN XVIII)*. Vol. 1. Sept. 14–18, 2024, Hagenberg, Mühlkreis, Austria. Ed. by Michael Affenzeller, Stephan M. Winkler, Anna V. Kononova, Heike Trautmann, Tea Tušar, Penousal Machado, and Thomas Bäck. Vol. 15148 of Lecture Notes in Computer Science (LNCS). Cham, Switzerland: Springer. ISSN: 0302-9743. ISBN: 978-3-031-70054-5. doi:10.1007/978-3-031-70055-2_23. Explores RLS, FRLS, and SA on the QAP. (Cit. on pp. 368, 373).

[570] Ulrich Wilhelm Thonemann and Andreas Bölte. *An Improved Simulated Annealing Algorithm for the Quadratic Assignment Problem*. Working Paper. Paderborn, Nordrhein-Westfalen, Germany: School of Business, Department of Production and Operations Research, University of Paderborn, 1994 (cit. on p. 377).

[571] "`timeit` – Measure Execution Time of Small Code Snippets". In: *Python 3 Documentation*. *The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/library/timeit.html (visited on 2024-11-07) (cit. on pp. 223–225, 358, 361).

[572] Marco Tomassini. "Complex Networks Analysis of the Energy Landscape of the Low Autocorrelation Binary Sequences Problem". *Physica A: Statistical Mechanics and its Applications* 577:126089, Sept. 1, 2021. Amsterdam, The Netherlands: Elsevier B.V. ISSN: 0378-4371. doi:10.1016/j.physa.2021.126089. Lists the LABS as $\mathcal{NP}$-hard. (Cit. on p. 371).

[573] Linus Torvalds. "The Linux Edge". *Communications of the ACM (CACM)* 42(4):38–39, Apr. 1999. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/299157.299165 (cit. on pp. 9, 371).

[574] Sherwin John C. Tragura. *Mastering Flask Web and API Development*. Birmingham, England, UK: Packt Publishing Ltd, Aug. 2024. ISBN: 978-1-83763-322-7 (cit. on pp. 4, 368).

[575] Brad Traversy. *Modern HTML & CSS From The Beginning 2.0*. 2nd ed. Birmingham, England, UK: Packt Publishing Ltd, July 2024. ISBN: 978-1-83588-056-2 (cit. on p. 369).

[576] "Truth Value Testing". In: *Python 3 Documentation*. *The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/library/stdtypes.html#truth (visited on 2025-10-10) (cit. on p. 77).

[577] Mariot Tsitoara. *Beginning Git and GitHub: Version Control, Project Management and Teamwork for the New Developer*. New York, NY, USA: Apress Media, LLC, Mar. 2024. ISBN: 979-8-8688-0215-7 (cit. on pp. 25, 344, 369, 381).

[578] *Turbo Pascal® Version 7.0 – User's Guide*. Scotts Valley, CA, USA: Borland International, Inc., 1992. URL: https://turbopascal.org/wp-content/uploads/Turbo_Pascal_Version_7.0_Users_Guide_1992.pdf (visited on 2025-07-28) (cit. on pp. 51, 379).

[579] Adam Turner, Bénédikt Tran, Chris Sewell, François Freitag, Jakob Lykke Andersen, Jean-François B., Stephen Finucane, Takayuki Shimizukawa, Takeshi Komiya, and Sphinx Developers. *Sphinx – Create Intelligent and Beautiful Documentation with Ease*. Oct. 13, 2024. URL: https://www.sphinx-doc.org (visited on 2024-12-12) (cit. on p. 377).

[580] Mark Tyson. "AI coding platform goes rogue during code freeze and deletes entire company database – Replit CEO apologizes after AI engine says it 'made a catastrophic error in judgment' and 'destroyed all production data'". *tom's Hardware*, July 21, 2025. New York, NY, USA: Future US, Inc. URL: https://www.tomshardware.com/tech-industry/artificial-intelligence/ai-coding-platform-goes-rogue-during-code-freeze-and-deletes-entire-company-database-replit-ceo-apologizes-after-ai-engine-says-it-made-a-catastrophic-error-in-judgment-and-destroyed-all-production-data (visited on 2025-07-31) (cit. on p. 61).

[581] Laurie A. Ulrich and Ken Cook. *Access For Dummies*. Hoboken, NJ, USA: For Dummies (Wiley), Dec. 2021. ISBN: 978-1-119-82908-9 (cit. on pp. 371, 373).

[582] *Unicode® 15.1.0*. South San Francisco, CA, USA: The Unicode Consortium, Sept. 12, 2023. ISBN: 978-1-936213-33-7. URL: https://www.unicode.org/versions/Unicode15.1.0 (visited on 2024-07-26) (cit. on pp. 80, 380).

[583] *USA Standard Code for Information Interchange (ASCII 1967)*. Tech. rep. USAS X3.4-1967. New York, NY, USA: United States of America Standards Institute (USAS), July 7, 1967. URL: https://www.sensitiveresearch.com/Archive/CharCodeHist/Files/CODES%20standards%20documents%20ASCII%20Sean%20Leonard%20Oct%202015/ASCII%2068,%20X3.4-1967.pdf (visited on 2024-07-26) (cit. on p. 79).

[584] Marat Valiev, Bogdan Vasilescu, and James D. Herbsleb. "Ecosystem-Level Determinants of Sustained Activity in Open-Source Projects: A Case Study of the PyPI Ecosystem". In: *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/SIGSOFT FSE'2018)*. Nov. 4–9, 2018, Lake Buena Vista, FL, USA. Ed. by Gary T. Leavens, Alessandro F. Garcia, and Corina S. Păsăreanu. New York, NY, USA: Association for Computing Machinery (ACM), 2018, pp. 644–655. ISBN: 978-1-4503-5573-5. doi:10.1145/3236024.3236062 (cit. on p. 375).

[585] Daan van den Berg, Florian Braam, Mark Moes, Emiel Suilen, and Sandjai Bhulai. "Almost Squares in Almost Squares: Solving the Final Instance". In: *5th International Conference on Data Analytics (DATA ANALYTICS'2016)*. Oct. 9–13, 2016, Venice, Venezia, Veneto, Italy. Ed. by Sandjai Bhulai and Ivana Semanjski. Wilmington, DE, USA: International Academy, Research, and Industry Association (IARIA), 2016, pp. 69–74. ISSN: 2308-4464. ISBN: 978-1-61208-510-4. URL: https://math.vu.nl/~sbhulai/publications/data_analytics2016b.pdf (visited on 2026-10-01). This poses a very interesting 2BP|R|F problem, where "almost squars," i.e., rectangles with sizes $(i, i+1)$, must be placed in an almost square. (Cit. on pp. 364, 365).

[586] Bruce M. Van Horn II and Quan Nguyen. *Hands-On Application Development with PyCharm*. 2nd ed. Birmingham, England, UK: Packt Publishing Ltd, Oct. 2023. ISBN: 978-1-83763-235-0 (cit. on pp. 9, 11, 375).

[587] Clarissa Van Hoyweghen, David Edward Goldberg, and Bart Naudts. "From Twomax To The Ising Model: Easy And Hard Symmetrical Problems". In: *4th Genetic and Evolutionary Computation Conference (GECCO'2002)*. July 9–13, 2002, New York, NY, USA. Ed. by William B. Langdon, Erick Cantú-Paz, Keith E. Mathias, Rajkumar Roy, David Davis, Riccardo Poli, Karthik Balakrishnan, Vasant G. Honavar, Günter Rudolph, Joachim Wegener, Larry Bull, Mitchell A. Potter, Alan C. Schultz, Julian F. Miller, Edmund K. Burke, and Natasa Jonoska. Burlington, MA, USA/San Mateo, CA, USA: Morgan Kaufmann Publishers, 2002, pp. 626–633. ISBN: 978-1-55860-878-8 (cit. on p. 380).

[588] Guido van Rossum. *Computer Programming for Everybody (Revised Proposal). A Scouting Expedition for the Programmers of Tomorrow*. CNRI Proposal 90120-1a. Reston, VA, USA: Corporation for National Research Initiatives (CNRI), July 1999. URL: https://www.python.org/doc/essays/cp4e (visited on 2024-06-27) (cit. on p. 4).

[589] Guido van Rossum and David Ascher. *Rich Comparisons*. Python Enhancement Proposal (PEP) 207. Beaverton, OR, USA: Python Software Foundation (PSF), July 25, 2000. URL: https://peps.python.org/pep-0207 (visited on 2024-12-08) (cit. on pp. 281, 295).

[590] Guido van Rossum and Alyssa Coghlan. *The "with" Statement*. Python Enhancement Proposal (PEP) 343. Beaverton, OR, USA: Python Software Foundation (PSF), May 13, 2005–July 30, 2006. URL: https://peps.python.org/pep-0343 (visited on 2024-12-15) (cit. on p. 208).

[591] Guido van Rossum and Raymond Hettinger. *Conditional Expressions*. Python Enhancement Proposal (PEP) 308. Beaverton, OR, USA: Python Software Foundation (PSF), Feb. 7–11, 2003. URL: https://peps.python.org/pep-0308 (visited on 2025-08-31) (cit. on p. 140).

[592] Guido van Rossum and Łukasz Langa. *Type Hints*. Python Enhancement Proposal (PEP) 484. Beaverton, OR, USA: Python Software Foundation (PSF), Sept. 29, 2014. URL: https://peps.python.org/pep-0484 (visited on 2024-08-22) (cit. on pp. 105, 106, 108, 122, 380).

[593] Guido van Rossum and Ivan Levkivskyi. *The Theory of Type Hints*. Python Enhancement Proposal (PEP) 483. Beaverton, OR, USA: Python Software Foundation (PSF), Dec. 19, 2014. URL: https://peps.python.org/pep-0483 (visited on 2026-10-11) (cit. on p. 122).

[594] Guido van Rossum, Barry Warsaw, and Alyssa Coghlan. *Style Guide for Python Code*. Python Enhancement Proposal (PEP) 8. Beaverton, OR, USA: Python Software Foundation (PSF), July 5, 2001. URL: https://peps.python.org/pep-0008 (visited on 2024-07-27) (cit. on pp. 2, 68, 79, 81, 82, 84, 86, 88, 119, 133, 161, 163, 168, 197, 251, 264, 266, 290, 355–359, 367).

[595] Klaas van Schelven. "Copilot Induced Crash". In: *Bugsink – Self-hosted Error Tracking: Blog*. Utrecht, The Netherlands: Bugsink B.V., Jan. 15, 2025. URL: https://www.bugsink.com/blog/copilot-induced-crash (visited on 2025-05-08) (cit. on p. 60).

[596] Sander van Vugt. *Linux Fundamentals*. 2nd ed. Hoboken, NJ, USA: Pearson IT Certification, June 2022. ISBN: 978-0-13-792931-3 (cit. on pp. 365, 371).

[597] Dieter Vandesande, Christoph Jabs, Jeremias Berg, Johannes Fichte, George Katsirelos, Ruben Martins, André Schidler, and Dominik Schreiber. *MaxSAT Evaluation 2026: Solvers, Benchmarks & Results*. v1. Geneva, Switzerland: zenodo (hosted by European Organization for Nuclear Research (CERN)), July 14, 2026. doi:10.5281/zenodo.21358014. See also https://maxsat-evaluations.github.io. (Cit. on p. 373).

[598] Daniele "dvarrazzo" Varrazzo, Federico "fogzot" Di Gregorio, and Jason "jerickso" Erickson. *Psycopg*. London, England, UK: The Psycopg Team, 2010–2023. URL: https://www.psycopg.org (visited on 2025-02-02) (cit. on pp. 3, 375).

[599] Daniele "dvarrazzo" Varrazzo, Federico "fogzot" Di Gregorio, and Jason "jerickso" Erickson. "Static Typing". In: *Psycopg 3 – PostgreSQL Database Adapter for Python*. London, England, UK: The Psycopg Team, June 21, 2022. URL: https://www.psycopg.org/psycopg3/docs/advanced/typing.html (visited on 2025-03-04) (cit. on pp. 107, 108, 378).

[600] "`venv` – Creation of Virtual Environments". In: *Python 3 Documentation. The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/library/venv.html (visited on 2024-12-25) (cit. on p. 381).

[601] Kristian Verduin, Sarah Louise Thomson, and Daan van den Berg. "Too Constrained for Genetic Algorithms. Too Hard for Evolutionary Computing. The Traveling Tournament Problem". In: *15th International Joint Conference on Computational Intelligence (IJCCI'23)*. Nov. 13–15, 2023, Rome, Italy. Ed. by Niki van Stein, Francesco Marcelloni, H. K. Lam, Marie Cottrell, and Joaquim Filipe. Porto, Portugal: SciTePress: Science and Technology Publications, Lda, 2023, pp. 246–257. ISSN: 2184-3236. ISBN: 978-989-758-674-3. doi:10.5220/0012192100003595 (cit. on p. 379).

[602] Brandon Vigliarolo. "Caveat Coder: Google Antigravity vibe-codes user's entire drive out of existence". *The Register*, Dec. 1, 2025. London, England, UK: Situation Publishing Limited. URL: https://www.theregister.com/2025/12/01/google_antigravity_wipes_d_drive (visited on 2025-12-01) (cit. on pp. 60, 61).

[603] Petr Vilím, Philippe Laborie, and Paul Shaw. "Failure-Directed Search for Constraint-Based Scheduling". In: *12th International Conference on the Integration of AI and OR Techniques in Constraint Programming (CPAIOR'2015)*. Ed. by Laurent Michel. Vol. 9075 of Theoretical Computer Science and General Issues (LNTCS), sub-series of Lecture Notes in Computer Science (LNCS). Barcelona, Spain: Springer, 2015, pp. 437–453. ISSN: 2512-2010. ISBN: 978-3-319-18007-6. doi:10.1007/978-3-319-18008-3_30. URL: https://www.researchgate.net/publication/280223626 (visited on 2026-09-30) (cit. on p. 370).

[604] Pauli "pv" Virtanen, Ralf Gommers, Travis E. Oliphant, Matt "mdhaber" Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, Ilhan "ilayn" Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". *Nature Methods* 17:261–272, Mar. 2, 2020. London, England, UK: Springer Nature Limited. ISSN: 1548-7091. doi:10.1038/s41592-019-0686-2. URL: http://arxiv.org/abs/1907.10121 (visited on 2024-06-26). See also arXiv:1907.10121v1 [cs.MS] 23 Jul 2019. (Cit. on pp. 3, 377).

[605] "Virtual Environments and Packages". In: *Python 3 Documentation. The Python Tutorial*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. Chap. 12. URL: https://docs.python.org/3/tutorial/venv.html (visited on 2024-12-24) (cit. on pp. 330, 381).

[606] *W3Schools: Python Tutorials*. Sandnes, Rogaland, Norway: Refsnes Data AS, 1999–2025. URL: https://www.w3schools.com/python (visited on 2025-04-17) (cit. on p. 6).

[607] Christian Walck. *Hand-Book on Statistical Distributions for Experimentalists*. Internal Report SUF-PFY/96-01. Stockholm, Sweden: University of Stockholm, Dec. 11, 1996–Sept. 10, 2007. URL: https://www.stat.rice.edu/~dobelman/textfiles/DistributionsHandbook.pdf (visited on 2025-09-05) (cit. on pp. 177, 184, 375).

[608] Gregory R. Warnes. *IEEE 754 Floating Point Special Values [Rejected]*. Python Enhancement Proposal (PEP) 754. Beaverton, OR, USA: Python Software Foundation (PSF), Mar. 28, 2003. URL: https://peps.python.org/pep-0754 (visited on 2024-12-08) (cit. on p. 281).

[609] Barry Warsaw. *Dict Comprehensions*. Python Enhancement Proposal (PEP) 274. Beaverton, OR, USA: Python Software Foundation (PSF), Oct. 25, 2001. URL: https://peps.python.org/pep-0274 (visited on 2024-11-08) (cit. on p. 229).

[610] Barry Warsaw. *List Comprehensions*. Python Enhancement Proposal (PEP) 202. Beaverton, OR, USA: Python Software Foundation (PSF), July 13, 2000. URL: https://peps.python.org/pep-0202 (visited on 2024-11-08) (cit. on p. 220).

[611] Thomas Weise (汤卫思). *Databases*. Hefei, Anhui, China (中国安徽省合肥市): Hefei University (合肥大学), School of Artificial Intelligence and Big Data (人工智能与大数据学院), 2025. URL: https://thomasweise.github.io/databases (visited on 2025-01-05) (cit. on pp. 108, 344, 350, 351, 367, 371, 376, 377).

[612] Thomas Weise (汤卫思). *Global Optimization Algorithms – Theory and Application*. self-published, 2009. URL: https://www.researchgate.net/publication/200622167 (visited on 2025-07-25) (cit. on pp. 367, 368, 372, 377).

[613] Thomas Weise (汤卫思). *jsspInstancesAndResults: Results, Data, and Instances of the Job Shop Scheduling Problem*. San Francisco, CA, USA: GitHub Inc, 2019–2020. URL: https://github.com/thomasWeise/jsspInstancesAndResults (visited on 2025-08-20) (cit. on p. 370).

[614] Thomas Weise (汤卫思). *Programming with C*. Hefei, Anhui, China (中国安徽省合肥市): Hefei University (合肥大学), School of Artificial Intelligence and Big Data (人工智能与大数据学院), 2026. URL: https://thomasweise.github.io/programmingWithC (visited on 2026-05-10) (cit. on p. 366).

[615] Thomas Weise (汤卫思). *Programming with Python*. Hefei, Anhui, China (中国安徽省合肥市): Hefei University (合肥大学), School of Artificial Intelligence and Big Data (人工智能与大数据学院), 2024–2026. URL: https://thomasweise.github.io/programmingWithPython (visited on 2025-01-05) (cit. on pp. iv, 344, 367, 375).

[616] Thomas Weise (汤卫思), Raymond Chiong, Jörg Lässig, Ke Tang (唐珂), Shigeyoshi Tsutsui, Wenxiang Chen (陈文祥), Zbigniew "Zbyszek" Michalewicz, and Xin Yao (姚新). "Benchmarking Optimization Algorithms: An Open Source Framework for the Traveling Salesman Problem". *IEEE Computational Intelligence Magazine (CIM)* 9(3):40–52, Aug. 2014. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE). ISSN: 1556-603X. doi:10.1109/MCI.2014.2326101. Applies a multitude of algorithms including SLSes, EAs, MAs, ACO, and EDAs to the TSPLib instances. (Cit. on pp. 223, 364, 379, 381).

[617] Thomas Weise (汤卫思), Xinlu Li (李新路), Yan Chen (陈岩), and Zhize Wu (吴志泽). "Solving Job Shop Scheduling Problems without using a Bias for Good Solutions". In: *23rd Genetic and Evolutionary Computation Conference (GECCO'2021), Companion Volume*. July 10–14, 2021, Lille, France. Ed. by Krzysztof Krawiec. New York, NY, USA: Association for Computing Machinery (ACM), 2021, pp. 1459–1466. ISBN: 978-1-4503-8351-6. doi:10.1145/3449726.3463124. Applies FFA to the JSSP. (Cit. on p. 368).

[618] Thomas Weise (汤卫思), Mingxu Wan (万明绪), Pu Wang (王璞), Ke Tang (唐珂), Alexandre Devert, and Xin Yao (姚新). "Frequency Fitness Assignment". *IEEE Transactions on Evolutionary Computation* 18(2):226–243, Apr. 2014. Los Alamitos, CA, USA: IEEE Computer Society. ISSN: 1089-778X. doi:10.1109/TEVC.2013.2251885. The article introducing FFA. (Cit. on p. 368).

[619] Thomas Weise (汤卫思) and Zhize Wu (吴志泽). "Replicable Self-Documenting Experiments with Arbitrary Search Spaces and Algorithms". In: *25th Conference on Genetic and Evolutionary Computation (GECCO'2023), Companion Volume*. July 15–19, 2023, Lisbon, Portugal. Ed. by Sara Silva and Luís Paquete. New York, NY, USA: Association for Computing Machinery (ACM), 2023, pp. 1891–1899. ISBN: 979-8-4007-0120-7. doi:10.1145/3583133.3596306. Introduces the experiment execution and data collection facilities of moptipy. (Cit. on pp. 3, 373).

[620] Thomas Weise (汤卫思), Zhize Wu (吴志泽), Xinlu Li (李新路), and Yan Chen (陈岩). "Frequency Fitness Assignment: Making Optimization Algorithms Invariant under Bijective Transformations of the Objective Function Value". *IEEE Transactions on Evolutionary Computation* 25(2):307–319, Apr. 2021. Los Alamitos, CA, USA: IEEE Computer Society. ISSN: 1089-778X. doi:10.1109/TEVC.2020.3032090 (cit. on pp. 363, 368, 370, 371, 374, 379, 380).

[621] Thomas Weise (汤卫思), Zhize Wu (吴志泽), Xinlu Li (李新路), Yan Chen (陈岩), and Jörg Lässig. "Frequency Fitness Assignment: Optimization without Bias for Good Solutions can be Efficient". *IEEE Transactions on Evolutionary Computation* 27(4):980–992, 2023. Los Alamitos, CA, USA: IEEE Computer Society. ISSN: 1089-778X. doi:10.1109/TEVC.2022.3191698. Applies many FFA-based algorithm variants to problems such as OneMax, LeadingOnes, Linear Harmonic Function (LinHarm), TwoMax, Trap, Jump, Plateau, Ising Model, and N-Queens. (Cit. on pp. 363, 368, 371, 379, 380).

[622] Thomas Weise (汤卫思), Yuezhong Wu (吴越钟), Raymond Chiong, Ke Tang (唐珂), and Jörg Lässig. "Global versus Local Search: The Impact of Population Sizes on Evolutionary Algorithm Performance". *Journal of Global Optimization* 66(3):511–534, Feb. 2016. London, England, UK: Springer Nature Limited. ISSN: 0925-5001. doi:10.1007/s10898-016-0417-5 (cit. on pp. 364, 379).

[623] Eric Wolfgang Weisstein. "Fibonacci Number". In: *MathWorld – A Wolfram Web Resource*. Champaign, IL, USA: Wolfram Research, Inc., Aug. 22, 2024. URL: https://mathworld.wolfram.com/FibonacciNumber.html (visited on 2024-11-08) (cit. on pp. 241, 242).

[624] Eric Wolfgang Weisstein. "Irrational Number". In: *MathWorld – A Wolfram Web Resource*. Champaign, IL, USA: Wolfram Research, Inc., Aug. 22, 2024. URL: https://mathworld.wolfram.com/IrrationalNumber.html (visited on 2026-11-04) (cit. on p. 370).

[625] Eric Wolfgang Weisstein. *MathWorld – A Wolfram Web Resource*. Champaign, IL, USA: Wolfram Research, Inc., Aug. 22, 2024. URL: https://mathworld.wolfram.com (visited on 2024-09-24).

[626] Eric Wolfgang Weisstein. "Prime Number". In: *MathWorld – A Wolfram Web Resource*. Champaign, IL, USA: Wolfram Research, Inc., Aug. 22, 2024. URL: https://mathworld.wolfram.com/PrimeNumber.html (visited on 2024-09-24) (cit. on pp. 147, 229, 241, 243).

[627] Eric Wolfgang Weisstein. "Second Fundamental Theorem of Calculus". In: *MathWorld – A Wolfram Web Resource*. Champaign, IL, USA: Wolfram Research, Inc., Sept. 4, 2025. URL: https://mathworld.wolfram.com/SecondFundamentalTheoremofCalculus.html (visited on 2025-09-08) (cit. on p. 181).

[628] Eric Wolfgang Weisstein. "Transcendental Number". In: *MathWorld – A Wolfram Web Resource*. Champaign, IL, USA: Wolfram Research, Inc., Aug. 22, 2024. URL: https://mathworld.wolfram.com/TranscendentalNumber.html (visited on 2026-11-04) (cit. on p. 379).

[629] B. P. Welford. "Note on a Method for Calculating Corrected Sums of Squares and Products". *Technometrics* 4(3):419–420, Aug. 1962. London, England, UK: Taylor and Francis Ltd. ISSN: 0040-1706. doi:10.2307/1266577. This algorithm is often used to compute the sample variance [108, 321] (cit. on pp. 380, 381).

[630] *What does PDF mean?* San Jose, CA, USA: Adobe Systems Incorporated, 2024. URL: https://www.adobe.com/acrobat/about-adobe-pdf.html (visited on 2024-12-12) (cit. on pp. 249, 277, 375).

[631] *What is a Relational Database?* Armonk, NY, USA: International Business Machines Corporation (IBM), Oct. 20, 2021–Dec. 12, 2024. URL: https://www.ibm.com/think/topics/relational-databases (visited on 2025-01-05) (cit. on p. 376).

[632] L. Darrell Whitley. "The GENITOR Algorithm and Selection Pressure: Why Rank-Based Allocation of Reproductive Trials is Best". In: *3rd International Conference on Genetic Algorithms (ICGA'1989)*. June 1989, Fairfax, VA, USA: George Mason University. Ed. by J. David Schaffer. Burlington, MA, USA/San Mateo, CA, USA: Morgan Kaufmann Publishers, 1989, pp. 116–123. ISBN: 978-1-55860-066-9. URL: https://www.researchgate.net/publication/2527551 (visited on 2025-08-08) (cit. on pp. 371, 377).

[633] James A. Whittaker. "What Is Software Testing? And Why Is It So Hard?" *IEEE Software* 17(1):70–79, Jan.–Feb. 2000. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE). ISSN: 0740-7459. doi:10.1109/52.819971. Practice Tutorial (cit. on p. 170).

[634] Ulf Michael "Monty" Widenius, David Axmark, and Uppsala, Sweden: MySQL AB. *MySQL Reference Manual – Documentation from the Source*. Sebastopol, CA, USA: O'Reilly Media, Inc., July 9, 2002. ISBN: 978-0-596-00265-7 (cit. on p. 373).

[635] James M. Wilson. "Gantt Charts: A Centenary Appreciation". *European Journal of Operational Research* 149(2):430–437, Sept. 2003. Amsterdam, The Netherlands: Elsevier B.V. ISSN: 0377-2217. doi:10.1016/S0377-2217(02)00769-5 (cit. on p. 368).

[636] Kevin Wilson. *Python Made Easy*. Birmingham, England, UK: Packt Publishing Ltd, Aug. 2024. ISBN: 978-1-83664-615-0 (cit. on pp. 4, 11, 303, 362, 367, 375).

[637] Marianne Winslett and Vanessa Braganholo. "Richard Hipp Speaks Out on SQLite". *ACM SIGMOD Record* 48(2):39–46, June 2019. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0163-5808. doi:10.1145/3377330.3377338 (cit. on pp. 177, 378).

[638] Collin Winter and Tony Lownds. *Function Annotations*. Python Enhancement Proposal (PEP). Beaverton, OR, USA: Python Software Foundation (PSF), Dec. 2, 2006. URL: https://peps.python.org/pep-3107 (visited on 2024-12-12) (cit. on pp. 162, 357).

[639] "With Statement Context Managers". In: *Python 3 Documentation*. *The Python Language Reference*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. Chap. 3.3.9. URL: https://docs.python.org/3/reference/datamodel.html#with-statement-context-managers (visited on 2024-12-15) (cit. on pp. 208, 318).

[640] L. Wong and Lai-Soon Lee. "Heuristic Placement Routines for Two-Dimensional Bin Packing Problem". 5(4):334–341, Dec. 2009. doi:/10.3844/jmssp.2009.334.341. URL: https://www.thescipub.com/pdf/jmssp.2009.334.341 (visited on 2026-10-02). A heuristic for the 2BP|R|F and the 2BP|O|F (cit. on p. 393).

[641] CAO Xiang (曹翔), Zhize Wu (吴志泽), Daan van den Berg, and Thomas Weise (汤卫思). "Randomized Local Search vs. NSGA-II vs. Frequency Fitness Assignment on The Traveling Tournament Problem". In: *16th International Joint Conference on Computational Intelligence (IJCCI'24)*. Nov. 20–22, 2024, Porto, Portugal. Ed. by Francesco Marcelloni, Kurosh Madani, Niki van Stein, and Joaquim Filipe. Porto, Portugal: SciTePress: Science and Technology Publications, Lda, 2024, pp. 38–49. ISSN: 2184-3236. ISBN: 978-989-758-721-4. doi:10.5220/0012891500003837 (cit. on pp. 368, 373, 379).

[642] Zhou Xu and Chung-Yee Lee. "New Lower Bound and Exact Method for the Continuous Berth Allocation Problem". *Operations Research* 66(3):ii–iv, 597–892, May–June 2018. Catonsville, MD, USA: The Institute for Operations Research and the Management Sciences (INFORMS). ISSN: 0030-364X. doi:10.1287/opre.2017.1687. URL: https://ira.lib.polyu.edu.hk/bitstream/10397/78451/1/Xu_New_Lower_Bound.pdf (visited on 2026-10-01) (cit. on p. 365).

[643] Takeshi Yamada and Ryohei Nakano. "A Genetic Algorithm Applicable to Large-Scale Job-Shop Instances". In: *Parallel Problem Solving from Nature 2 (PPSN-II)*. Sept. 28–30, 1992, Brussels, Belgium. Ed. by Reinhard Männer and Bernard Manderick. Amsterdam, The Netherlands: Elsevier B.V., 1992, pp. 281–290. URL: https://www.researchgate.net/publication/220701684 (visited on 2026-09-30) (cit. on p. 370).

[644] Martin Yanev. *PyCharm Productivity and Debugging Techniques*. Birmingham, England, UK: Packt Publishing Ltd, Oct. 2022. ISBN: 978-1-83763-244-2 (cit. on pp. 9, 11, 375).

[645] Kinza Yasar and Craig S. Mullins. *Definition: Database Management System (DBMS)*. Newton, MA, USA: TechTarget, Inc., June 2024. URL: https://www.techtarget.com/searchdatamanagement/definition/database-management-system (visited on 2025-01-11) (cit. on p. 367).

[646] Jeffrey Yasskin. *A Type Hierarchy for Numbers*. Python Enhancement Proposal (PEP) 3141. Beaverton, OR, USA: Python Software Foundation (PSF), Apr. 25–Aug. 2, 2007. URL: https://peps.python.org/pep-3141 (visited on 2025-07-28) (cit. on pp. 41, 323, 324).

[647] Ka-Ping Yee and Guido van Rossum. *Iterators*. Python Enhancement Proposal (PEP) 234. Beaverton, OR, USA: Python Software Foundation (PSF), Jan. 30–Apr. 30, 2001. URL: https://peps.python.org/pep-0234 (visited on 2025-02-02) (cit. on pp. 216, 217, 219).

[648] François Yergeau. *UTF-8, A Transformation Format of ISO 10646*. Request for Comments (RFC) 3629. Wilmington, DE, USA: Internet Engineering Task Force (IETF), Nov. 2003. URL: https://www.ietf.org/rfc/rfc3629.txt (visited on 2025-02-05). See Unicode and [285] (cit. on pp. 206, 380).

[649] Wenqi Ying (应雯棋), ed. *Commemoration of Ancient Chinese Mathematical Master Liu Hui for his Timeless Influence on Mathematics and Civilizational Exchange*. Vol. 48 (Special Issue) of CAST Newsletter. China, Beijing (中国北京市): 中国科学技术协会 (China Association for Science and Technology, CAST), Nov. 2024. URL: https://english.cast.org.cn/cms_files/filemanager/1941250207/attach/202412/8f23655a82364d19ad7874eb37b23035.pdf (visited on 2025-08-24). Proofreader: Yumeng Wei (魏雨萌), Designer: Shan Zhang (张珊) (cit. on pp. 88, 363).

[650] Tatu Ylonen and Chris Lonvick. *The Secure Shell (SSH) Transport Layer Protocol*. Request for Comments (RFC) 4253. Wilmington, DE, USA: Internet Engineering Task Force (IETF), Jan. 2006. URL: https://www.ietf.org/rfc/rfc4253.txt (visited on 2025-02-05) (cit. on p. 378).

[651] Moshe Zadka and Guido van Rossum. *Changing the Division Operator*. Python Enhancement Proposal (PEP) 238. Beaverton, OR, USA: Python Software Foundation (PSF), Mar. 11–July 27, 2001. URL: https://peps.python.org/pep-0238 (visited on 2025-07-28) (cit. on pp. 33, 55, 56, 102).

[652] Pavlo V. Zahorodko and Pavlo V. Merzlykin. "An Approach for Processing and Document Flow Automation for Microsoft Word and LibreOffice Writer File Formats". In: *4th Workshop for Young Scientists in Computer Science & Software Engineering (CS&SE@SW'2021)*. Dec. 18, 2021, Virtual Event and Kryvyi Rih, Ukraine. Ed. by Arnold E. Kiv, Serhiy O. Semerikov, Vladimir N. Soloviev, and Andrii M. Striuk. Vol. 3077 of CEUR Workshop Proceedings (CEUR-WS.org). Aachen, Nordrhein-Westfalen, Germany: CEUR-WS Team, Rheinisch-Westfälische Technische Hochschule (RWTH) Aachen, 2022, pp. 66–82. ISSN: 1613-0073. URL: https://ceur-ws.org/Vol-3077/paper12.pdf (visited on 2025-10-04) (cit. on pp. 371, 373).

[653] Giorgio Zarrelli. *Mastering Bash*. Birmingham, England, UK: Packt Publishing Ltd, June 2017. ISBN: 978-1-78439-687-9 (cit. on pp. 365, 439).

[654] Jiazheng Zeng (曾嘉政), Thomas Weise (汤卫思), Zhize Wu (吴志泽), and Markus Wagner. "When Frequency Fitness Assignment Fails: Trapped States in Frequency-Guided Local Search". In: *28th Genetic and Evolutionary Computation Conference (GECCO'2026)*. July 13–17, 2026, San José, Costa Rica. Ed. by Leonardo Trujillo and Ting Hu. New York, NY, USA: Association for Computing Machinery (ACM), 2026, pp. 233–242. ISBN: 979-8-4007-2487-9. doi:10.1145/3795095.3805098. Explores $(1+1)$ EA, $(1+1)$ FEA, RLS, and FRLS as well as hybrid algorithms on a variety of discrete benchmarks such as OneMax, LeadingOnes, LinHarm, Binary Integer (BinInt), TwoMax, Trap, Jump, Plateau, Ising Model, N-Queens and also applies them to the search for Low-Autocorrelation Binary Sequences (LABS). (Cit. on pp. 363, 364, 368, 370, 371, 374–376, 379, 380).

[655] Chaoyong Zhang, Xinyu Shao, Yunqing Rao, and Haobo Qiu. "Some New Results on Tabu Search Algorithm Applied to the Job-Shop Scheduling Problem". In: *Tabu Search*. Ed. by Wassim Jaziri. London, England, UK: IntechOpen Limited, Aug. 1, 2008. Chap. 8. ISBN: 978-3-902613-34-9. doi:10.5772/5593. URL: https://www.intechopen.com/chapters/4596 (visited on 2026-09-30) (cit. on pp. 370, 379).

[656] Jingxuan Zhang (张静宣), Weiqin Zou (邹卫琴), and Zhiqiu Huang (黄志球). "An Empirical Study on the Usage and Evolution of Identifier Styles in Practice". In: *28th Asia-Pacific Software Engineering Conferenc (APSEC'2021)*. Dec. 6–9, 2021, Taipei, Taiwan, China; Virtual Event. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE), 2022, pp. 171–180. ISSN: 2640-0715. ISBN: 978-1-6654-3785-1. doi:10.1109/APSEC53868.2021.00025. URL: https://weiqin-zou.github.io/papers/APSEC21.pdf (visited on 2026-10-10). A study based on Java projects on GitHub (cit. on pp. 86, 161, 251, 355, 357, 358).

[657] Rui Zhao (赵睿), Tianyu Liang (梁天宇), Zhize Wu (吴志泽), Daan van den Berg, Matthias Thürer, and Thomas Weise (汤卫思). "Randomized Local Search on the 2D Rectangular Bin Packing Problem with Item Rotation". In: *26th Genetic and Evolutionary Computation Conference (GECCO'2024)*. July 14–18, 2024, Melbourne, VIC, Australia. Ed. by Xiaodong Li and Julia

Handl. New York, NY, USA: Association for Computing Machinery (ACM), 2024, pp. 235–238. ISBN: 979-8-4007-0494-9. doi:10.1145/3638530.3654139 (cit. on pp. 364, 369, 373).

[658] Rui Zhao (赵睿), Zhize Wu (吴志泽), Daan van den Berg, Matthias Thürer, Tianyu Liang (梁天宇), Ming Tan (檀明), and Thomas Weise (汤卫思). "Randomized Local Search for Two-Dimensional Bin Packing and a Negative Result for Frequency Fitness Assignment". In: *16th International Joint Conference on Computational Intelligence (IJCCI'24)*. Nov. 20–22, 2024, Porto, Portugal. Ed. by Francesco Marcelloni, Kurosh Madani, Niki van Stein, and Joaquim Filipe. Porto, Portugal: SciTePress: Science and Technology Publications, Lda, 2024, pp. 15–26. ISSN: 2184-3236. ISBN: 978-989-758-721-4. doi:10.5220/0012888500003837 (cit. on pp. 364, 366, 369, 373).

[659] Nicola Abdo Ziadeh, Michael B. Rowton, A. Geoffrey Woodhead, Wolfgang Helck, Jean L.A. Filliozat, Hiroyuki Momo, Eric Thompson, E.J. Wiesenberg, and Shih-ch'ang Wu. "Chronology – Christian History, Dates, Events". In: *Encyclopaedia Britannica*. Ed. by The Editors of Encyclopaedia Britannica. Chicago, IL, USA: Encyclopædia Britannica, Inc., July 26, 1999–Mar. 20, 2024. URL: https://www.britannica.com/topic/chronology/Christian (visited on 2025-08-27) (cit. on pp. 365, 366).

[660] Jelle "JelleZijlstra" Zijlstra, Mehdi "hmc-cs-mdrissi" Drissi, Alex "AlexWaygood" Waygood, Daniele "dvarrazzo" Varrazzo, Shantanu "hauntsaninja", François-Michel "FinchPowers" L'Heureux, and Rupesh "rupeshs" Sreeraman. *Issue #12554: Support PEP 675 (`LiteralString`)*. San Francisco, CA, USA: GitHub Inc, Apr. 9, 2022–Nov. 29, 2024. URL: https://github.com/python/mypy/issues/12554 (visited on 2025-03-05) (cit. on pp. 108, 378).

[661] Dmitry Zinoviev. *Discrete Event Simulation: It's Easy with SimPy!* arXiv.org: Computing Research Repository (CoRR) abs/2405.01562. Ithaca, NY, USA: Cornell Universiy Library, Apr. 3, 2024. doi:10.48550/ARXIV.2405.01562. URL: https://arxiv.org/abs/2405.01562 (visited on 2024-06-27). arXiv:2405.01562v1 [cs.MS] 3 Apr 2024 (cit. on pp. 3, 377).

[662] *信息交换用汉字编码字符集 (Code of Chinese Graphic Character Set for Information Interchange – Primary Set)*. 中华人民共和国国家标准 (National Standard of the People's Republic of China, GB) GB/T 2312-1980. China, Beijing (中国北京市): 中国国家标准化管理委员会 (Standardization Administration of the People's Republic of China, SAC), 1980–May 1, 1981. URL: https://openstd.samr.gov.cn/bzgk/gb/newGbInfo?hcno=5664A728BD9D523DE3B99BC37AC7A2CC (visited on 2024-07-26) (cit. on p. 80).

[663] *信息技术中文编码字符集 (Information Technology – Chinese Coded Character Set)*. 中华人民共和国国家标准 (National Standard of the People's Republic of China, GB) GB 18030-2022. China, Beijing (中国北京市): 中华人民共和国工业和信息化部 (Ministry of Industry and Information Technology, MIIT): 国家市场监督管理总局 (State Administration for Market Regulation) and 中国国家标准化管理委员会 (Standardization Administration of the People's Republic of China, SAC), July 19, 2022–Aug. 1, 2023. URL: https://openstd.samr.gov.cn/bzgk/gb/newGbInfo?hcno=A1931A578FE14957104988029B0833D3 (visited on 2024-07-26) (cit. on p. 80).

[664] *国务院关于公布汉字简化方案的决议(1956年) (Resolution of the State Council on Publishing the Simplified Chinese Character Scheme (1956))*. China, Beijing (中国北京市): 中华人民共和国国务院 (The State Council of the People's Republic of China), Jan. 28, 1956. URL: http://www.moe.gov.cn/jyb_xwfb/xw_fbh/moe_2128/moe_2326/moe_1144/tnull_14350.html (visited on 2025-05-07) (cit. on pp. 51, 52, 80).

[665] *汉字内码扩展规范 (Chinese Internal Code Specification, National Standard Extended GBK)*. 中华人民共和国国家标准 (National Standard of the People's Republic of China, GB) 1995 229. China, Beijing (中国北京市): 中华人民共和国全国信息技术标准化技术委员会 (National Information Technology Standardization Technical Committee of the People's Republic of China), 国家技术监督局 (National Bureau of Technical Supervision), and 中华人民共和国电子工业部 (Ministry of Electronics Industry of the People's Republic of China), Dec. 15, 1995 (cit. on p. 80).

Scripts

Here we provide some scripts that are used within this book. These scripts are written for the Bash shell, which is the default interpreter running in Ubuntu Linux terminals. Therefore, they will not work under Microsoft Windows or other operating systems. Now, our book focuses on Python programming, so Bash shell scripts are not in the center of our attention. We here cannot explain how Bash scripts

Listing 16.1: A Bash script for executing Mypy, which prints the command line and the exit code; see Useful Tool 4. (src)

```
#!/bin/bash -

# This is our internal mypy execution script.
# It prints the mypy command and the results of mypy.
# It always exits with an exit code 0, even if mypy fails.
# Argument $1: the folder to run it in
# Argument $2: the file(s) to scan

# We enforce strict error handling, i.e., fail on any unexpected error.
set -o pipefail  # trace errors through pipes
set -o errtrace  # trace errors through commands and functions
set -o nounset   # exit if encountering an uninitialized variable
set -o errexit   # exit if any statement returns a non-0 return value

# Check if mypy is installed. If yes, get its version. If no, install.
infos="$(python3 -m pip show mypy 2>/dev/null || true)"
if [ -z "$infos" ]; then
  # mypy is not installed, so we install it now.
  # We do this silently, without printing any information...
  python3 -m pip install --require-virtualenv mypy 1>/dev/null 2>&1
  infos="$(python3 -m pip show mypy 2>/dev/null)"
fi
# We now extract the version of mypy from the information string.
version="$(grep Version: <<< "$infos")"
version="$(sed -n 's/.*Version:\s*\([.0-9]*\)/\1/p' <<< "$version")"

# Construct the mypy command.
command="mypy $2 --no-strict-optional --check-untyped-defs"
echo "\$ $command"  # We print the command line which will be executed.
cd "$1"  # We enter the folder inside of which we should execute mypy.

# Switch of "exit-on-error", run mypy, and afterwards switch it back on.
set +o errexit  # Turn off exit-on-error.
$command 2>&1 | sed '/^[[:space:]]*$/d'  # Run mypy, clean output.
exitCode="$?"  # Store exit code of program in variable exitCode.
set -o errexit  # Turn exit-on-error back on.

# Convert exit code to success or failure string.
[ "$exitCode" -eq 0 ] && exitCodeStr="succeeded" || exitCodeStr="failed"

# Finally, we print the result string.
echo "# mypy $version $exitCodeStr with exit code $exitCode."
```

work or what their syntax is. There exist plenty of books and resources on this interesting topic, such as [73, 410, 653] or https://www.gnu.org/software/bash.

If we would include the scripts in the places where we use them in this book, then this would lead to confusion or tangents in the text which would mess up the flow of the chapters. Nevertheless, the book would be incomplete if these scripts were not provided at all. So we put them here, at the end of the book, where they do not hurt anyone and where the interested reader may check them out.

In the scripts, we install tools if they are not yet installed. We then apply the tools to whatever the parameters of the scripts state. We collect the stderr and stdout of the tools. Often, we remove empty lines and otherwise useless characters by piping this output through `sed` (just to save space). This sanitized output appears then in the code listings.

Now, in a real environment, a tool can either succeed or fail. A unit test executed with pytest will fail and exit with a non-zero exit code if the test, well, fails. A static code analysis tool like Ruff will fail with a non-zero exit code if it discovers any issue with the code. However, our scripts invoking these tools will *not* fail in such cases. They will collect the exit code of the tool they are invoking and print it. Then they will exit with exit code 0. This is necessary for our book building process, which invokes these scripts to construct the outputs of the examples. If any of them would fail with non-zero exit code, the book building would fail as well. However, to illustrate that the tools are useful, we must apply them to cases where they would fail. So for our book, it is necessary that the scripts just print the exit codes while still returning successfully. For a real productive environment, this is usually not what we want: We apply the tools to source code precisely to get them to fail on error, because if nothing fails, we know that everything seems to be OK. In such a practical environment, you would thus not want to use *our* scripts, but you could use the same comments that we use. Anyway, here is the list of tool scripts.

Listing 16.1 is a Bash script which first checks whether Mypy is installed. If Mypy is not installed, it silently installs it. The script then composes the command for applying Mypy to the target file(s) in the target directory. It then prints the command with a prepended `$`. Then, it executes it. Mypy will print its comments and messages to the standard output. Finally, the script shows the Mypy version and exit code in a brief success or failure message (with a prepended `#`). Listings 4.22 and 4.23 are example outputs of this script.

Listing 16.2 works basically the same way, just for Ruff. However, it sets a lot more parameters to Ruff. See, Ruff offers many more configuration options and many more different things that it can check for, compared to Mypy. In it's present form, it does not check for type errors, as far as I know, though. Either way, it is harder to balance the strictness of the tool and there are even some rules which are sometimes mutually exclusive. Hence, we compose a more complex command. Apart from that, this script works pretty much the same as Listing 16.1. Listings 5.10 and 5.14 are examples of the output of this script.

Listing 16.3 offers exactly the same functionality for Pylint. It checks if this tool is installed and installs it if not. It then applies Pylint to a the selected set of files, using a reasonable default configuration. Listing 7.17 is an example of the output of this linter.

Listing 16.4 is similarly structured, but instead of performing *static* code analysis, it executes unit test cases. The directory and list of Python files with the test cases are provided as command line arguments. This script checks if pytest and its plugin `pytest-timeout` are installed and, if not, installs them. It then executes the tests with a fixed ten second timeout that we use in this book. Of course, in practical scenarios, you would use a larger timeout, but within the context of this book, ten seconds are enough. Either way, the script executes the test cases, prints the results as well as potential errors. Notice that we select some options to make the output less verbose, because for this book, we do want listings that are not too long. In practical scenarios, you may use different options. Either way, the script also prints the command that was executed at the beginning and the pytest version used and the exit code of the process at the end of the output. Listings 8.9, 8.12 and 8.15 are examples of the output of pytest. Listing 16.5 uses pytest as well, but this time executes doctests. An example can be found in Listing 10.14.

Listing 16.2: A Bash script for executing Ruff, which prints the command line and the exit code; see Useful Tool 5. (src)

```
#!/bin/bash -

# This is our internal ruff execution script.
# It prints the ruff command and the results of ruff.
# It always exits with an exit code 0, even if ruff fails.
# Argument $1: the folder to run it in
# Argument $2: the file(s) to scan

# We enforce strict error handling, i.e., fail on any unexpected error.
set -o pipefail  # trace errors through pipes
set -o errtrace  # trace errors through commands and functions
set -o nounset   # exit if encountering an uninitialized variable
set -o errexit   # exit if any statement returns a non-0 return value

# Check if ruff is installed. If yes, get its version. If no, install.
infos="$(python3 -m pip show ruff 2>/dev/null || true)"
if [ -z "$infos" ]; then
  # ruff is not installed, so we install it now.
  # We do this silently, without printing any information...
  python3 -m pip install --require-virtualenv ruff 1>/dev/null 2>&1
  infos="$(python3 -m pip show ruff 2>/dev/null)"
fi
# We now extract the version of ruff from the information string.
version="$(grep Version: <<< "$infos")"
version="$(sed -n 's/.*Version:\s*\([.0-9]*\)/\1/p' <<< "$version")"

# Construct the ruff command.
command="ruff check --target-version py312 --select=A,AIR,ANN,ASYNC,B,BLE,C
    ↪ ,C4,COM,D,DJ,DTZ,E,ERA,EXE,F,FA,FIX,FLY,FURB,G,I,ICN,INP,ISC,INT,LOG,
    ↪ N,NPY,PERF,PIE,PLC,PLE,PLR,PLW,PT,PYI,Q,RET,RSE,RUF,S,SIM,T,T10,TD,
    ↪ TID,TRY,UP,W,YTT --ignore=A005,ANN001,ANN002,ANN003,ANN204,ANN401,
    ↪ B008,B009,B010,C901,D203,D208,D212,D401,D407,D413,INP001,N801,PLC2801
    ↪ ,PLR0904,PLR0911,PLR0912,PLR0913,PLR0914,PLR0915,PLR0916,PLR0917,
    ↪ PLR1702,PLR2004,PLR6301,PT011,PT012,PT013,PYI041,RUF100,S,T201,TRY003
    ↪ ,UP035,W --line-length 79 $2"
echo "\$ $command"  # We print the command line which will be executed.
cd "$1"  # We enter the folder inside of which we should execute ruff.

# Switch of "exit-on-error", run ruff, and afterwards switch it back on.
set +o errexit  # Turn off exit-on-error.
$command 2>&1 | sed '/^[[:space:]]*[\|]*[[:space:]]*$/d' # Ruff + clean
exitCode="$?"  # Store exit code of program in variable exitCode.
set -o errexit  # Turn exit-on-error back on.

# Convert exit code to success or failure string.
[ "$exitCode" -eq 0 ] && exitCodeStr="succeeded" || exitCodeStr="failed"

# Finally, we print the result string.
echo "# ruff $version $exitCodeStr with exit code $exitCode."
```

Listing 16.3: A Bash script for executing Pylint, which prints the command line and the exit code; see Useful Tool 6. (src)

```
#!/bin/bash -

# This is our internal pylint execution script.
# It prints the pylint command and the results of pylint.
# It always exits with an exit code 0, even if pylint fails.
# Argument $1: the folder to run it in
# Argument $2: the file(s) to scan

# We enforce strict error handling, i.e., fail on any unexpected error.
set -o pipefail  # trace errors through pipes
set -o errtrace  # trace errors through commands and functions
set -o nounset   # exit if encountering an uninitialized variable
set -o errexit   # exit if any statement returns a non-0 return value

# Is pylint is installed? If yes, get its version. If no, install it.
infos="$(python3 -m pip show pylint 2>/dev/null || true)"
if [ -z "$infos" ]; then
  # pylint is not installed, so we install it now.
  # We do this silently, without printing any information...
  python3 -m pip install --require-virtualenv pylint 1>/dev/null 2>&1
  infos="$(python3 -m pip show pylint 2>/dev/null)"
fi
# We now extract the version of pylint from the information string.
version="$(grep Version: <<< "$infos")"
version="$(sed -n 's/.*Version:\s*\([.0-9]*\)/\1/p' <<< "$version")"

# Construct the pylint command.
command="pylint $2 --disable=C0103,C0302,C0325,R0801,R0901,R0902,R0903,
    ↪ R0911,R0912,R0913,R0914,R0915,R1702,R1728,W0212,W0238,W0703"
echo "\$ $command"  # We print the command line which will be executed.
cd "$1"  # We enter the folder inside of which we should execute pylint.

# Switch of "exit-on-error", run pylint, and afterwards switch it back on.
set +o errexit  # Turn off exit-on-error.
$command 2>&1 | sed '/^[[:space:]]*$/d'  # Run pylint, clean output.
exitCode="$?"  # Store exit code of program in variable exitCode.
set -o errexit  # Turn exit-on-error back on.

# Convert exit code to success or failure string.
[ "$exitCode" -eq 0 ] && exitCodeStr="succeeded" || exitCodeStr="failed"

# Finally, we print the result string.
echo "# pylint $version $exitCodeStr with exit code $exitCode."
```

Listing 16.4: A Bash script for executing test cases with pytest, which prints the command line and the exit code; see Useful Tool 7. (src)

```
#!/bin/bash -

# This is our internal pytest execution script.
# It prints the pytest command and the results of pytest.
# It always exits with an exit code 0, even if pytest fails.
# Argument $1: the folder to run it in
# Argument $2: the file(s) to scan

# We enforce strict error handling, i.e., fail on any unexpected error.
set -o pipefail  # trace errors through pipes
set -o errtrace  # trace errors through commands and functions
set -o nounset   # exit if encountering an uninitialized variable
set -o errexit   # exit if any statement returns a non-0 return value

# Check if pytest and all required plugins are installed.
versions=""  # This variable will receive all tool versions.
for pack in "pytest" "pytest-timeout"; do
  infos="$(python3 -m pip show "$pack" 2>/dev/null || true)"
  if [ -z "$infos" ]; then
    # pytest or the plugin is not installed, so we install it now.
    # We do this silently, without printing any information...
    python3 -m pip install --require-virtualenv "$pack" 1>/dev/null 2>&1
    infos="$(python3 -m pip show "$pack" 2>/dev/null)"
  fi

  # For each tool or plugin, we get the version separately.
  infos="$(grep Version: <<< "$infos")"
  infos="$(sed -n 's/.*Version:\s*\([.0-9]*\)/\1/p' <<< "$infos")"
  if [ -z "$versions" ]; then  # ... and we concatenate them
    versions="$pack $infos with"
  elif [[ "$versions" == *with ]]; then
    versions="$versions $pack $infos"
  else
    versions="$versions, $pack $infos"
  fi
done

# Construct the pytest command.
command="pytest --timeout=10 --no-header --tb=short $2"
echo "\$ $command"  # We print the command line which will be executed.
cd "$1"  # We enter the folder inside of which we should execute pytest.
export COLUMNS=73
# Switch of "exit-on-error", run pytest, and afterwards switch it back on.
set +o errexit  # Turn off exit-on-error.
$command 2>&1 | sed '/^[[:space:]]*$/d'  # Run pytest, clean output.
exitCode="$?"  # Store exit code of program in variable exitCode.
set -o errexit  # Turn exit-on-error back on.

# Convert exit code to success or failure string.
[ "$exitCode" -eq 0 ] && exitCodeStr="succeeded" || exitCodeStr="failed"

# Finally, we print the result string.
echo "# $versions $exitCodeStr with exit code $exitCode."
```

Listing 16.5: A Bash script for executing doctests with pytest, which prints the command line and the exit code; see Useful Tool 9. (src)

```
#!/bin/bash -

# This is our internal pytest-doctest execution script.
# It prints the pytest-doctest command and the results of pytest.
# It always exits with an exit code 0, even if pytest fails.
# Argument $1: the folder to run it in
# Argument $2: the file(s) to scan

# We enforce strict error handling, i.e., fail on any unexpected error.
set -o pipefail  # trace errors through pipes
set -o errtrace  # trace errors through commands and functions
set -o nounset   # exit if encountering an uninitialized variable
set -o errexit   # exit if any statement returns a non-0 return value

# Check if pytest and all required plugins are installed.
versions=""  # This variable will receive all tool versions.
for pack in "pytest" "pytest-timeout"; do
  infos="$(python3 -m pip show "$pack" 2>/dev/null || true)"
  if [ -z "$infos" ]; then
    # pytest or the plugin is not installed, so we install it now.
    # We do this silently, without printing any information...
    python3 -m pip install --require-virtualenv "$pack" 1>/dev/null 2>&1
    infos="$(python3 -m pip show "$pack" 2>/dev/null)"
  fi

  # For each tool or plugin, we get the version separately.
  infos="$(grep Version: <<< "$infos")"
  infos="$(sed -n 's/.*Version:\s*\([.0-9]*\)/\1/p' <<< "$infos")"
  if [ -z "$versions" ]; then  # ... and we concatenate them
    versions="$pack $infos with"
  elif [[ "$versions" == *with ]]; then
    versions="$versions $pack $infos"
  else
    versions="$versions, $pack $infos"
  fi
done

# Construct the pytest command for documentation testing.
command="pytest --timeout=10 --no-header --tb=short --doctest-modules $2"
echo "\$ $command"  # We print the command line which will be executed.
cd "$1"  # We enter the folder inside of which we should execute pytest.
export COLUMNS=73
# Switch of "exit-on-error", run pytest, and afterwards switch it back on.
set +o errexit  # Turn off exit-on-error.
$command 2>&1 | sed '/^[[:space:]]*$/d'  # Run doctests, clean output.
exitCode="$?"  # Store exit code of program in variable exitCode.
set -o errexit  # Turn exit-on-error back on.

# Convert exit code to success or failure string.
[ "$exitCode" -eq 0 ] && exitCodeStr="succeeded" || exitCodeStr="failed"

# Finally, we print the result string.
echo "# $versions $exitCodeStr with exit code $exitCode."
```