Programming with Python

Thomas Weise (汤卫思)

April 1, 2025

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 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

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

This book is freely available. You can download its newest version from https://thomasweise.github.io/programmingWithPython/. This version may change since this book is, well, work in progress. The book is released under the *Attribution-NonCommercial-ShareAlike 4.0 International license* (CC BY-NC-SA 4.0). You can freely share it. You can also copy text or figures, as long as you cite the book as the original source [313], e.g., by using the following BibTeX:

```
@book{programmingWithPython,
 author = {Thomas Weise},
 title = {Programming with Python},
 year = {2024--2025},
 publisher = {Institute of Applied Optimization,
              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).

Prof. Dr. Thomas Weise (汤卫思教授)
at the Institute of Applied Optimization (应用优化研究所, IAO)
of 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 3.21.3
Linux 6.8.0-1021-azure x86_64

python: 3.12.9
pytest: 8.3.5
pytest-timeout: 2.3.1
mypy: 1.15.0
ruff: 0.11.2
pylint: 3.3.6

latexgit_py: 0.8.27
latexgit_tex: 0.8.5
pycommons: 0.8.64
pdflatex: pdfTeX 3.141592653-2.6-1.40.27 (TeX Live 2025)
biber: 2.20
makeglossaries: 4.58 (2025-03-19)
ghostscript: 10.04.0 (2024-09-18)

date: 2025-04-01 06:15:34 +0000
```

Chapter 1

Introduction

Welcome to *Programming with Python*, an introduction into Python programming. Our book is just one of many books that teach this programming language [9, 132, 153, 170]. Our focus is on teaching good programming practices right from the start, support by many examples and tools.

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

Definition 2 (Programming) *Programming* is the activity or job of writing computer programs [225].

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 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 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 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 [302].

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 [76, 181]. Other studies even suggest that less than 20% of the working time spent with coding, maybe with another 15% of bug fixing [185]. 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. 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 [46]. We plot the number of pushes to GitHub over time for the most popular programming and web development languages in Figure 1.1. 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

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

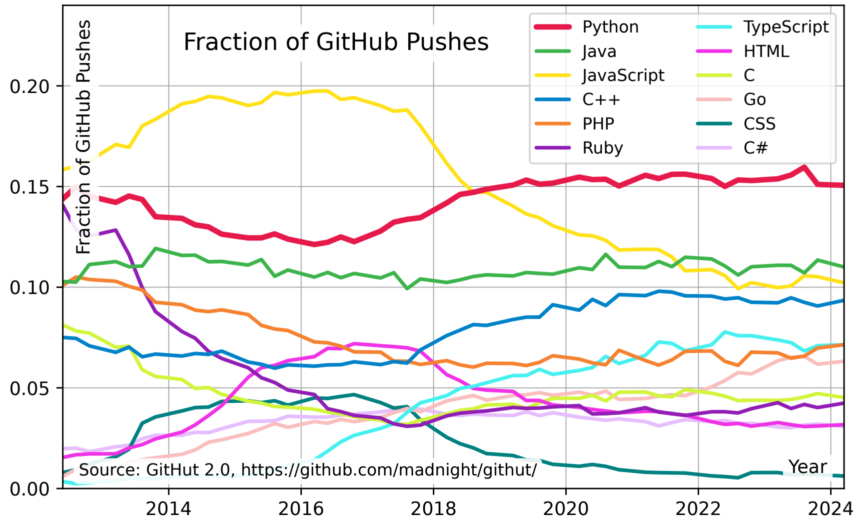


Figure 1.1: The twelve most popular programming languages chosen based on the GitHub pushes over the years. Source: [26].

major search engines, `Python` ranked one in January 2025 and was named the programming language of the year for 2024 [142].

> `Python` is everywhere nowadays, and it is the undisputed default language of choice in many fields.
>
> — *Paul Jansen [142], 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 [329], `Python` was the second most popular programming language, after JavaScript and HTML/CSS. In GitHub's Octoverse Report from October 2024 [106], `Python` is named the most popular programming language, ranking right before JavaScript.

Second, `Python` is intensely used [46] in the fields of Artificial Intelligence (AI) [242], Machine Learning (ML) [253], and Data Science (DS) [114] as well as optimization, which are among the most important areas of future technology. Indeed, the aforementioned Octoverse report [106] 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` [75, 119, 143, 197], `Pandas` [18, 175, 208], `Scikit-learn` [212, 230], `SciPy` [143, 307], `TensorFlow` [1, 166], `PyTorch` [209, 230], `Matplotlib` [133, 135, 143, 205], `SimPy` [334], and `moptipy` [314][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) [304], or support for web application development [2, 289]. 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 [111, 298]. 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 [329]. 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 1.2. 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.

[2] Yes, I list `moptipy` here, next to very well-known and widely-used frameworks, because I am its developer.
[3] at least during the first steps of learning

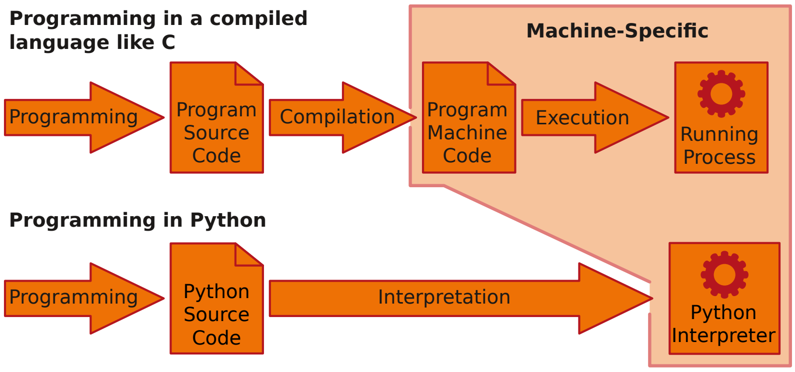


Figure 1.2: `Python` code is interpreted, which leads to an easier programming workflow compared to compiled programming languages like `C`.

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 looking 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 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 Systems (VCS) like Git. You want to do that convenient 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 [297, 326]. 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 [14, 120, 288] 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 [57, 122]. Either way, in the following, I will try to provide examples and instructions for both Ubuntu and the commercial Microsoft Windows [31] 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 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 https://docs.python.org/3/using [228],
- the Python Downloads at https://www.python.org/downloads, and
- the Python 3 Installation & Setup Guide at https://realpython.com/installing-python

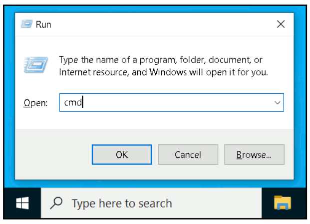
2.1.1 Python under Ubuntu Linux

Under Ubuntu Linux, Python 3 is already pre-installed. You can open a terminal [14] by pressing Ctrl + Alt + T, then type in `python3 --version`, hit ↲, and get the result illustrated in Figure 2.1:

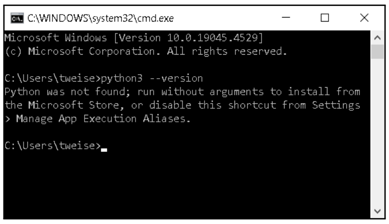
Figure 2.1: Under my `Ubuntu Linux 22.04` system, typing `python3 --version` in a terminal and hitting return yields version `3.10.12`.

2.1.2 Installing Python 3 under Microsoft Windows

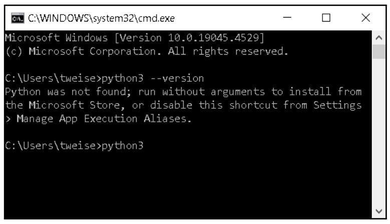
Example installation steps for `Python` on Microsoft `Windows` (version 10) are sketched in Figure 2.2. First, you would open a terminal using and press ⊞ + R, type in `cmd`, and hit ↲, as shown in Figure 2.2.1. If `Python` is installed, then typing `python3 --version` in the terminal and hitting ↲ would 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).0 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`.



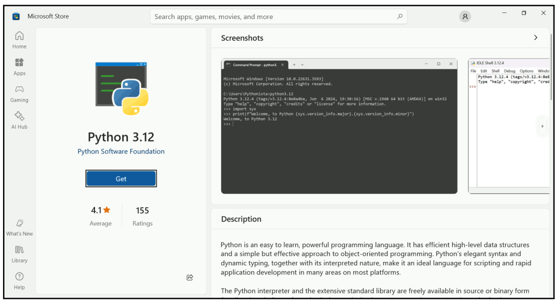
(2.2.1) Opening the terminal under Microsoft `Windows`: press ⊞ + R, type in `cmd`, and hit ↲.



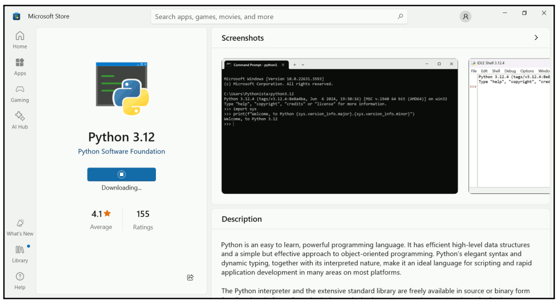
(2.2.2) Trying to get the `Python` version via `python3 --version`, but it is not installed.



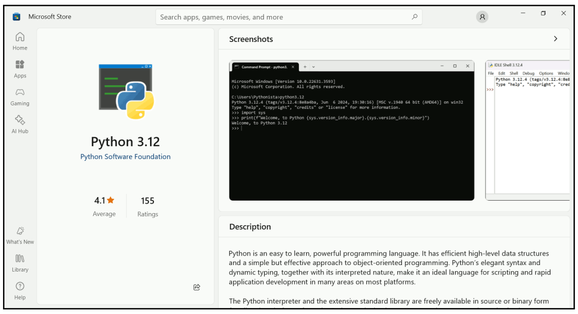
(2.2.3) Installing it by typing `python3` and hitting ↲.



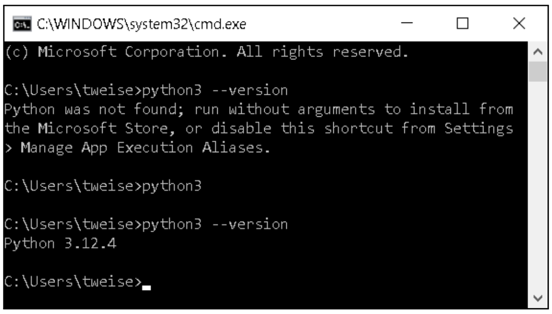
(2.2.4) The install screen, where we click Get.



(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.

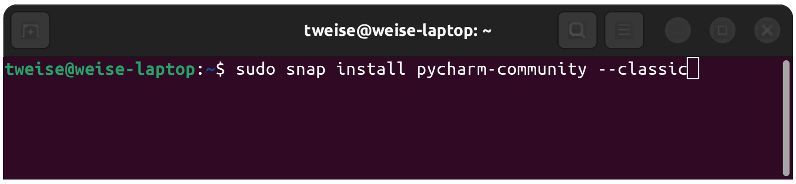
Figure 2.2: Cropped screenshots of the installation steps for `Python` on Microsoft `Windows`.

2.2 Installing PyCharm

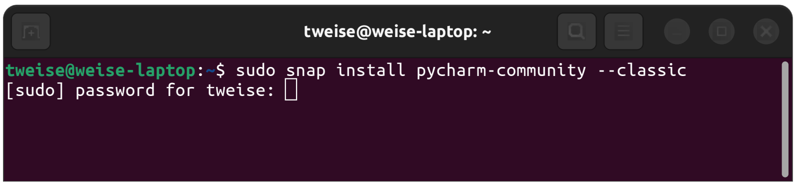
Just having a programming language and the corresponding interpreter on your system is not enough. Well, it is enough for just running `Python` programs. But it is not enough if you want to develop software efficiently. Are you going to write programs in a simple text editor like a caveperson? No, of 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`, whose community edition is freely available. The installation guide for `PyCharm` can be found at `https://www.jetbrains.com/help/pycharm/installation-guide.html`.

2.2.1 Installing PyCharm under Ubuntu Linux

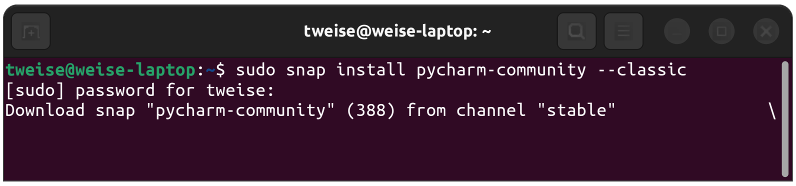
`PyCharm` is available as Snap package under `Ubuntu` `Linux` [222, 265]. The installation process is very easy and follows the steps illustrated in Figure 2.3. First, you open a terminal by pressing Ctrl + Alt + T. Then, enter the command `sudo snap install pycharm-community --classic` and hit return (Figure 2.3.1). 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 2.3.2. Then, the installation process basically runs automatically. Once it has completed (see Figure 2.3.5), you can press the ⊞ key and type `pycharm` in the launcher window to find



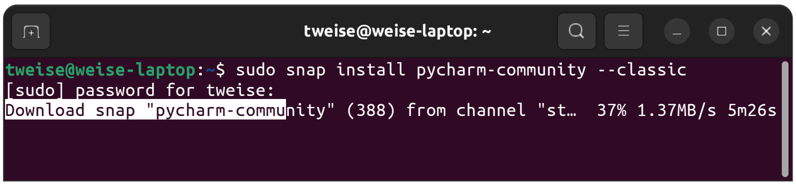
(2.3.1) Installing PyCharm using the `snap install` command in a terminal opened with Ctrl+Alt+T.



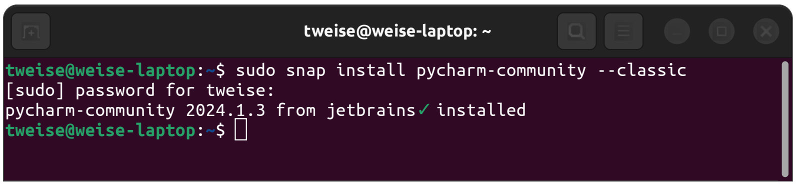
(2.3.2) This command requires the super user password, which we type in and then press Enter.



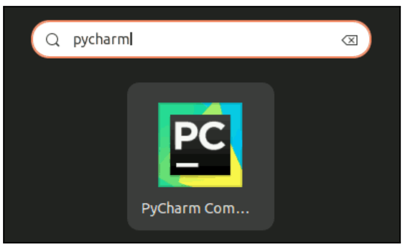
(2.3.3) The installation begins.



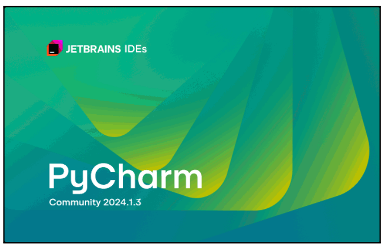
(2.3.4) The software package is downloaded.



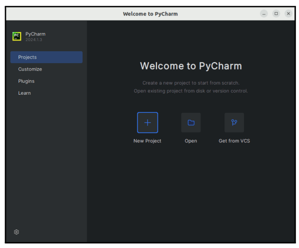
(2.3.5) The package is installed.



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



(2.3.7) The PyCharm welcome screen appears.



(2.3.8) PyCharm has been started.

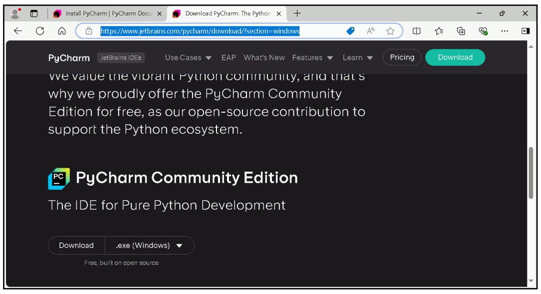
Figure 2.3: The installation steps of PyCharm under Ubuntu linux.

PyCharm (Figure 2.3.6). A double-click will open PyCharm.

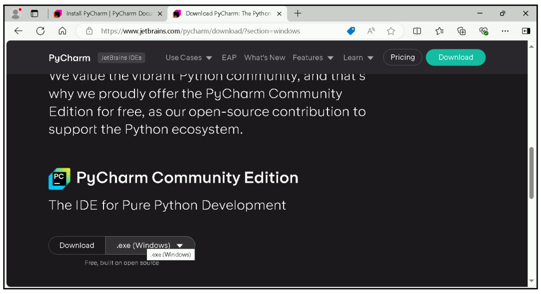
In the installation instructions for Microsoft Windows, now a user agreement (Figure 2.4.16) and data upload statement (Figure 2.4.17) need to be performed. Since I already had PyCharm installed previously and probably already agreed/disagreed to them, respectively, these windows did not open in my current 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 2.3.8).

2.2.2 Installing PyCharm under Microsoft Windows

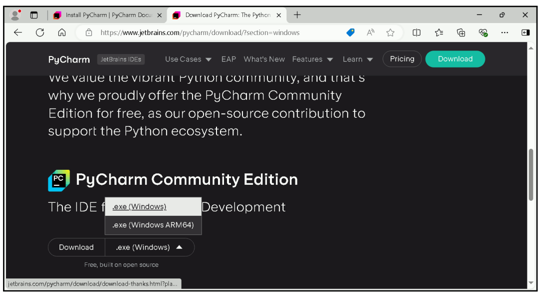
The process of installing PyCharm under Microsoft Windows is illustrated in Figure 2.4. You first need to download the PyCharm Community Edition installation executable from https://www.jetbrains.com/pycharm/download. Make sure to download the Community Edition and nothing else, as shown in Figures 2.4.1 to 2.4.3. Once the installer is downloaded, you start it and confirm that you wish to install PyCharm, as illustrated in Figures 2.4.5 to 2.4.8. As Figures 2.4.9 to 2.4.11 show, the installation setup process is more or less automated, we just need to click Next here and there and finally click Install (Figure 2.4.12). After the installation completes, we run PyCharm for the first time. Now we need to agree to the user agreement (Figure 2.4.16) and should probably choose that we do not with to send any information about our PyCharm usage out (Figure 2.4.17). Finally, as sketched in Figure 2.4.18, we have a running and ready PyCharm IDE.



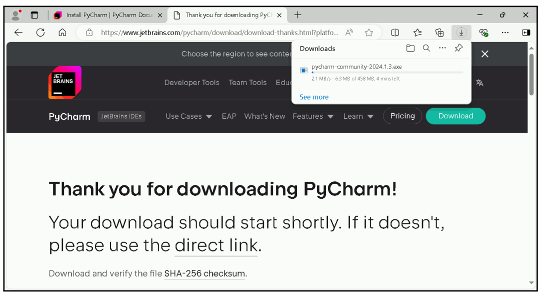
(2.4.1) The download website https://www.jetbrains.com/pycharm/download for PyCharm.



(2.4.2) Downloading the PyCharm Community Edition.

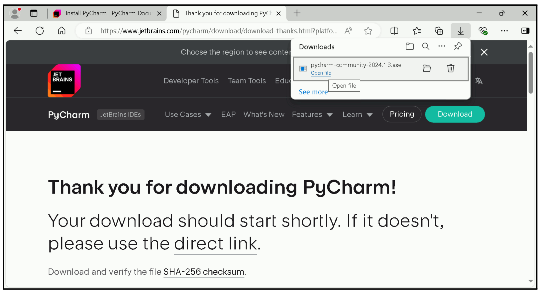


(2.4.3) Selecting the normal Microsoft Windows installer for the PyCharm Community Edition.

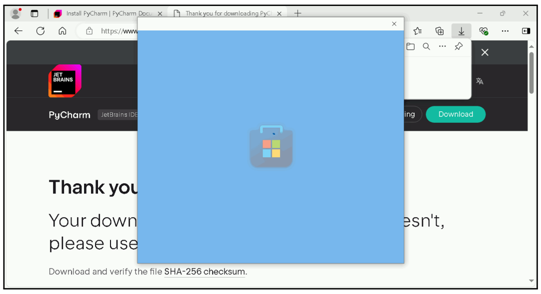


(2.4.4) The download is starting.

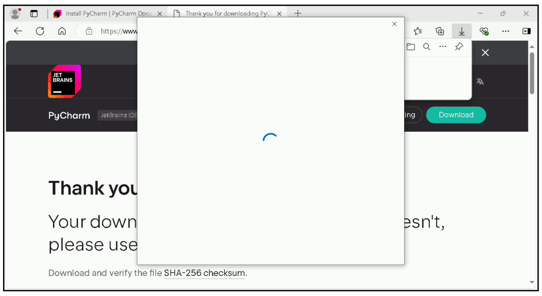
Figure 2.4: The installation steps of PyCharm under Microsoft Windows.



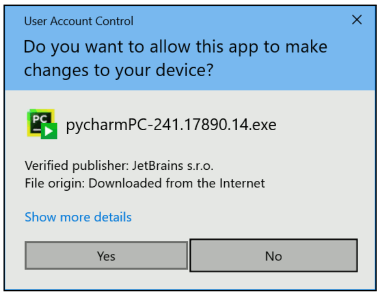
(2.4.5) The download is completed. We click Open file.



(2.4.6) The installer is starting.

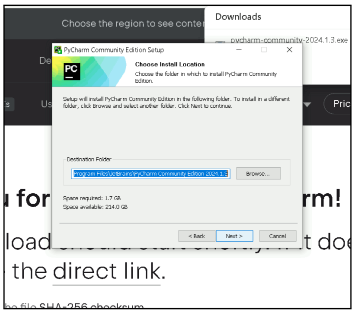


(2.4.7) The installer is starting.

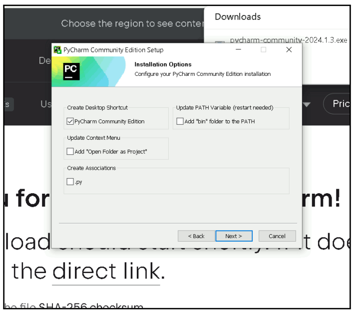


(2.4.8) We do want to install, so click Yes.

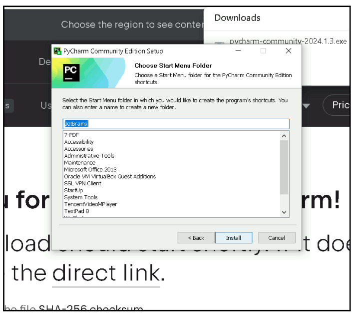
(2.4.9) The welcome screen of the installer. We click Next.



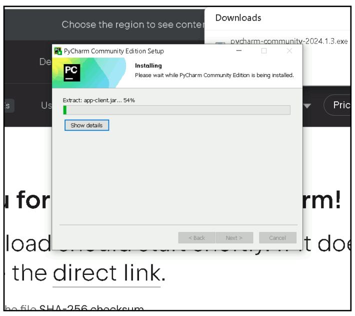
(2.4.10) The installation folder selection. We click Next.



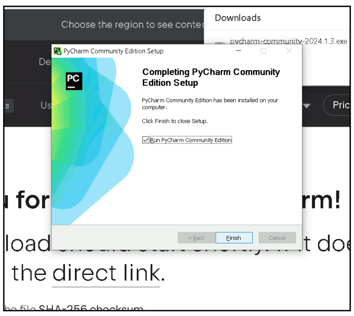
(2.4.11) The installation options. We can click Next.



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

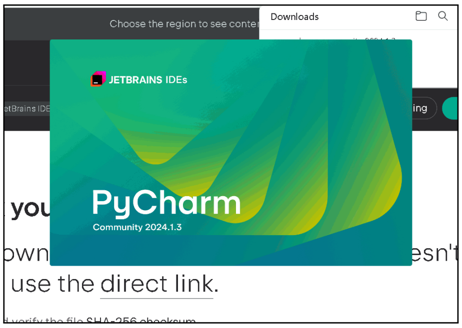


(2.4.13) The install process starts.

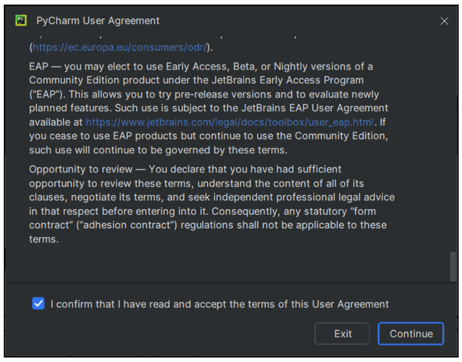


(2.4.14) The installation is finished. Select “Run PyCharm Community Edition” and click Finish.

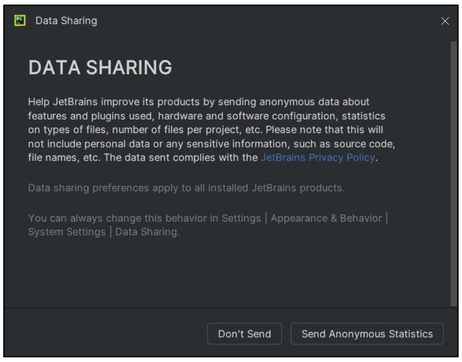
Figure 2.4: The installation steps of `PyCharm` under Microsoft `Windows` (continued).



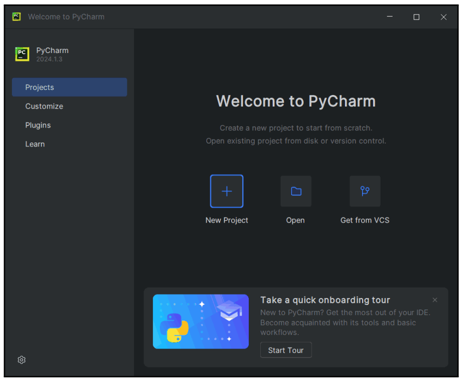
(2.4.15) The welcome screen of PyCharm.



(2.4.16) We read the user agreement, confirm that we read it, and click Continue.



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



(2.4.18) Finally, PyCharm is ready to use.

Figure 2.4: 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 text output and then exit. It therefore will consist of the single statement `print("Hello World!")`, 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 2.5, which were taken on my Ubuntu machine. On Microsoft Windows, this will look very similar (maybe with the exception of "\" instead of "/" as directory separator characters in paths).

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

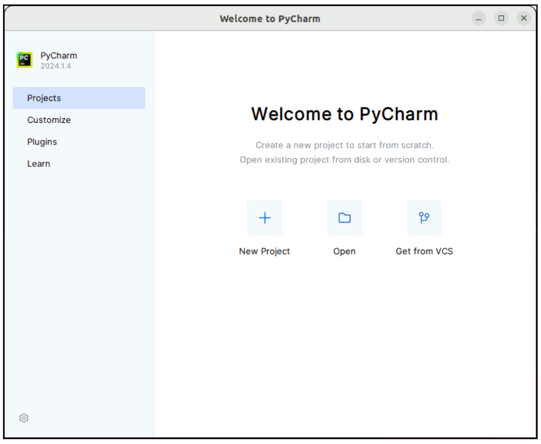
Listing 2.1: Our very first Python program, which just prints "Hello World!" (stored in file `very_first_program.py`; output in Listing 2.2)

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

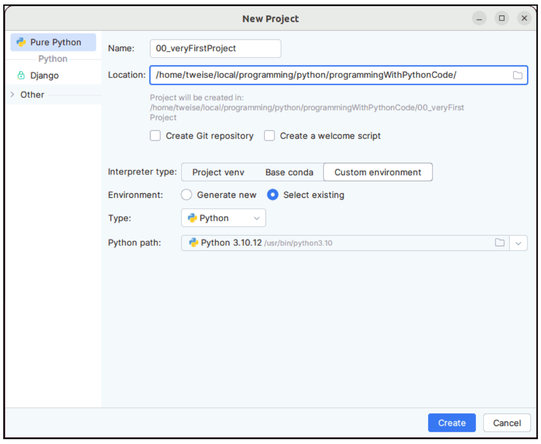
↓ `python3 very_first_program.py` ↓

Listing 2.2: The standard output stream (stdout) of the program `very_first_program.py` given in Listing 2.1.

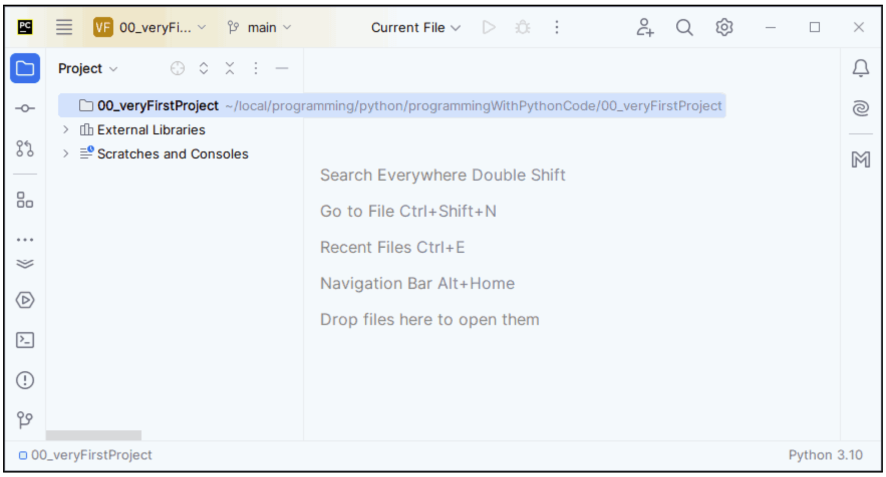
```
Hello World!
```



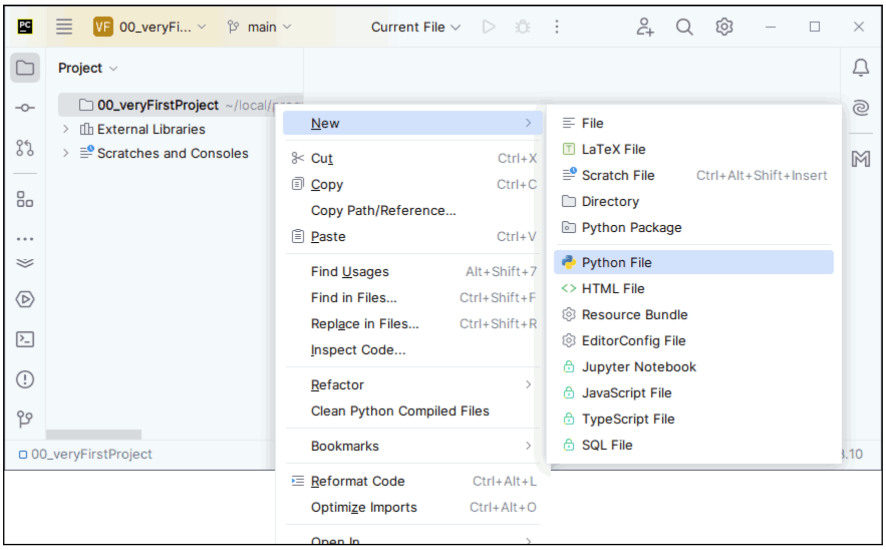
(2.5.1) Creating a new project in PyCharm, step 1: click New Project.



(2.5.2) Creating a new project in PyCharm, step 2: Make sure that Pure Python is selected in the left pane, then select a name for the project (here: `00_veryFirstProject`), a directory location (some path on my Ubuntu machine, you will pick some other directory), and we select the current Python installation as Custom Environment. We finally click Create.

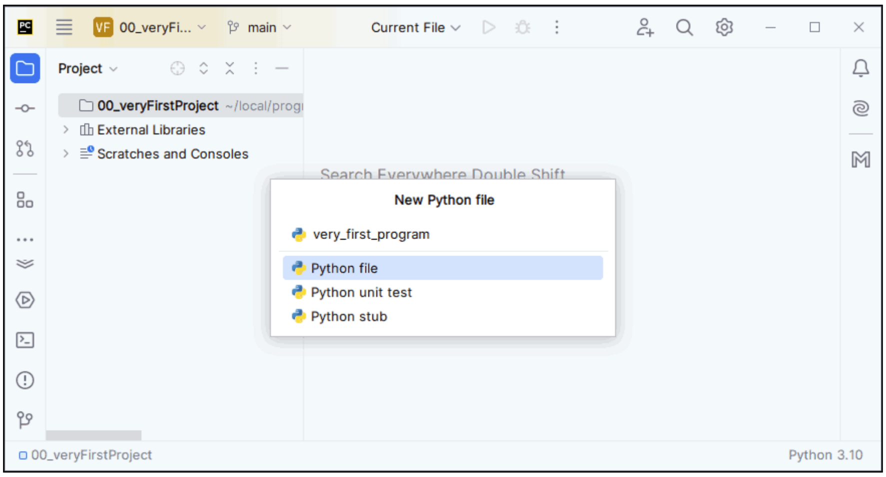


(2.5.3) The new and empty project has been created.

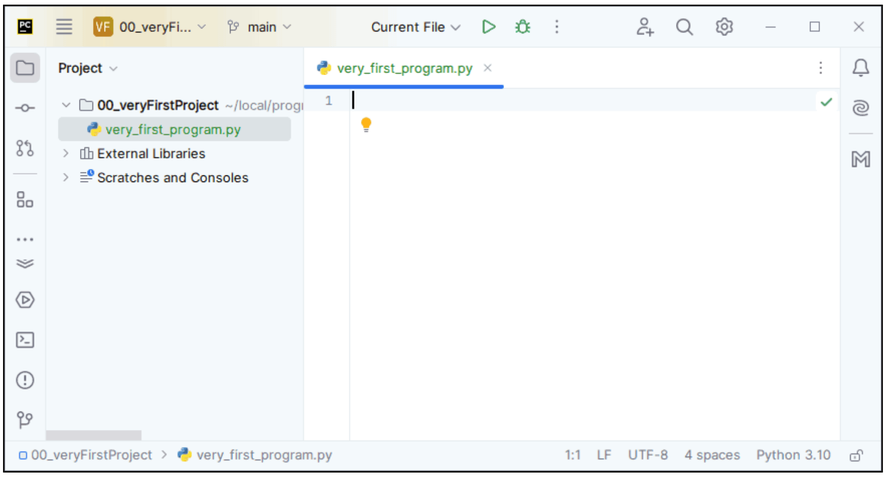


(2.5.4) We create a new Python file within this project by right-clicking on the project folder `00_veryFirstProject` and selecting New » Python File.

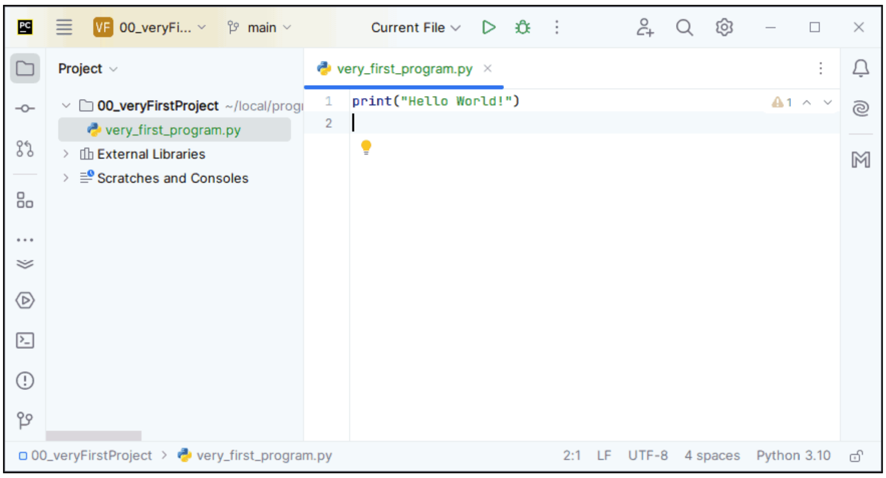
Figure 2.5: The steps to create a new Python file in a new PyCharm project and to then run it (Ubuntu).



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

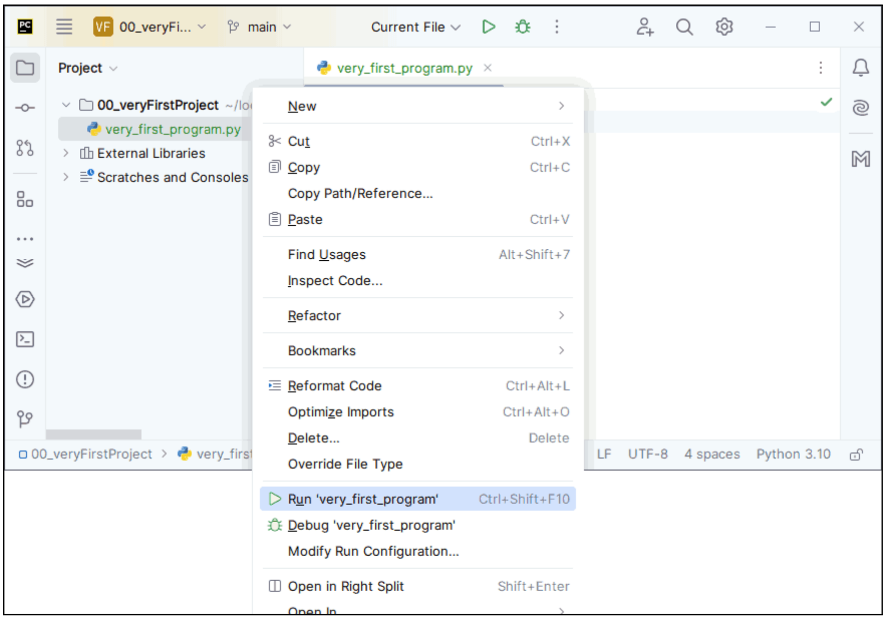


(2.5.6) The new and empty file `very_first_program.py` has been created in the project `00_veryFirstProject`.

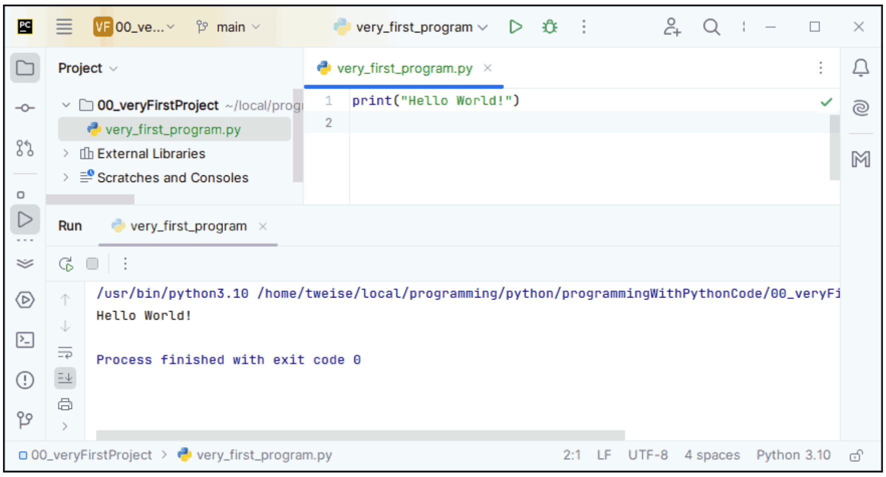


(2.5.7) We enter the text from Listing 2.1. `PyCharm` automatically saves it.

Figure 2.5: The steps to create a new `Python` file in a new `PyCharm` project and to then run it (`Ubuntu`, continued).



(2.5.8) 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.



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

Figure 2.5: The steps to create a new Python file in a new PyCharm project and to then run it (Ubuntu, continued).

Figure 2.5.1. Here, we will New Project, which takes us to the second project creation screen shown in Figure 2.5.2. 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 00_veryFirstProject.

Below the name text box, you can select the destination directory in which the project folder should be created in the Location: box. You can ignore the text in the screenshot, as this is just the path that I am using on my Ubuntu machine. Instead, you will choose some suitable directory on your computer. (Notice that in Microsoft Windows, the directory separator character is "\" and not "/" as on my Ubuntu Linux.) Either way, on my machine, I choose the rather elaborate path home ▸ tweise ▸ local ▸ programming ▸ python ▸ programmingWithPythonCode. PyCharm will create the folder 00_veryFirstProject for our new project *inside* this directory. But, as said, you will choose some other suitable place on your computer.

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 3.10.12" at the time of this writing.[1] We finally click Create. We now have created our first and empty PyCharm Python project – as illustrated in Figure 2.5.3.

We can now create the Python file to write our actual program code. To do this, we right-click on the folder 00_veryFirstProject in our Project tree view pane on the left-hand side (see Figure 2.5.4). 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 2.5.5. Here, we enter the name for our first program. 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 2.5.6.

We now enter the single line of code from Listing 2.1 into the editor, as illustrated in Figure 2.5.7. We do not need to 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 2.5.8.

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. 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.

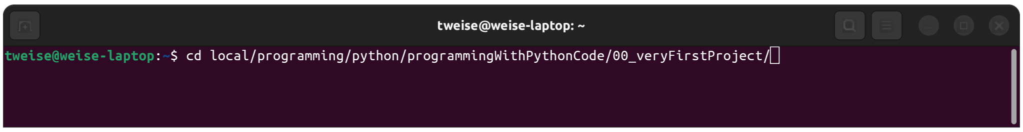
Congratulations. You now have written, saved, and executed your first ever Python program!

2.4 Python in the Terminal

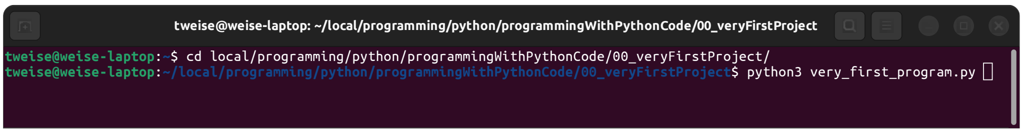
In total, are four more ways in which we can execute a Python program:

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 2.5.

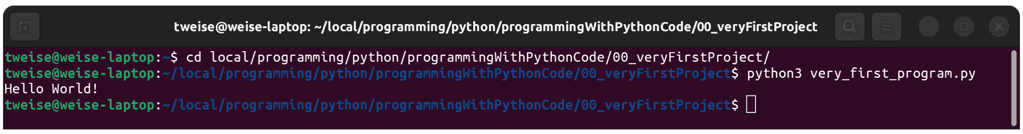
[1] Now I am using Python 3.12, but this is not important.



(2.6.1) 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 Enter.

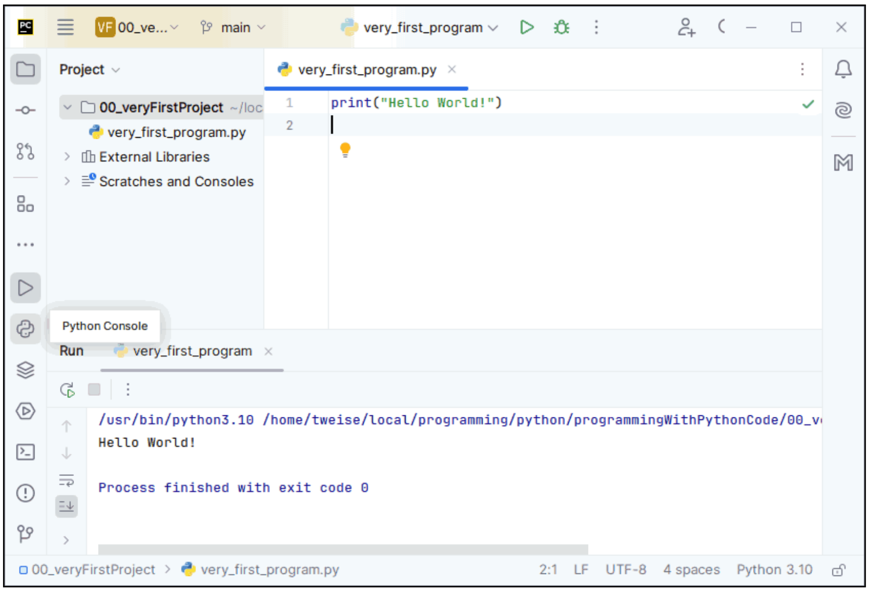


(2.6.2) Execute a Python program "program.py" by typing `python3 program.py` and hit Enter. In our case, the program is "very_first_program.py".

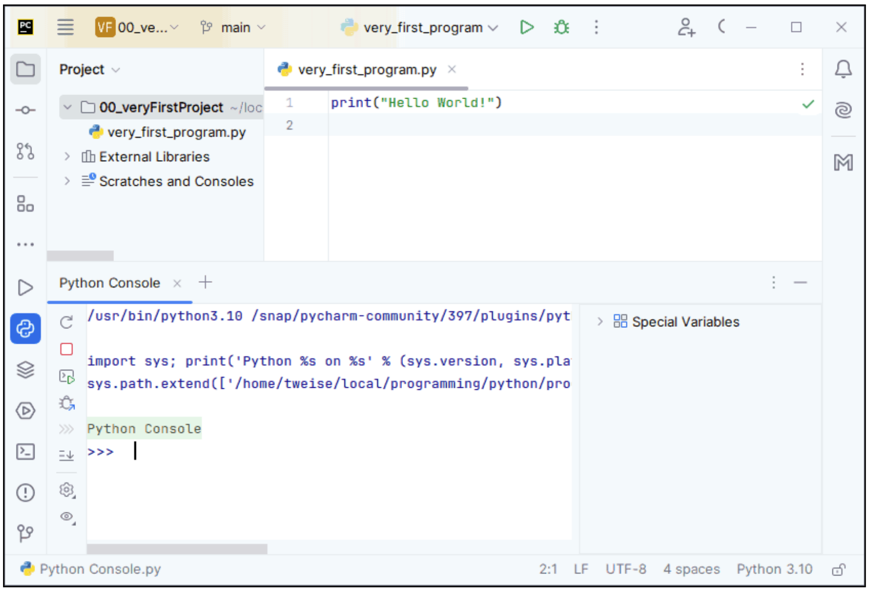


(2.6.3) As expected, the text "Hello World!" appears in the terminal.

Figure 2.6: Example of executing a Python program in a terminal (on Ubuntu).



(2.7.1) Pressing the on the vertical icon bar on the left side of the `PyCharm` window.



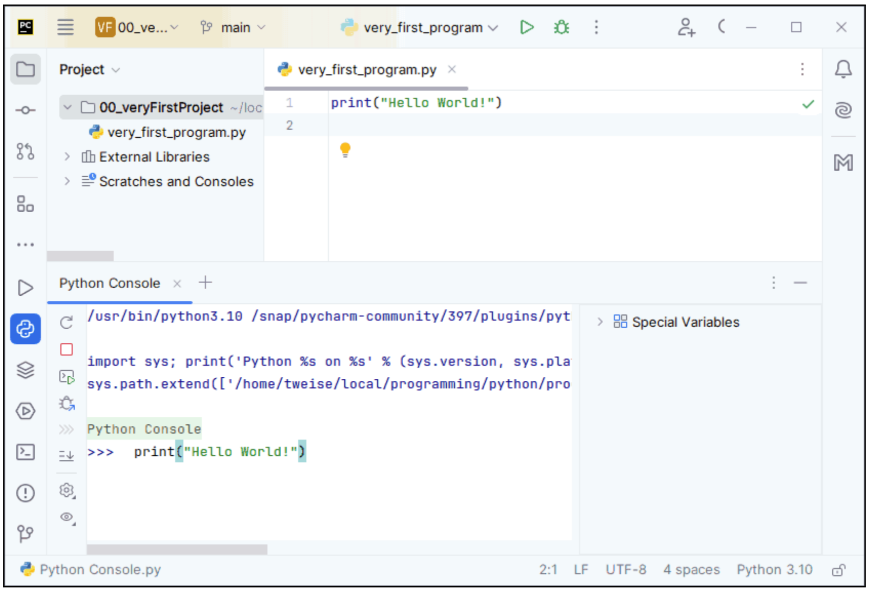
(2.7.2) The `PyCharm` `Python` console is open.

Figure 2.7: Entering the "Hello World!" program from Listing 2.1 directly into the `Python` console offered by `PyCharm`.

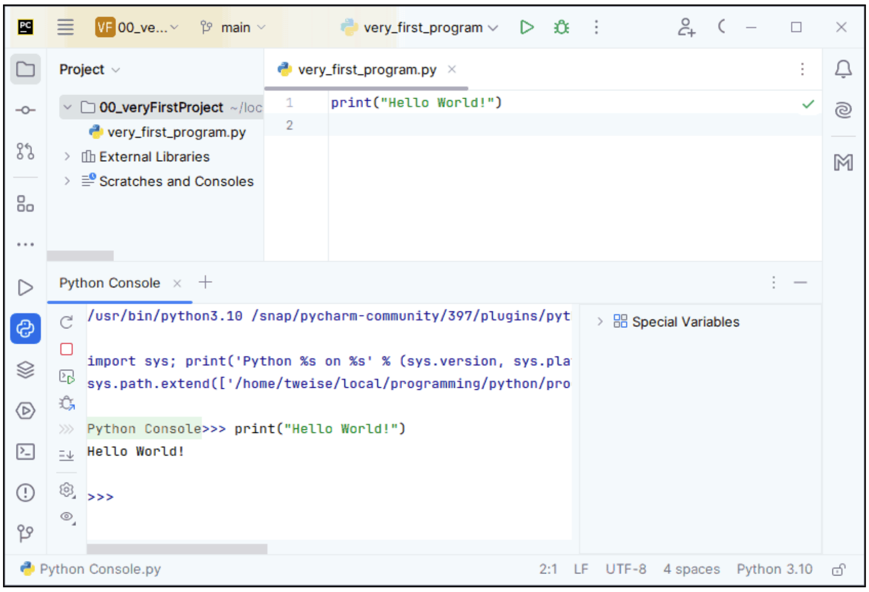
2. Actually, we can also write a `Python` program with a normal text editor. A `Python` program is just a normal text file, after all. We can execute such a text file by entering its directory and typing `python3 programName` (where `programName` is `very_first_program.py`, in our case) and hitting Enter. Then the program is executed directly in the terminal. This process is shown in Section 2.4.1 and Figure 2.6.

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 2.7.

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 2.8.



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



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

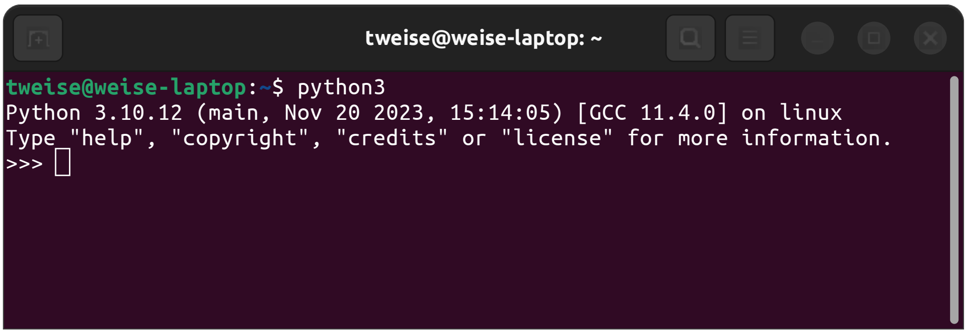
Figure 2.7: Entering the "Hello World!" program from Listing 2.1 directly into the `Python` console offered by `PyCharm` (continued).

2.4.1 Executing a `Python` Program in a Terminal

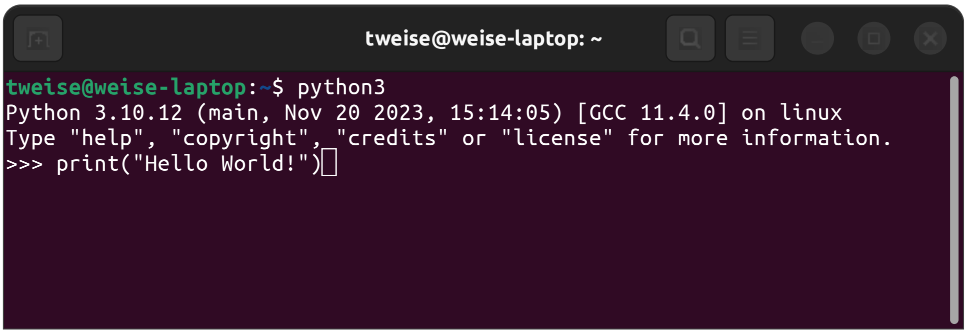
In order to directly execute a `Python` program in a terminal, we first need to open one. Under `Ubuntu Linux`, we simply press Ctrl+Alt+T. Under Microsoft `Windows`, we have to press ⊞+R, type in `cmd`, and hit ↵. Once the terminal is open, we need to change into the directory where the program is located. Under both `Linux` and Microsoft `Windows`, this can be done by typing the command `cd`, followed by the path to the directory, and hitting Enter.[2] We provide a screenshot for that, taken under `Ubuntu Linux`, in Figure 2.6.1. 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 2.6.2 we do this and hit Enter, which causes the `Python` interpreter to execute our program. The output "Hello World!" is then printed into the terminal in Figure 2.6.3.

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

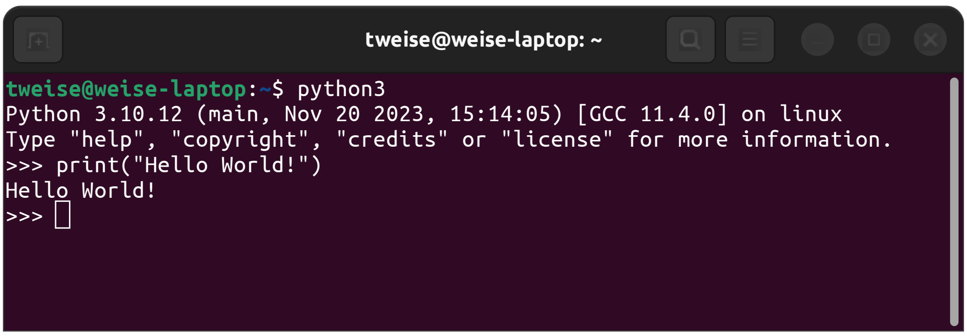
(2.8.1) Open a terminal (by pressing Ctrl+Alt+T under Ubuntu Linux; under Microsoft Windows press ⊞+R, type in `cmd`, and hit ↲), enter `python3`, then hit Enter.



(2.8.2) The Python console opens in the terminal.

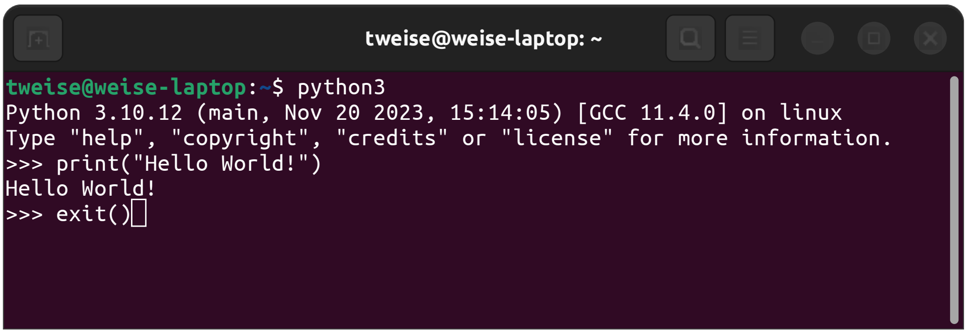


(2.8.3) We enter the "Hello World!" program from Listing 2.1 and press Enter.

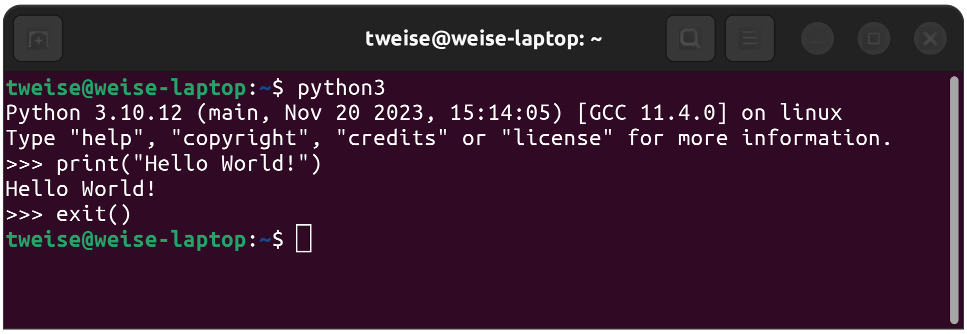


(2.8.4) And indeed, the output is "Hello World!".

Figure 2.8: Writing a program in the Python console in the terminal (Ubuntu).



(2.8.5) We exit the console by typing `exit()` and pressing Enter.



(2.8.6) We are back in the normal terminal.

Figure 2.8: Writing a program directly in the `Python` console in the terminal (`Ubuntu`, continued).

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 2.7) or opened in a terminal (Figure 2.8).

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 2.7.1. This directly brings us to the `Python` console (Figure 2.7.2). We can enter the one-line-program from Listing 2.1, as illustrated in Figure 2.7.3. Notice that the input prompt of the console is marked by the three greater characters `>>>` after which we enter our text. Pressing Enter after writing the code leads to the expected output shown in Figure 2.7.4. 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 `Microsoft Windows`, we have to press ⊞ + R, type in `cmd`, and hit ↵. Either way, the terminal opens and we can enter `python3` and press Enter, as shown in Figure 2.8.1. Now the `Python` interpreter starts right inside the terminal (Figure 2.8.2). The prompt, i.e., the place where we can write our code, again is preceded by the `>>>` characters. As illustrated in Figure 2.8.3, we copy the single line of code, `print("Hello World!")` from Listing 2.1 and press Enter. The output "Hello World!" is printed as expected in Figure 2.8.4. 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 Enter, as shown in Figure 2.8.5, which causes the `Python` interpreter to exit. We are now back in the basic terminal, as shown in Figure 2.8.6. In these figures, I was using `Ubuntu Linux`. On Microsoft `Windows` or other `Linux` variants, the process would have looked quite similar.

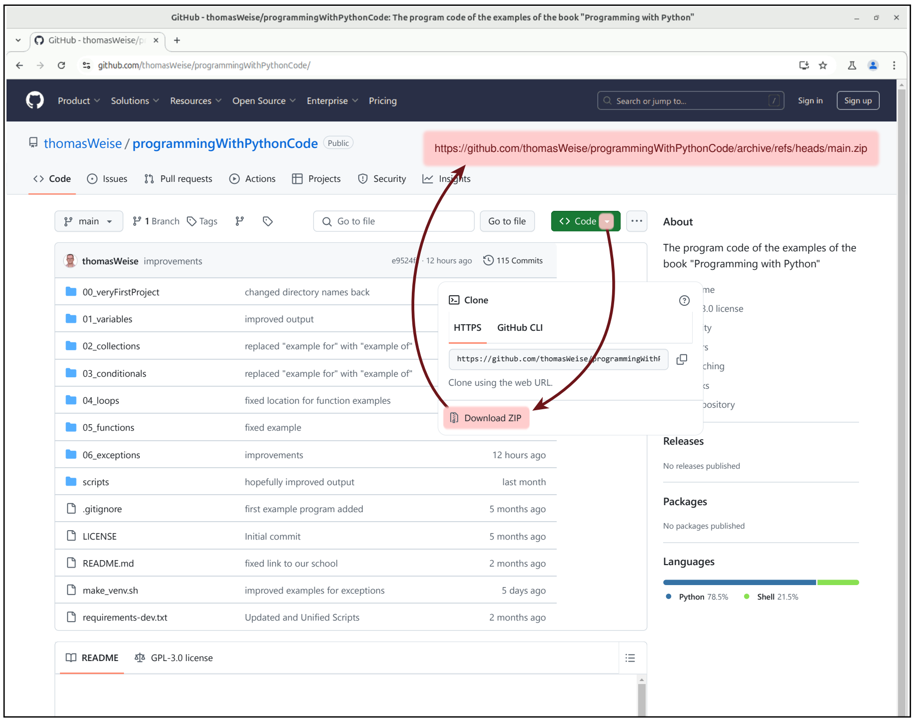


Figure 2.9: Downloading all the example source codes as a single `zip` archive from `https://github.com/thomasWeise/programmingWithPythonCode`.

> **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.

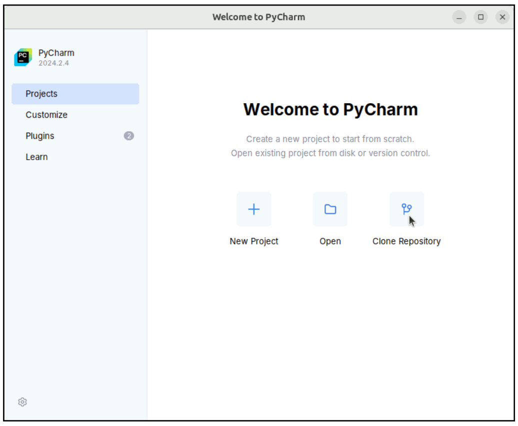
2.5 Getting the Examples from this Book

This book comes with a lot of examples 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 `programmingWithPythonCode`. You can directly access this repository at GitHub under `https://github.com/thomasWeise/programmingWithPythonCode`.

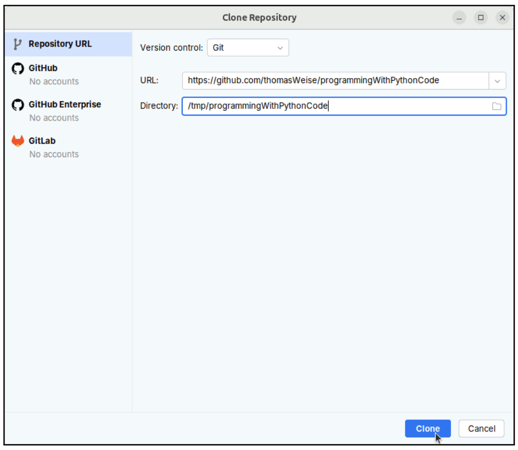
On the website, you can directly download all the examples as illustrated in Figure 2.9. First, you would click on the little downward facing triangle in the button [Code]. This will open a small dialog [Clone] in which you can click on the [Download ZIP] button. This, in turn, enters the Uniform Resource Locator (URL) `https://github.com/thomasWeise/programmingWithPythonCode/archive/refs/heads/main.zip` into your browser's download queue. 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.

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 2.10. In the `PyCharm` welcome screen, you click [Clone Repository] as shown in Figure 2.10.1. In the 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 2.10.2, 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 2.10.3.

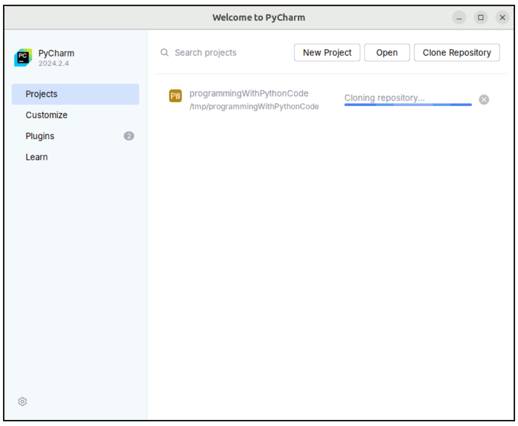
Once the repository has been downloaded, `PyCharm` may ask you whether you trust this project. After making sure that you indeed downloaded the examples for this book (and if you deem this code trustworthy), you can click [Trust Project], as Figure 2.10.4. Finally, as Figure 2.10.5 shows, you can now



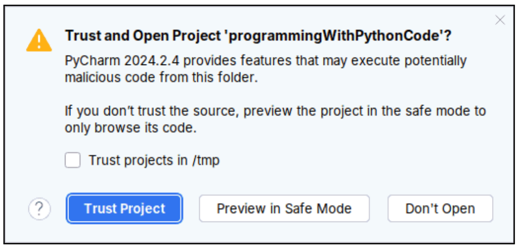
(2.10.1) Click Clone Repository in the PyCharm welcome screen.



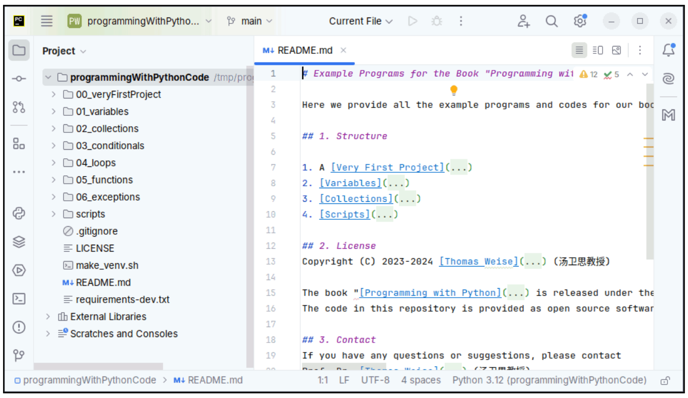
(2.10.2) Selecting URL: `https://github.com/thomasWeise/programmingWithPythonCode` and a reasonable destination Directory: in the next dialog, then click Clone.



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



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



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

Figure 2.10: Using PyCharm to clone (download) and import all the examples from this book.

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.

2.6 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 print "Hello World!". That would not make much sense. `print` expects one parameter. This parameter cannot just be anything. It must be a text.

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 need to invoke `exit` if we use the Python console in the terminal explicitly. If we just run a program, then after the last instruction of the program was executed, then the interpreter will also terminate with exit code 0. Indeed, when we executed our first program in Section 2.3, we saw exactly that happen in Figure 2.5.9. Different from the parameter of `print`, which must be some text, the parameter of `exit` needs to be a number.

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. Somtimes, 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 3.4),
- `str`: strings, i.e., portions of text of arbitrary length (Section 3.5), and
- `None`: nothing, which is the result of any command that does not explicitly return a value (Section 3.6).

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.

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 [224]. 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.[1]

3.2.1 Integer Arithmetics

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 for this. 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 Enter, 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, we enter `4 + 3` and hit Enter. 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 Enter. The result is `35`.

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

Fractional division, 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.

> **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.

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 Figure 3.2, 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 most 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 $-2^{63}..2^{63}-1$.

[1] 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` $\equiv \mathbb{Z}$.

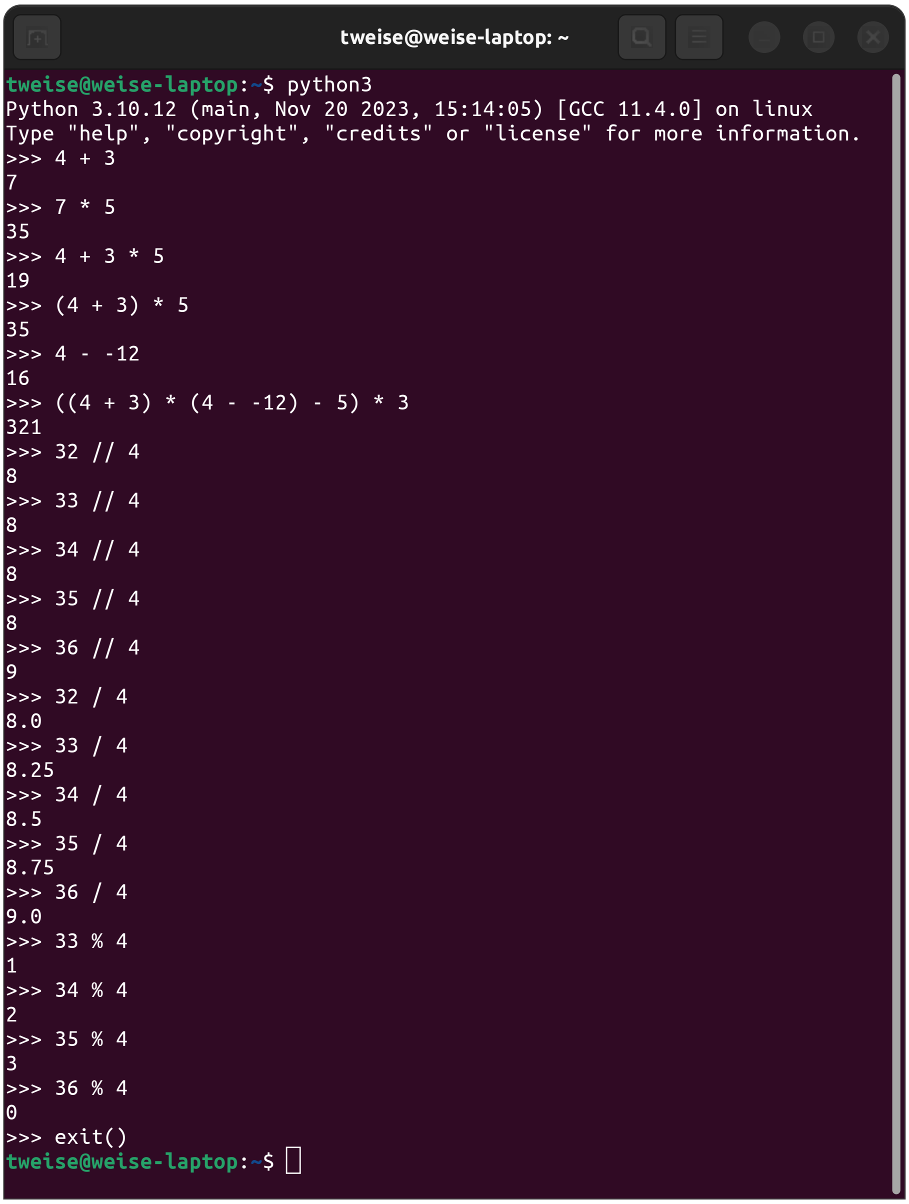


Figure 3.1: Examples for Python integer math in the console, part 1 (see Figure 3.2 for part 2).

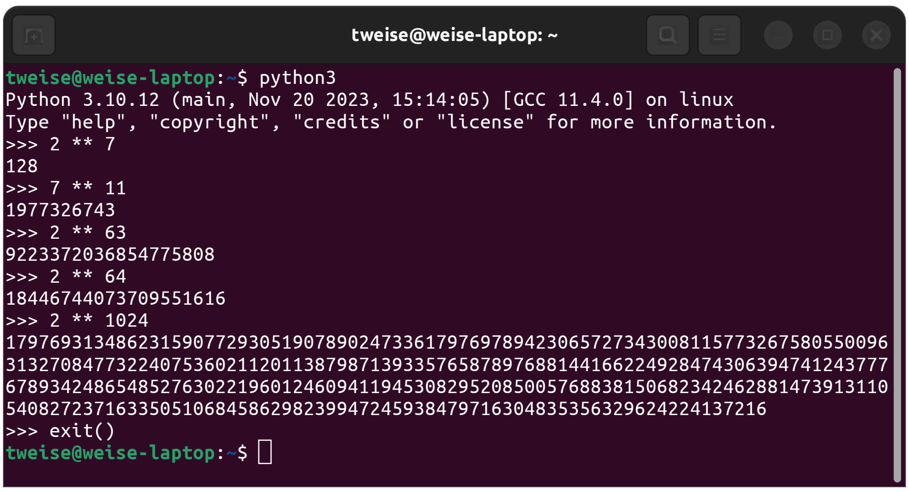


Figure 3.2: Examples for Python integer math in the console, part 2 (see Figure 3.1 for part 1).

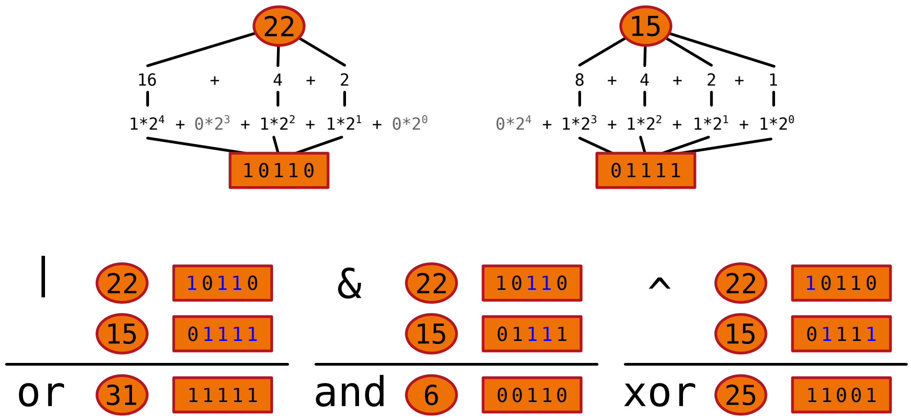


Figure 3.3: Examples for how integer numbers are represented as bit strings in the computer (upper part) and for the binary (bitwise) operations *and*, *or*, and *exclusive or* (often called *xor*).

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 `2 ** 1024`, which is such a huge number that it wraps four times in the output of our Python console in Figure 3.2! Python integers are basically unbounded. 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 integers 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 $b_0 2^0 + b_1 2^1 + b_2 2^2 + b_3 2^3 + b_4 2^4 \dots$, where the b_i-values each are either 0 or 1. Then, $\dots b_4 b_3 b_2 b_1 b_0$ 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 3.3, 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 Figure 3.4, `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 3.3 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

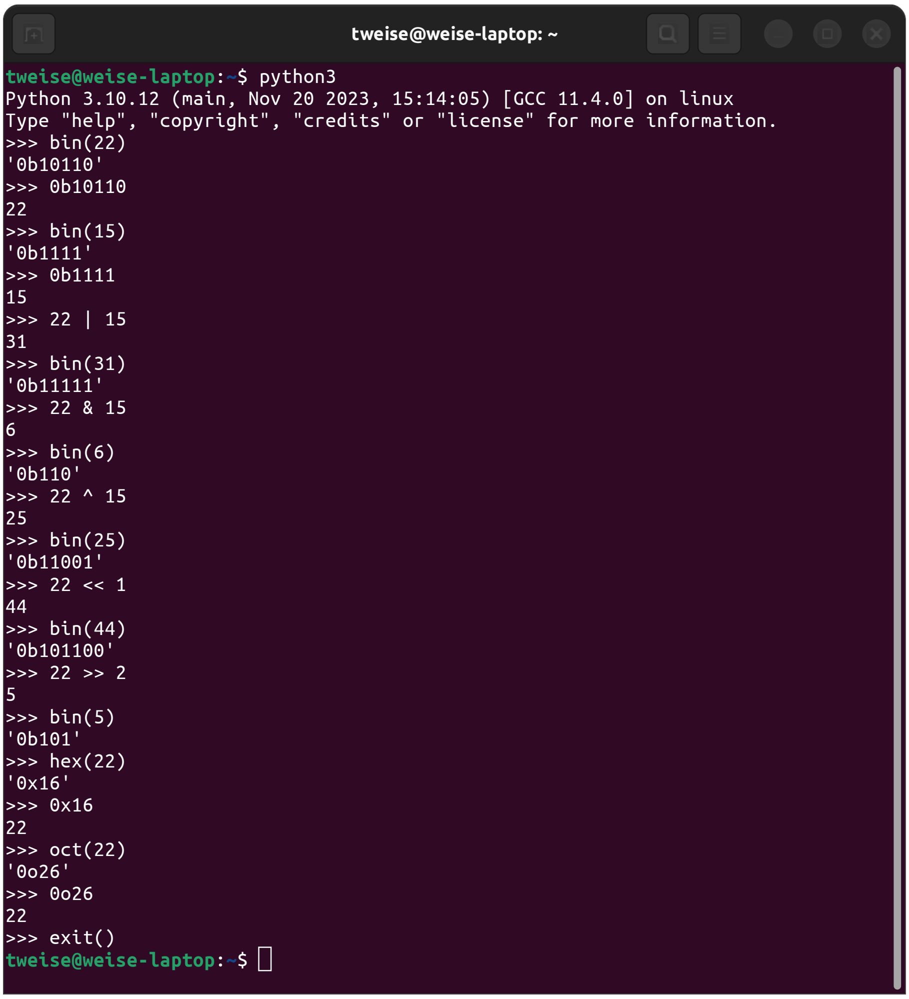


Figure 3.4: Examples for the binary representation of integers and the operations that apply to it.

(whereas 2^2 and 2^1 occured in both of them). This is confirmed by typing `bin(25)`, which results in `'0b11001'`.

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 at 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 basis 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 to $2 * 8^1 + 6$. Similarly, this octal number can be entered as `0o26`. 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

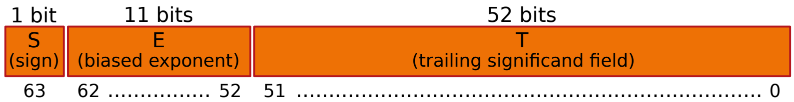


Figure 3.5: The structure of an 64 bit / double precision IEEE Standard 754 floating point number [129, 136].

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 the numbers $\pi \approx 3.141\,592\,653\,590\ldots$ and $e \approx 2.718\,281\,828\,459\ldots$ from highschool maths. They are transcendental [97, 145, 195], i.e., their fractional digits never end and 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 dynamically represent fractional numbers exactly even in typical everyday cases? 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 [196], which, in turn, internally have a 64 bit IEEE Standard 754 floating point number layout [129, 136]. The idea behind this standard is to represent both very large numbers, like 10^{300} and very small numbers, like 10^{-300}. In order to achieve this, the 64 bits are divided into three pieces, as illustrated in Figure 3.5.

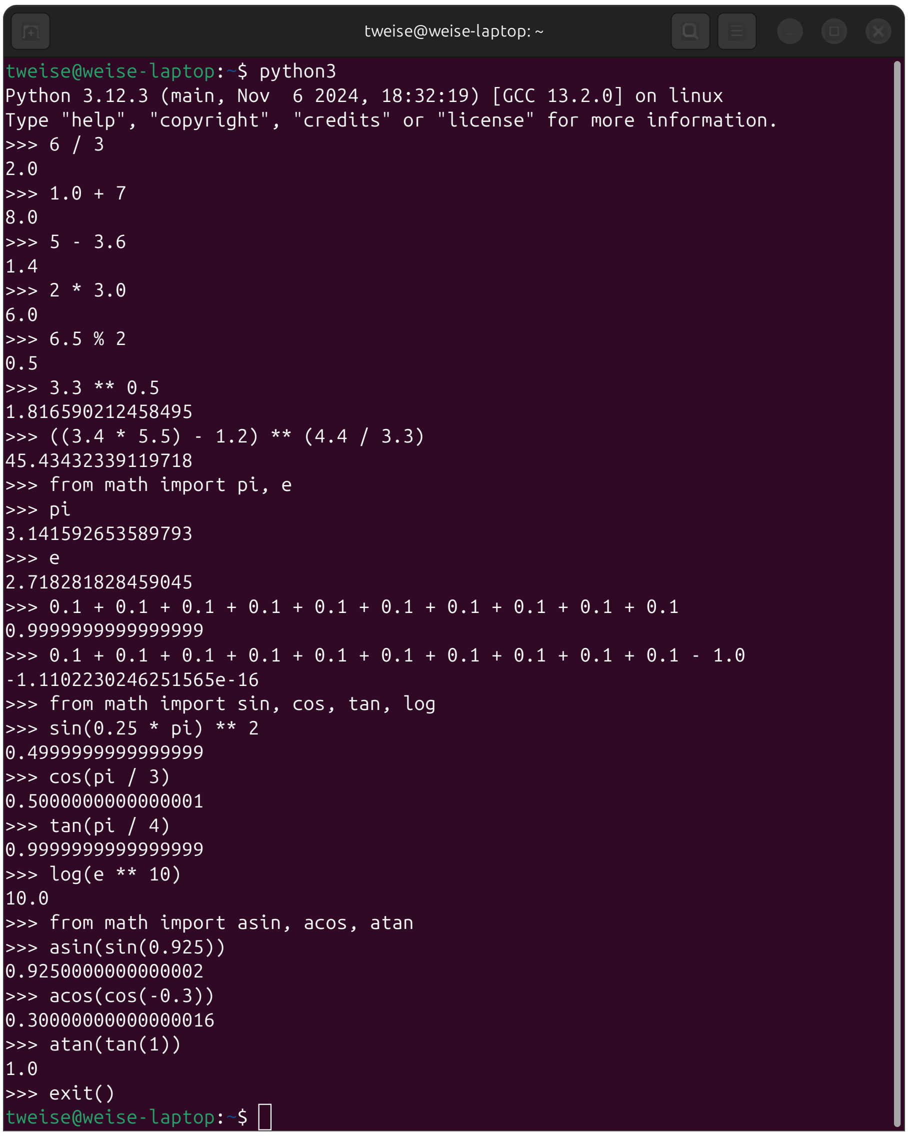
First Time Readers and Novices: You just need to understand that `floats` have limited precision. You can jump right to the next section.

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 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 bias of 1023 is subtracted. Thus, a stored value of 2000 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.

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. The important thing to remember is: Floating point numbers 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



```
tweise@weise-laptop:~$ python3
Python 3.12.3 (main, Nov  6 2024, 18:32:19) [GCC 13.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 6 / 3
2.0
>>> 1.0 + 7
8.0
>>> 5 - 3.6
1.4
>>> 2 * 3.0
6.0
>>> 6.5 % 2
0.5
>>> 3.3 ** 0.5
1.816590212458495
>>> ((3.4 * 5.5) - 1.2) ** (4.4 / 3.3)
45.43432339119718
>>> from math import pi, e
>>> pi
3.141592653589793
>>> e
2.718281828459045
>>> 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1
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
>>> sin(0.25 * pi) ** 2
0.4999999999999999
>>> cos(pi / 3)
0.5000000000000001
>>> tan(pi / 4)
0.9999999999999999
>>> log(e ** 10)
10.0
>>> from math import asin, acos, atan
>>> asin(sin(0.925))
0.9250000000000002
>>> acos(cos(-0.3))
0.30000000000000016
>>> atan(tan(1))
1.0
>>> exit()
tweise@weise-laptop:~$
```

Figure 3.6: Basic arithmetics with floating point numbers in Python.

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 [98].

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 for the basic arithmetic operations available for `floats` in Figure 3.6.

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 [19, 244] 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 irrational 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.[2] 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 not necessarily need irrational or 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 $\frac{1}{10} = 0.1$ [98]. 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.

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 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. Calculating `asin(sin(0.925))` indeed yields `0.9250000000000002`. 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 [17, 223].

Due to the limited precision, it could be that you add two numbers $c = a + b$ 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 [223]. 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 [17]...

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

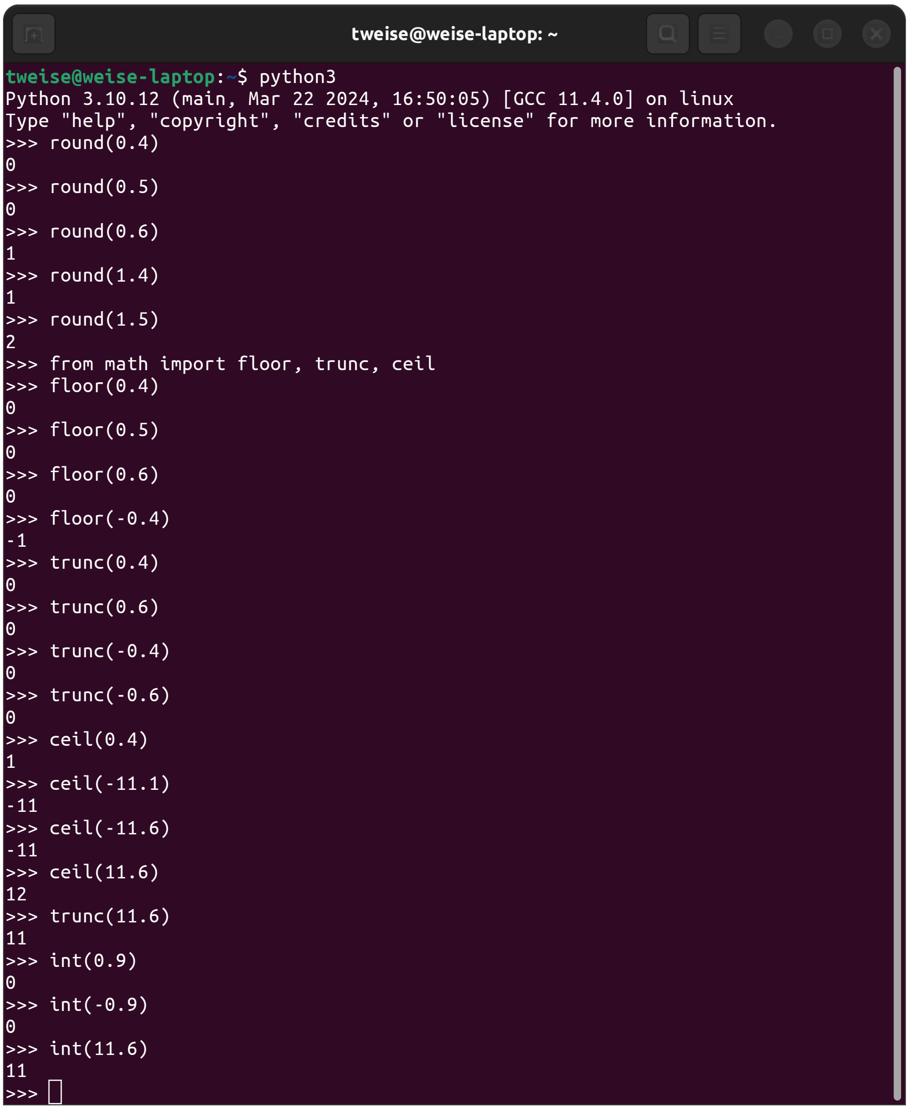


Figure 3.7: Rounding `float` values to `int` values.

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, it rounds it to the one that is even. This can be very surprising, as we learned in school that $x.5$ should be rounded to $x+1$. This will only happen with `round` if $x+1$ is even. We find examples for the behavior of `round` in Figure 3.7. `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)`

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

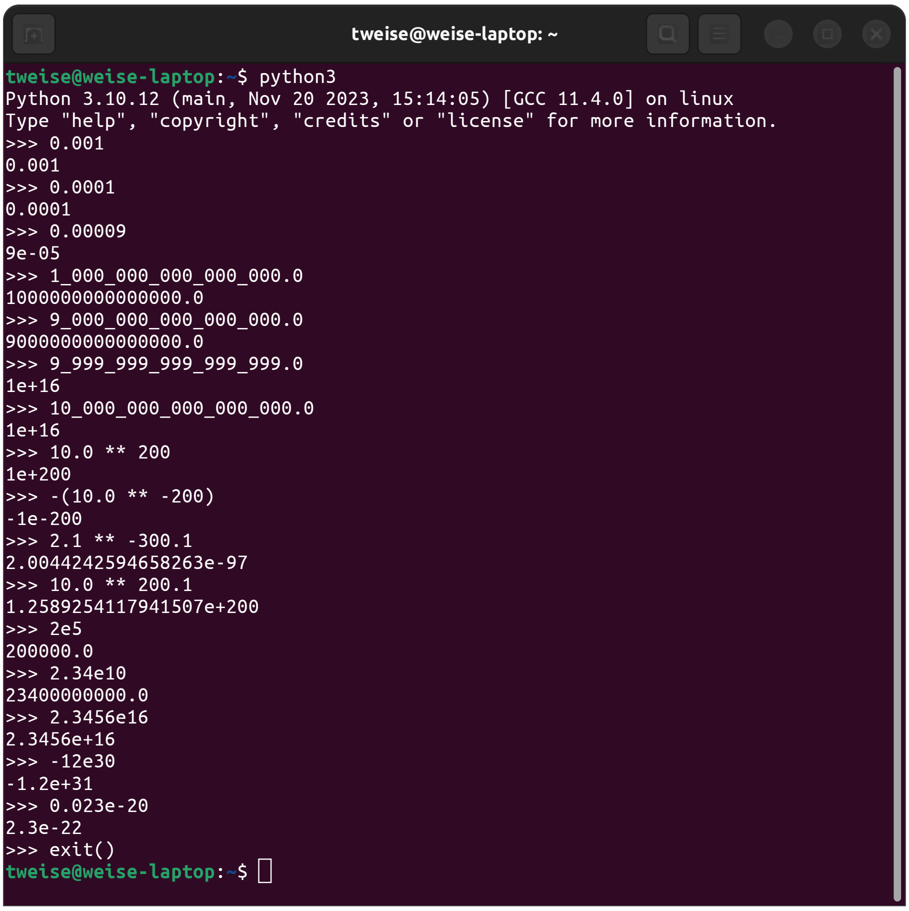


Figure 3.9: Examples for the scientific notation in Python.

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`.

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* work as one would expect!

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 15 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 $\beta * 10^{\gamma}$ (such that

$\beta * 10^\gamma = \alpha$, obviously). 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 order to make sure that each number α has unique representation, it is defined that α 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 3.8. *This notation only applies to `floats`, **not** `ints`.*

In Figure 3.9 we provide some examples for 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 [32]? If not, you know now.

Best Practice 4 If you need to specify large integers or floats, using underscores (`_`) to separate groups of digits can be very helpful [32]. 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 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 standard for its class `Double`. In its documentation [56], 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 Figure 3.11, 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 [129, 136], 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`.

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

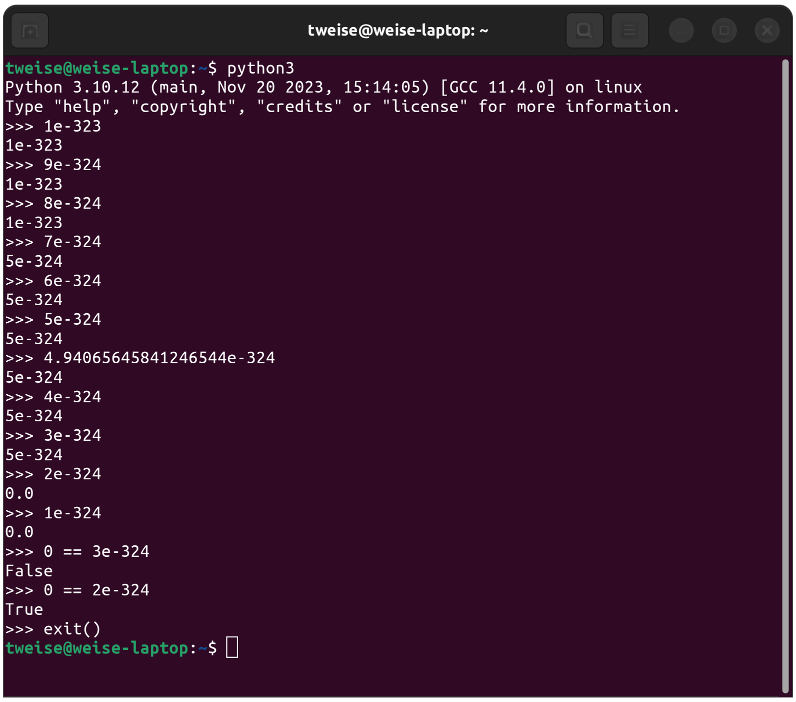


Figure 3.10: What happens with very small floating point numbers in `Python`?

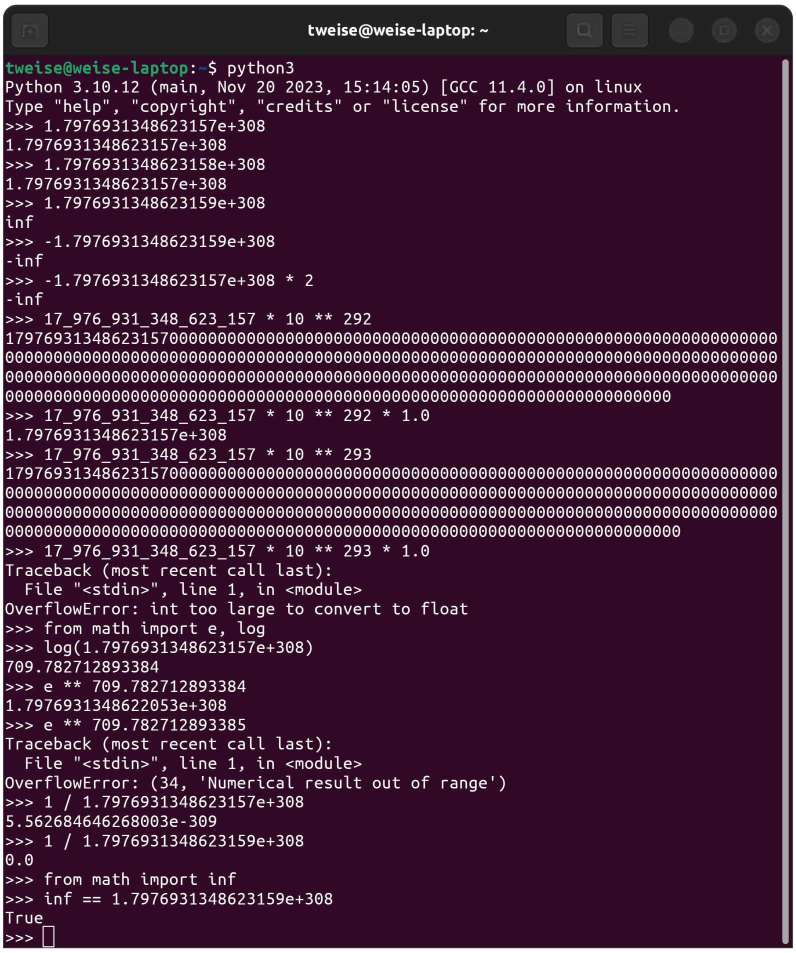
IEEE Standard 754 floating point number [129, 136]. The text `2e-324` that we enter into the `Python` console will therefore be translated to the `float` `0.0`. The comparison `2e-324 == 0.0` therefore results in `True`, while `3e-324 == 0.0` is still `False`.[3] So we learned what happens if we try 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, 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 Figure 3.11. 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`. `-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 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 9.3.4, 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`

[3]See Section 3.4 for more information on comparisons and the `bool` datatype with its values `True` and `False`.



```
tweise@weise-laptop: ~
tweise@weise-laptop:~$ python3
Python 3.10.12 (main, Nov 20 2023, 15:14:05) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 1.7976931348623157e+308
1.7976931348623157e+308
>>> 1.7976931348623158e+308
1.7976931348623157e+308
>>> 1.7976931348623159e+308
inf
>>> -1.7976931348623159e+308
-inf
>>> -1.7976931348623157e+308 * 2
-inf
>>> 17_976_931_348_623_157 * 10 ** 292
17976931348623157000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000
>>> 17_976_931_348_623_157 * 10 ** 292 * 1.0
1.7976931348623157e+308
>>> 17_976_931_348_623_157 * 10 ** 293
17976931348623157000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000
>>> 17_976_931_348_623_157 * 10 ** 293 * 1.0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: int too large to convert to float
>>> from math import e, log
>>> log(1.7976931348623157e+308)
709.782712893384
>>> e ** 709.782712893384
1.7976931348622053e+308
>>> e ** 709.782712893385
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: (34, 'Numerical result out of range')
>>> 1 / 1.7976931348623157e+308
5.562684646268003e-309
>>> 1 / 1.7976931348623159e+308
0.0
>>> from math import inf
>>> inf == 1.7976931348623159e+308
True
>>>
```

Figure 3.11: What happens with very large floating point numbers in Python?

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`.

Finally, `inf` also 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`.

list Now, `inf` not actually being ∞ is a little bit weird. But it can get even stranger, as you will see in Figure 3.12. `inf` is a number which is too big to be represented as `float` *or* ∞. 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`. 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`[4]. However, `nan` is *really* different from really *anything*. Therefore, `nan == nan` is `False` and `nan != nan` is `True`!

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

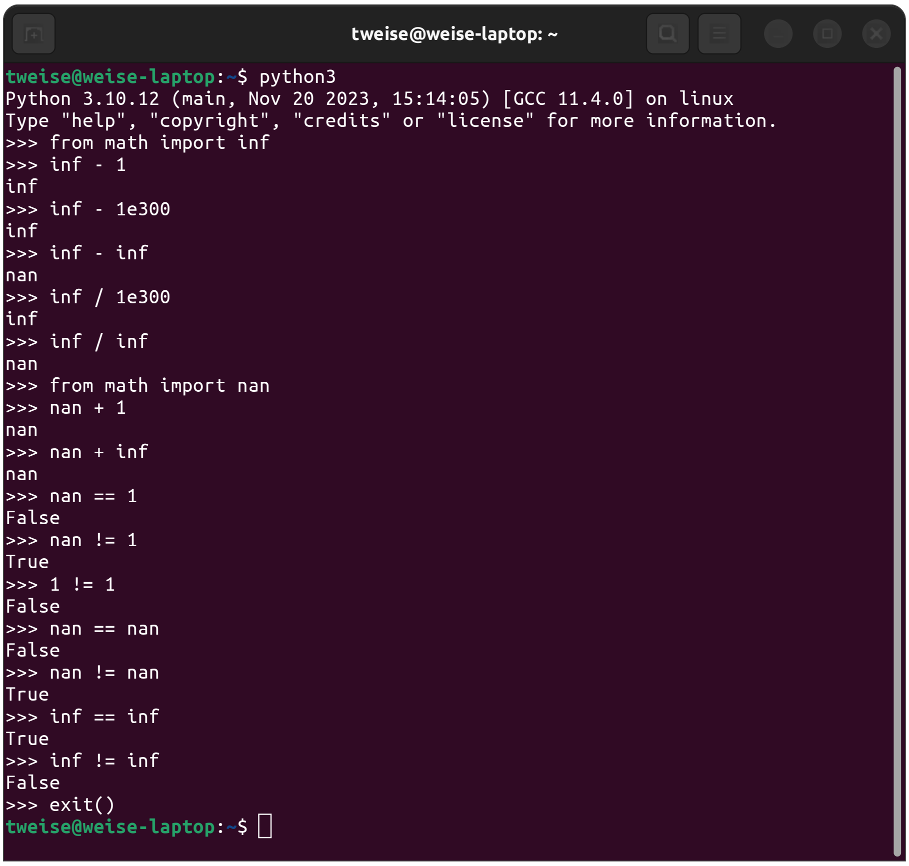


Figure 3.12: Not a Number, i.e., `nan`.

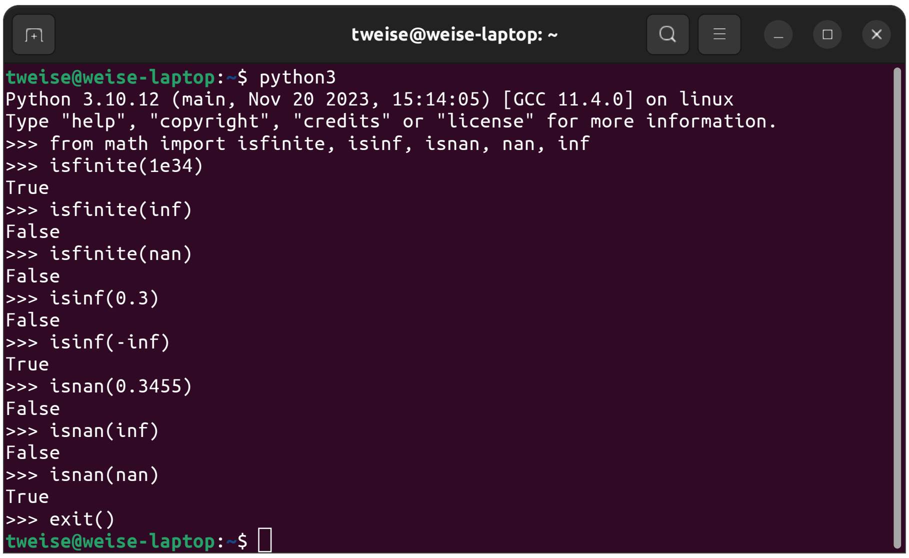


Figure 3.13: Checking for `nan` and `inf` via `isfinite`, `isinf`, and `isnan`.

Either way, the possible occurrence `inf` and `nan` in floating point computations is a cumbersome issue. If we want to further process results of a computation, 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 Figure 3.13. `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 [115], 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 $10^{\texttt{XXX}}$.

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.

3.4 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`. They are fundamental for making decisions in a program, i.e., for deciding what to do based on data.

3.4.1 Comparisons

In the sections on `floats` and `ints`, we learned how to do arithmetics 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:

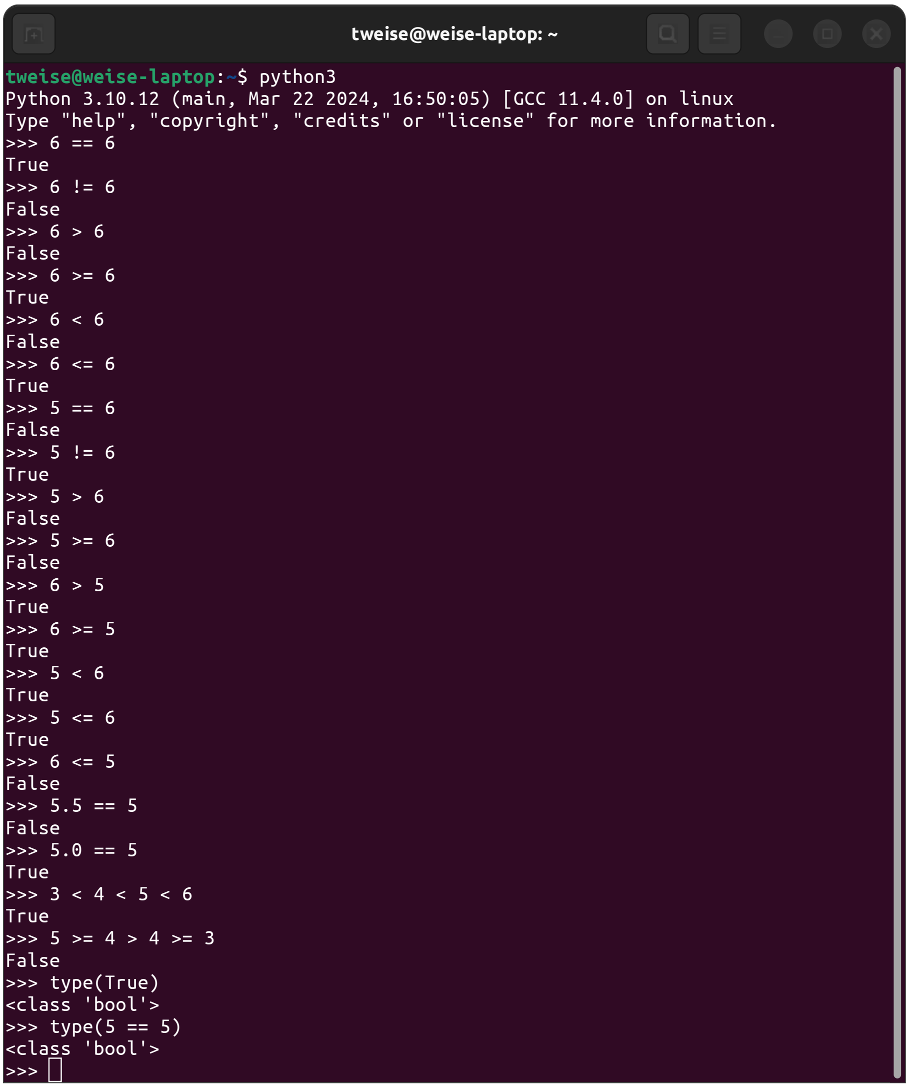


Figure 3.14: The results of basic comparisons are instances of `bool`.

- 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 Figure 3.14. 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`.

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 Figure 3.12, 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.

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

(3.15.1) 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

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

a	not a
False	True
True	False

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

Figure 3.15: The truth tables for the Boolean operators `and`, `or`, and `not`.

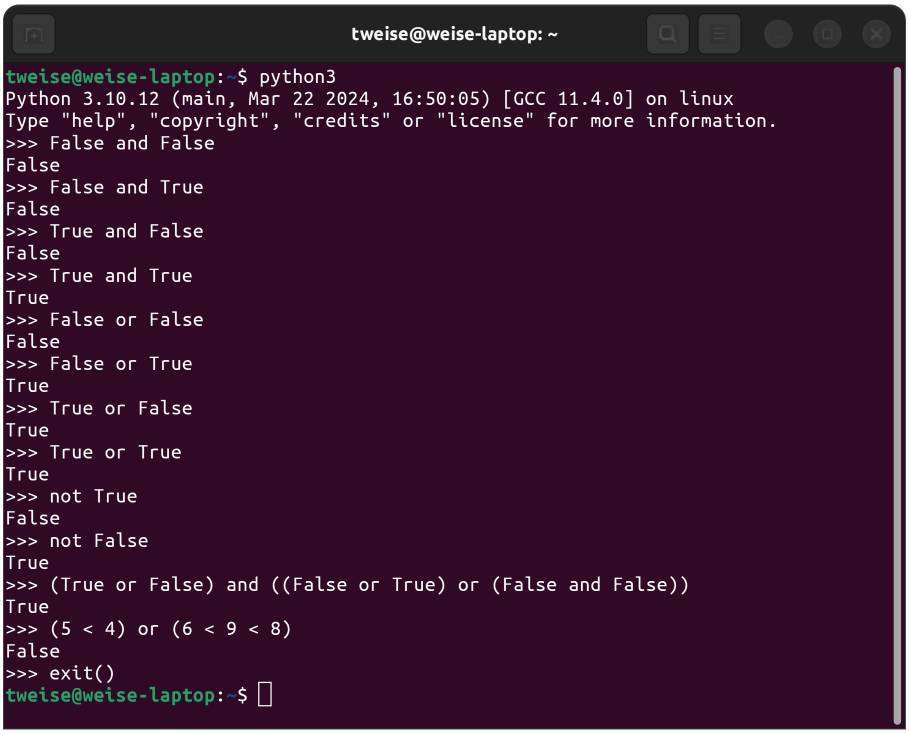


Figure 3.16: The `bool` values can be combined with the Boolean logical operators `and`, `or`, and `not`.

3.4.2 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 3.15.

- 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 3.15.1.
- A Boolean disjunction, i.e., `and`, is `True` if at least one of its two operands is `True` and `False` otherwise, as shown in Figure 3.15.2.
- The Boolean negation, i.e., `not`, is `True` if its operand is `False`. Otherwise, it is `False`, as shown in Figure 3.15.3.

In Figure 3.16 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`.

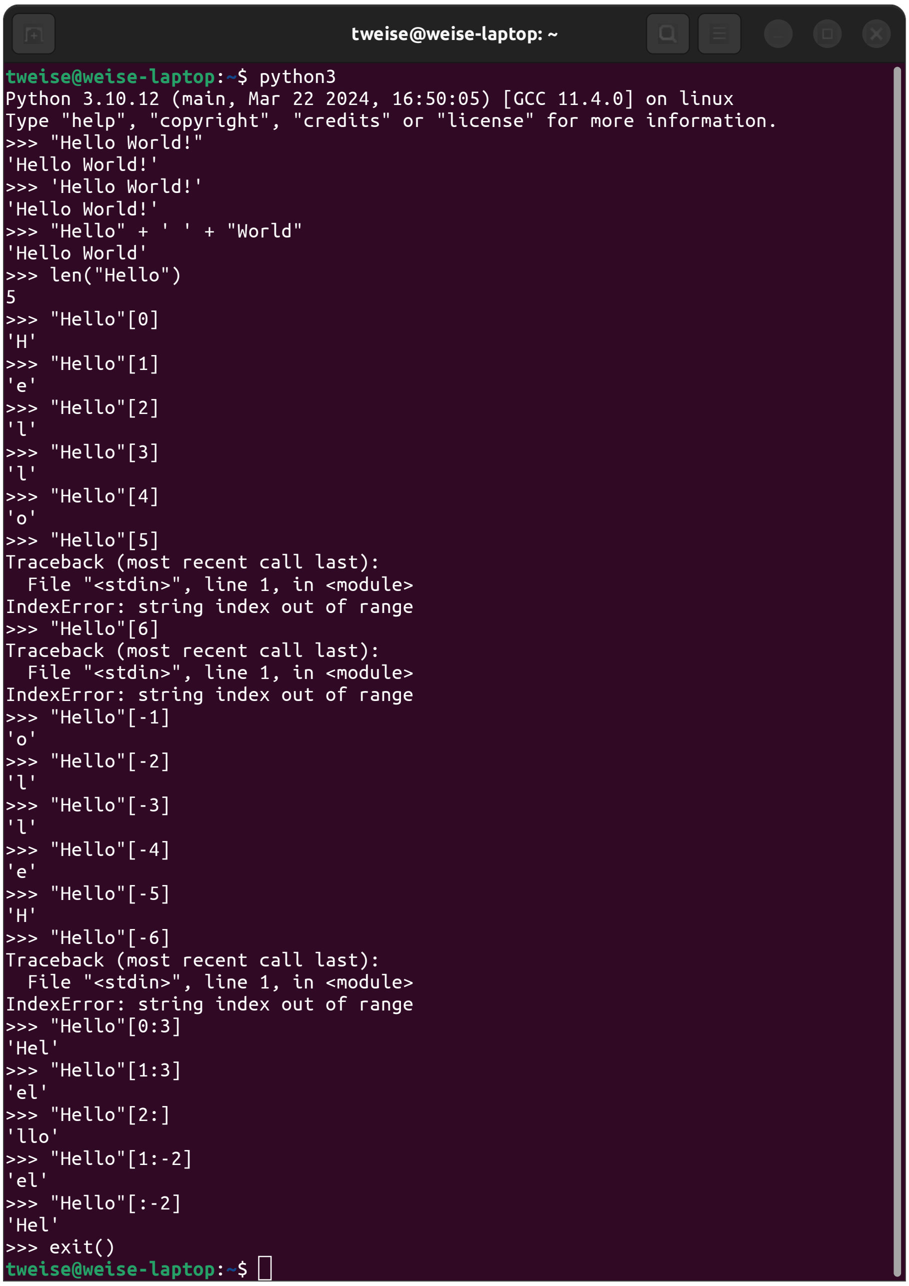


Figure 3.17: Specifying string literals and indexing its characters.

3.4.3 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.

3.5 Text Strings

The fourth important 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.

3.5.1 Basic String Operations

As Figure 3.17 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 5 When defining a string literal, the double-quotation mark variant (`"..."`) may be preferred over the single-quotation mark variant (`'...'`); see also Best Practice 6.

One basic operation is string concatenation: `"Hello"+ ' ' + "World"` concatenates the three strings `"Hello"`, `" "`, and `"World"`. The result is `"Hello World"`. Notice how the singe 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"`. 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 Figure 3.18 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

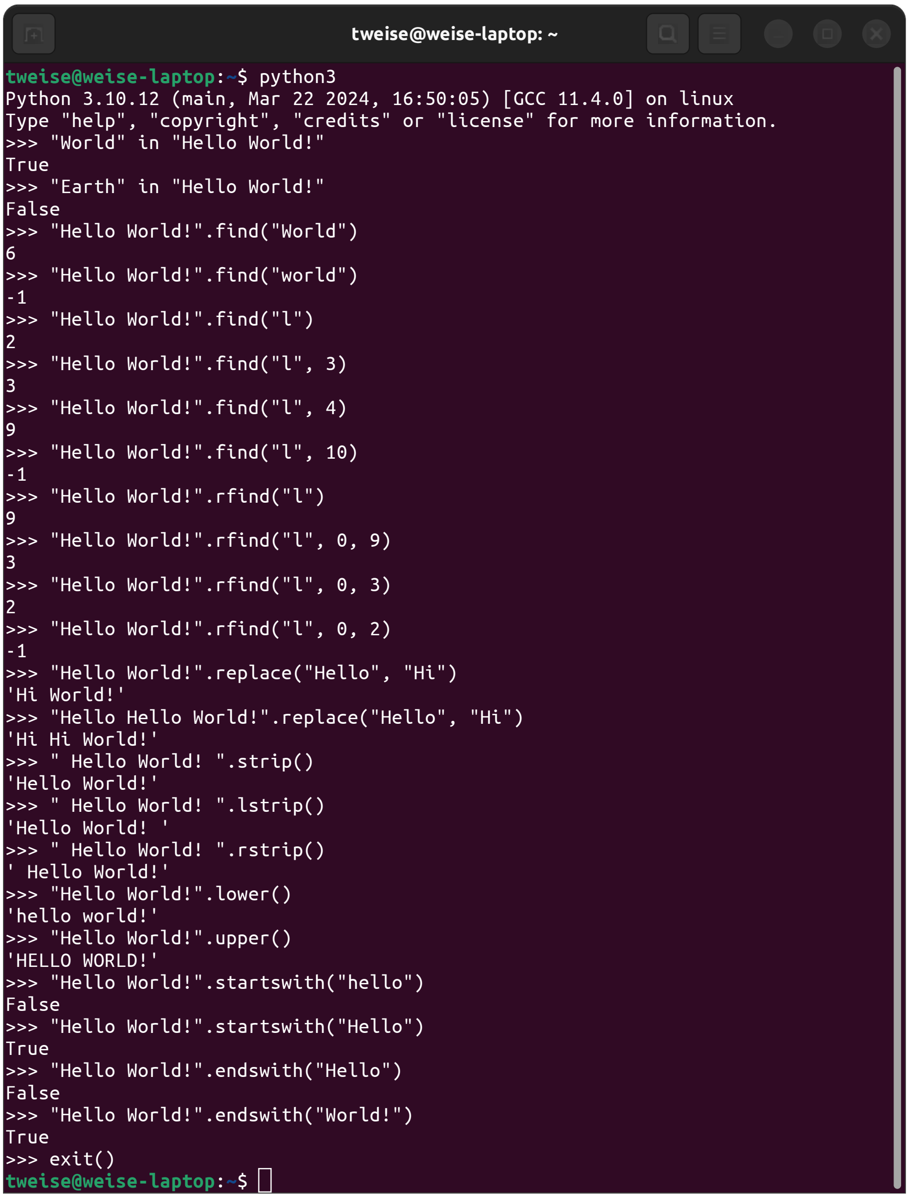


Figure 3.18: Some more basic string operations.

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 string operations are case-sensitive: `"World" != "world"` would be `True`. 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,

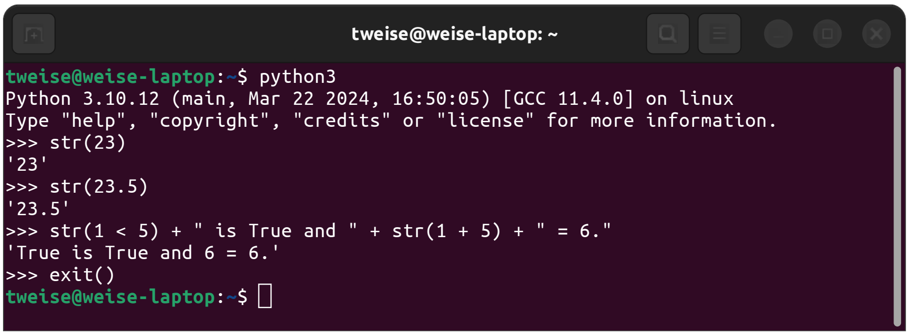


Figure 3.19: The `str` function converts objects to strings.

we need to supply an inclusive starting and exclusive ending index of the range to be searched. `"Hello World!".rfind("l", 0, 9)` searches for any "l" from index 8 down to 0 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 Hello World!".replace("Hello", "Hi")` becomes `"Hi Hi World!"`.

Often, we want to remove all leading or trailing whitespace characters 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 final functions, we can check whether a string begins or ends with another, we can use `startswith` and `endswith`, respectively. `"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 [276].

3.5.2 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 Figure 3.19 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 [37, 99, 183, 263]. 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 Figure 3.20, we first consider the f-string `f"{12345678901234}is a really big integer."`[5].

[5] The code that formats my inline Python examples sometimes eats spaces after { or }. Therefore, some of the strings

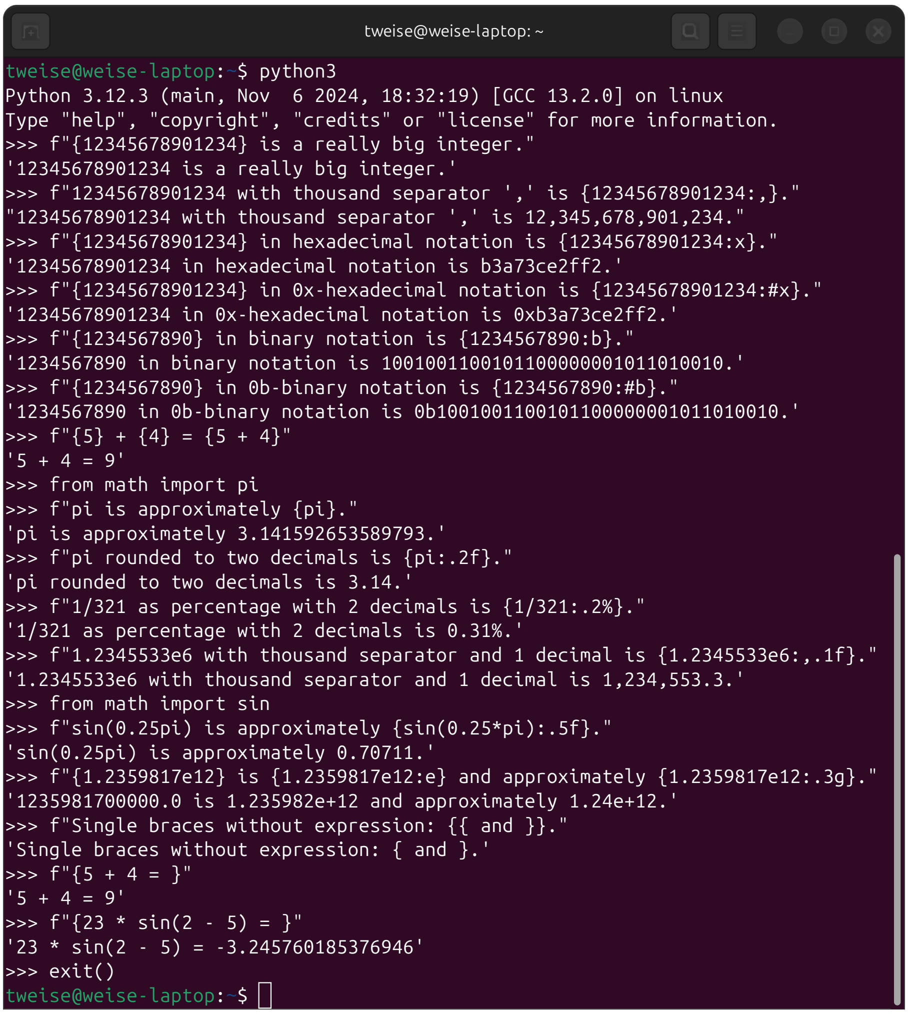


Figure 3.20: Python f-strings in action.

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 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 f-string `f"{1234567890}in binary notation is {1234567890:b}."` is turned to `"1234567890 in binary notation is 1001001100101100000001011010010."`. We can also add

presented here may look a bit off. In Figure 3.20, they are printed correctly, though.

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 final example, let us look at a very cool ability of f-strings. Often, we want to print an expression together with its result [125]. 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.

You are now able to convert the results of your computations to nice text.

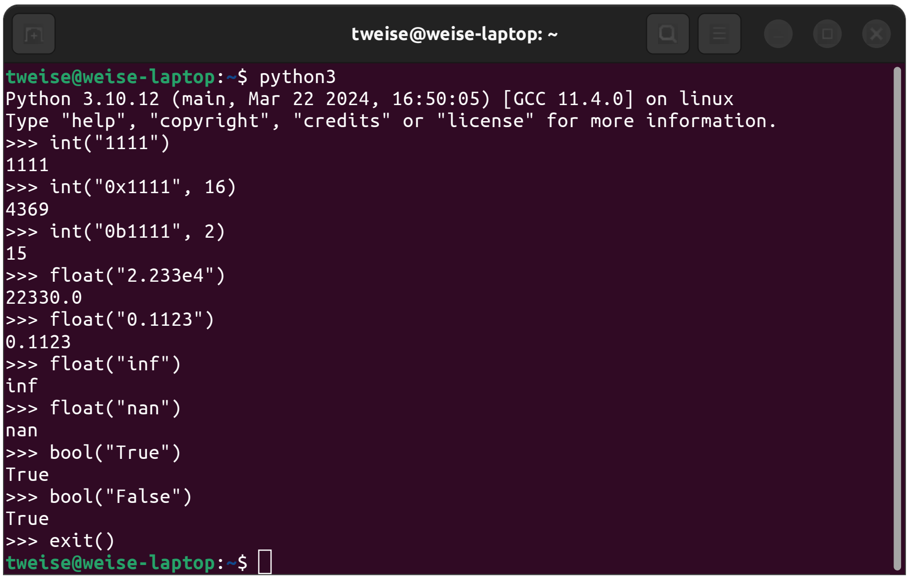


Figure 3.21: Converting strings to other datatypes.

3.5.3 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.

Figure 3.21 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 $1 + 2 + 4 + 8 = 15$.

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`.

Finally, the function `bool` converts the strings `"True"` and `"False"` to `True` and `False`, respectively. With this, you are also able to convert strings to data that you can use as input for your computations.

3.5.4 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*. 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

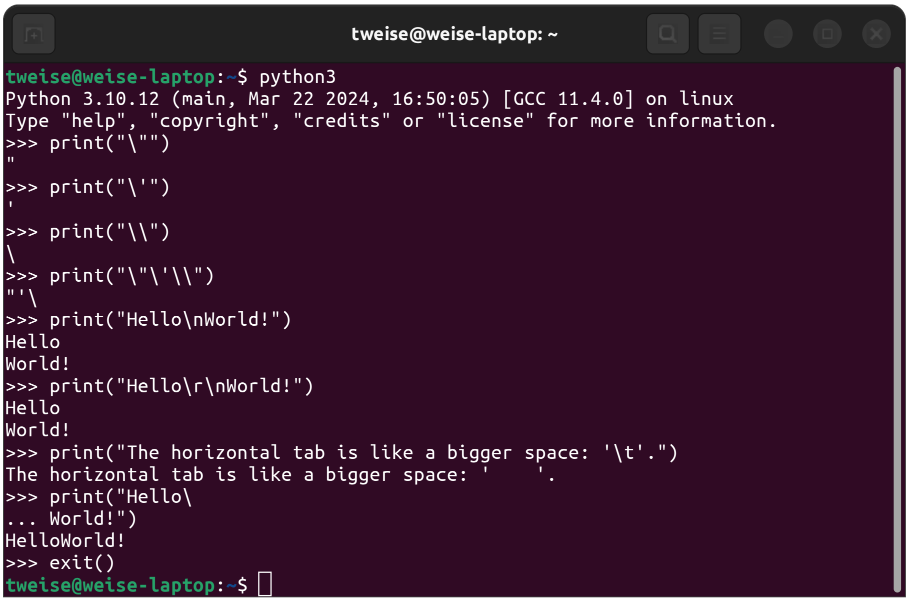


Figure 3.22: Escaping special characters in Python strings.

the next character should be considered as a normal character and not be interpreted as any special character, like a string delimiter.

In Figure 3.22, 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. 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 over 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 Enter, and continue the typing the string. The linebreak will then be ignored entirely. Therefore, if you print `print("Hello\`, hit Enter, 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 this section work for both f-strings and normal strings.

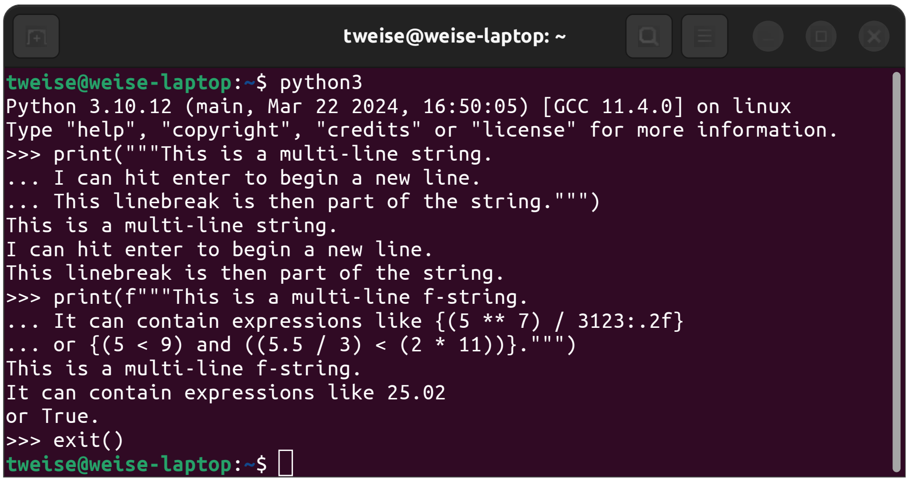


Figure 3.23: Examples for multi-line strings.

3.5.5 Multi-Line Strings

Before we discussed that strings in `Python` are delimited either by `"` or `'` on each side. However, we can actually delimit them 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 Enter 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 16.

Best Practice 6 When defining a multi-line string literal, the double-quotation mark variant (`"""..."""`) is usually preferred over the single-quotation mark variant (`'''...'''`) [108, 302].

Figure 3.23 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 Figure 3.23 presents such a multi-line f-string with two expressions for (string) interpolation which spans over three lines.

3.5.6 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 of8 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 3.5). But how does this work with text?

Well, by mapping characters to numbers. 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 [5, 295], which, however, 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, many different mappings have historically evolved and still exist today. In China, different mappings specialized to Chinese characters exist additionally, including the historical GB 2312 [60], GBK [53], or the newer GB 18030 [335]. Today, the vast majority of computers and systems understand one common standard that covers all

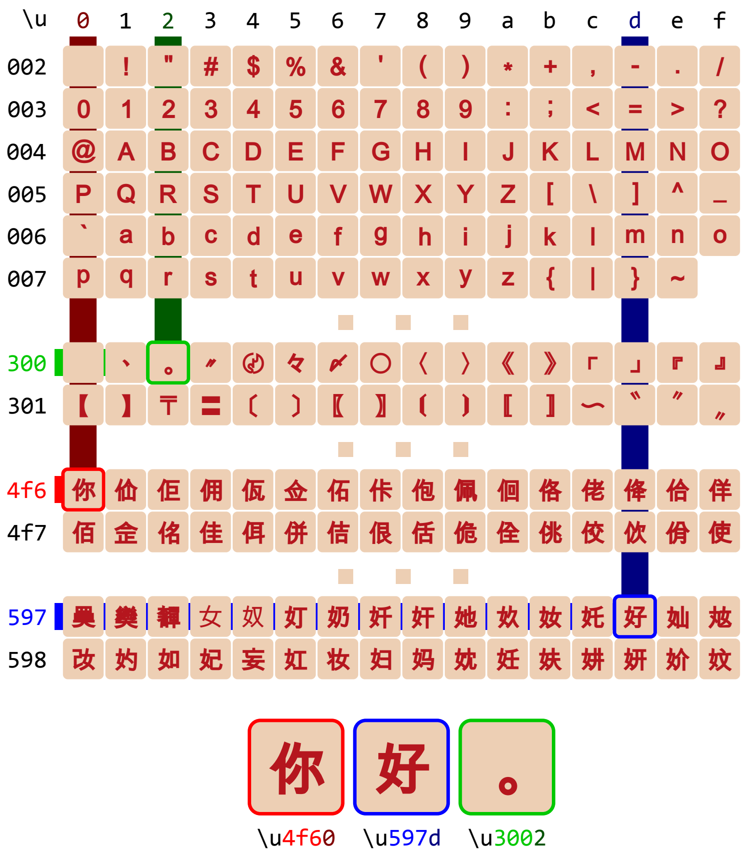


Figure 3.24: A subset of the Unicode character table including the Basic Lating characters as well as some Simplified Chinese characters.

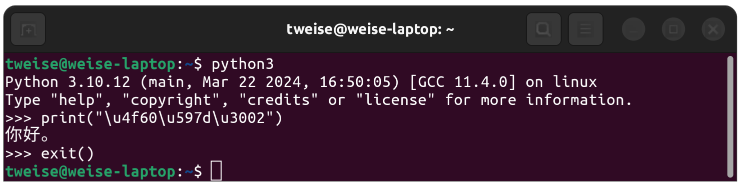


Figure 3.25: "Hello" in Simplified Chinese and entered via Unicode escaped string.

languages: Unicode [138, 284, 294], the most frequently used mapping of characters to numbers. Therefore, `Python` uses Unicode as well.

Figure 3.24 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. Most Unicode characters can be identified by a number represented as four hexadecimal digits (mentioned back in Section 3.2.2). The rows Figure 3.24 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 always 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 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 3.25, we use the information obtained in Figure 3.24 to print the Chinese text "Ni Hao." standing for "Hello." as unicode escaped string. We found that the character for "Ni" has unicode number 4f60, "Hao" has 597d, and the big period has 3002. The string `"\u4f60\u597d\u3002"` then corresponds to the correct Chinese text.

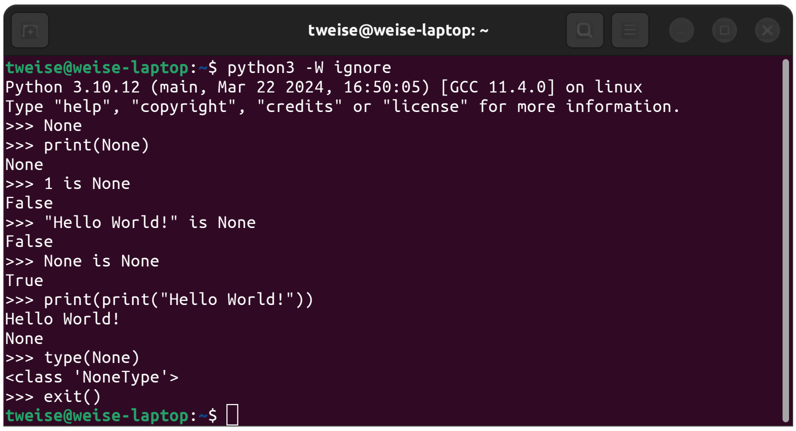


Figure 3.26: Examples of using the value `None`.

3.5.7 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 `ints`, `floats`, or `bools` 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`. And the function `bool`, well, you guess it.

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.

3.6 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.

Figure 3.26 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 Enter, 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`. And we can do this using the operator `is`.[6] `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.

3.7 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 arithmetics 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 absence of any value. Equipped with this knowledge, we now can embark to learn how to write programs that compute with these datatypes.

[6] `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.

Chapter 4

Variables

We did already learn 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

Like 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.

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

With this, we can now store intermediate results. This allows us, for the first time, two write programs that perform computations in multiple steps and that consist of multiple lines of code.

> **Best Practice 7** 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 [302].

Listing 4.1 shows the source code of such a 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 as well, as long as it does not contain spaces, e.g., `my_value`, `cow`, `race_car`. 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 `3` to the `stdout` of our program. We can also use `int_var` in f-strings about which we learned back in Section 3.5.2. `f"int_var has value {int_var}."` will render 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`. As sketched in Figure 4.1.2, 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. Doing `print(f"int_var is now {int_var}.")` will print `int_var is now 4.` to the `stdout`. The value `1` is now no longer referenced. Eventually, the `Python` interpreter could free the corresponding memory to use it for something else.

Ofcourse, 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

Listing 4.1: A Python program showing some examples for 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.
```

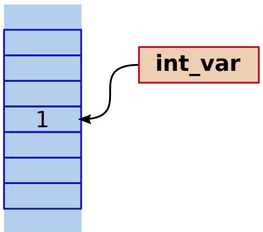
it, and lets `float_var` point to that piece of memory, as illustrated in Figure 4.1.3. We can use this variable in an f-string as well: `print(f"float_var has value {float_var}.")` is interpolated to `"float_var has value 3.5."`.

In a final step, we create a third variables with the name `new_var` by computing `new_var = float_var * int_var`. The result is `3.5 * 4`, i.e., `14.0`, `float` value. Figure 4.1.3 illustrates this variable assignment step. Finally, `print(f"new_var = {new_var}.")` then prints `new_var = 14.0.`.

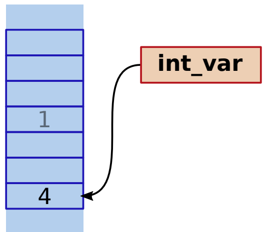
The standard output stream (stdout) of the complete program is given in Listing 4.2. For your convenience, we also showed the results when executing the program in PyCharm or the Ubuntu terminal in Figure 4.2. They are obviously identical.

Notice that we wrote the names of variables in a certain style, which is somewhat standard in Python programming. For the sake of creating readable code that fits nicely together with code from other projects. . .

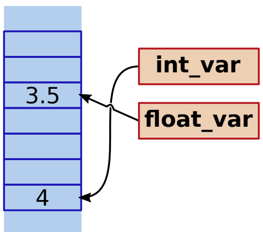
Best Practice 8 Variable names should be lowercase, with words separated by underscores [302].



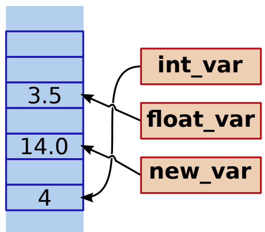
(4.1.1) `int_var = 1`



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

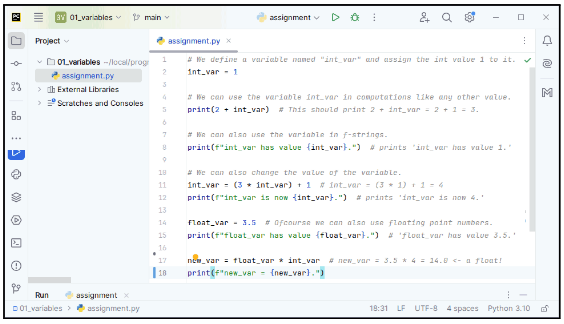


(4.1.3) `float_var = 3.5`

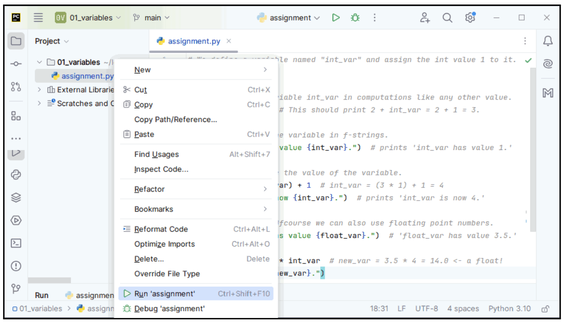


(4.1.4) `new_var = float_var * int_var`

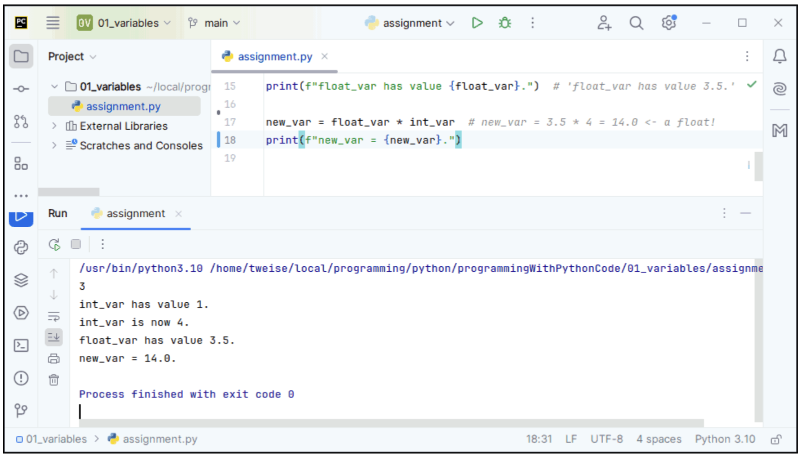
Figure 4.1: Illustrations of the variable assignments in Listing 4.1 in the same order in which they appear in the program: Variables are basically names that point to objects which are located somewhere in memory.



(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).

4.1.2 LIU Hui's Method and the Approximation of π

Let us now come to a more serious example. I am not good at mathematics, but I 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 transcendental, a never-ending and never-repeating sequence of digits. 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.

Well, we 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) somewhere in the third century AD [199] in his commentary to the famous Chinese mathematics book *Jiu Zhang Suanshu* (九章算术) [70, 73, 150, 199, 273]. In Figure 4.3, we show how π, i.e., the ratio of the circumference and the diameter of a circle can be approximated by inscribing regular e-gons into a circle. 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. 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 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 4.3. 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 th 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

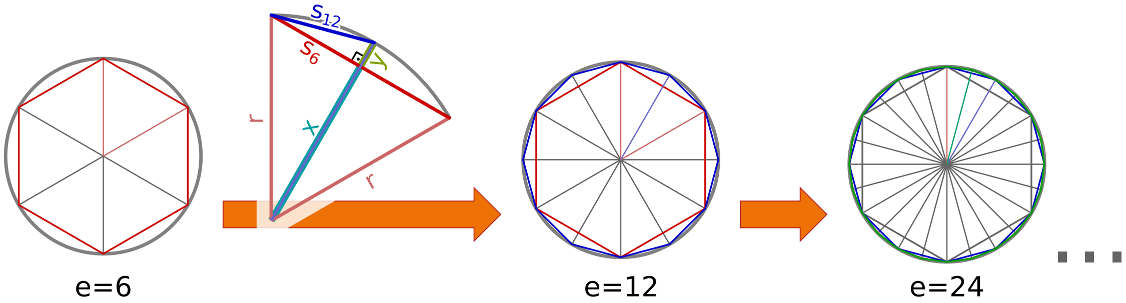


Figure 4.3: Approximating the ratio of the circumference and the diameter of a circle, i.e., π, by inscribing regular $3 * 2^n$-gons.

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

```
from math import pi, sqrt

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 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.
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.
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.
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.
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)))
print(f"{e} edges, side length={s} give us \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 us π≈3.0.
12 edges, side length=0.5176380902050416 give us π≈3.1058285412302498.
24 edges, side length=0.2610523844401031 give us π≈3.132628613281237.
48 edges, side length=0.13080625846028635 give us π≈3.139350203046872.
96 edges, side length=0.0654381656435527 give us π≈3.14103195089053.
192 edges, side length=0.03272346325297234 give us π≈3.1414524722853443.
```

calculations and observing Figure 4.3, we get the equation:

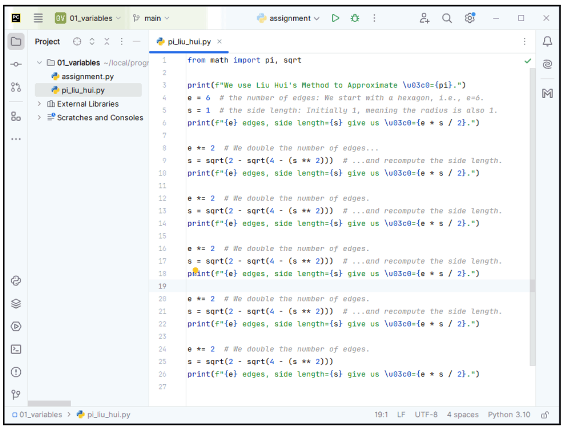
$$s_{2e} = \sqrt{2 - \sqrt{4 - s_e^2}} \tag{4.1}$$

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

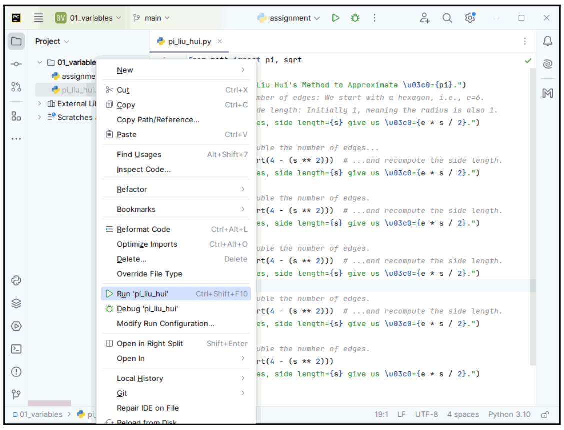
Now that we have learned some programming, we do no longer need to type the numbers and computation steps into a calculator. Instead, we can simply write them into a program, as illustrated in Listing 4.3. We begin by setting the number of edges `e = 6` and the side length to `s = 1`, still choosing $r = 1$. 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 escapes `\u03c0` and `\u2248`, respectively, from back in Section 3.5.6 (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`.

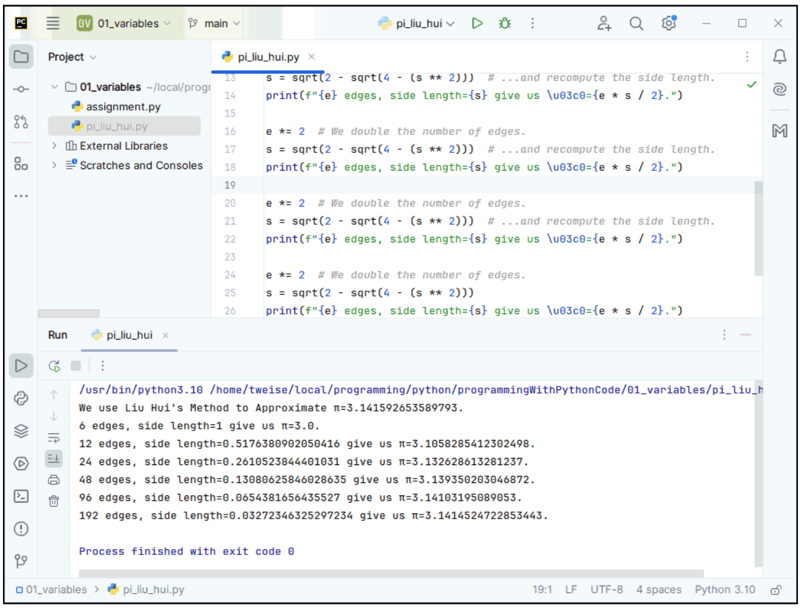
Given that the constant `pi` from the `math` module is `3.141592653589793`, we find that the first four digits are correct and that the number is only off by only 0.0045%! For your convenience, we also showed the results when executing the program in PyCharm or the Ubuntu terminal in Figure 4.4. They are obviously identical. 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.



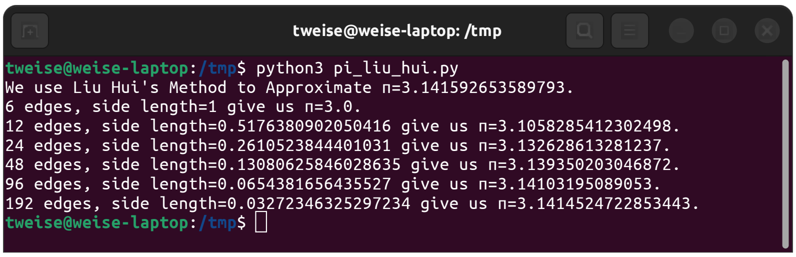
(4.4.1) The file `pi_liu_hui.py` opened in PyCharm.



(4.4.2) 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.



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



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

Figure 4.4: Running the program `pi_liu_hui.py` from Listing 4.1 in PyCharm (Figures 4.2.1 to 4.2.3) or the Ubuntu terminal (Figure 4.2.4).

4.2 Interlude: Finding Errors in your Code with the IDE

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. Every programmer sometimes makes a typo, accidentally switches the order of parameters of a function, stores a `float` in an `int` variable, and so on. Some errors are obvious and easy to fix. Some require more serious debugging (see Section 13.4). In many cases, however, our IDE can already show us what and where the mistake happened.

In Listing 4.5, 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 type in line 12 of the program: They misspelled `int_var` and `intvar`. Executing the program with the error leads to the output given in Listing 4.6.

The questions now are: How can we see this same error in our `PyCharm` IDE? Could we have found this error even without executing the program?

To answer these questions, we open the program `assignment_wrong.py` given in Listing 4.5 in the `PyCharm` IDE. We execute this program manually by clicking on the ▷ button or by pressing ⇧+F10 in Figure 4.5.1. As you can see, the output in the run window is the same as given in Listing 4.6 (Figure 4.5.2). Reading this output is the *first* way to find out what went wrong. 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 (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

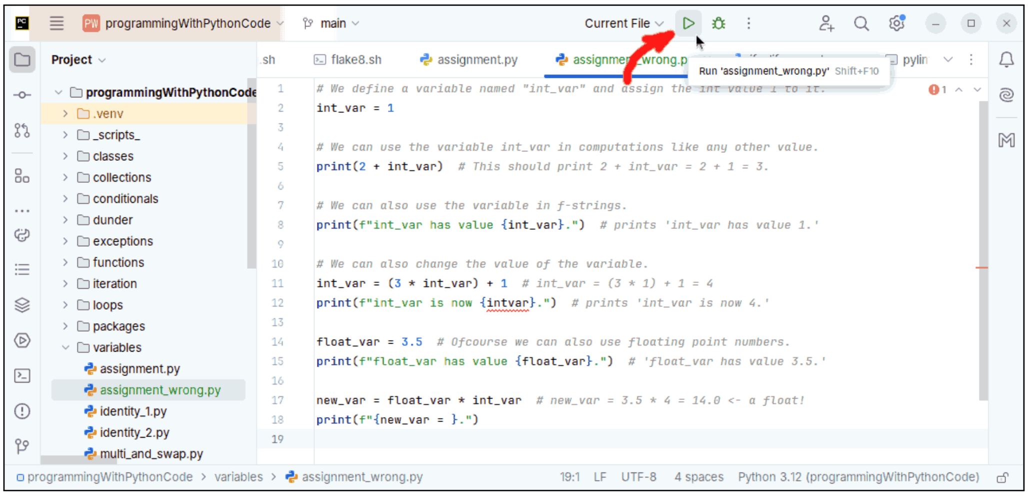
Listing 4.5: A variant of Listing 4.1 with an error: `int_var` is accidentally spelled as `intvar` in one location. (stored in file `assignment_wrong.py`; output in Listing 4.6)

```
1  # We define a variable named "int_var" and assign the int value 1 to it.
2  int_var = 1
3
4  # We can use the variable int_var in computations like any other value.
5  print(2 + int_var)  # This should print 2 + int_var = 2 + 1 = 3.
6
7  # We can also use the variable in f-strings.
8  print(f"int_var has value {int_var}.")  # prints 'int_var has value 1.'
9
10 # We can also change the value of the variable.
11 int_var = (3 * int_var) + 1  # int_var = (3 * 1) + 1 = 4
12 print(f"int_var is now {intvar}.")  # prints 'int_var is now 4.'
13
14 float_var = 3.5  # Ofcourse we can also use floating point numbers.
15 print(f"float_var has value {float_var}.")  # 'float_var has value 3.5.'
16
17 new_var = float_var * int_var  # new_var = 3.5 * 4 = 14.0 <- a float!
18 print(f"{new_var = }.")
```

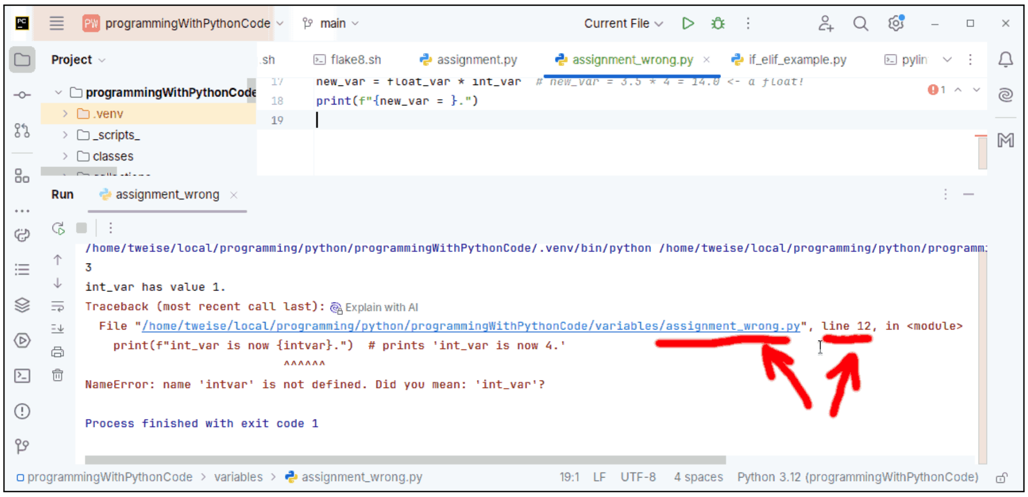
↓ `python3 assignment_wrong.py` ↓

Listing 4.6: The stdout and standard error stream (stderr) as well as the exit code of the program `assignment_wrong.py` given in Listing 4.5.

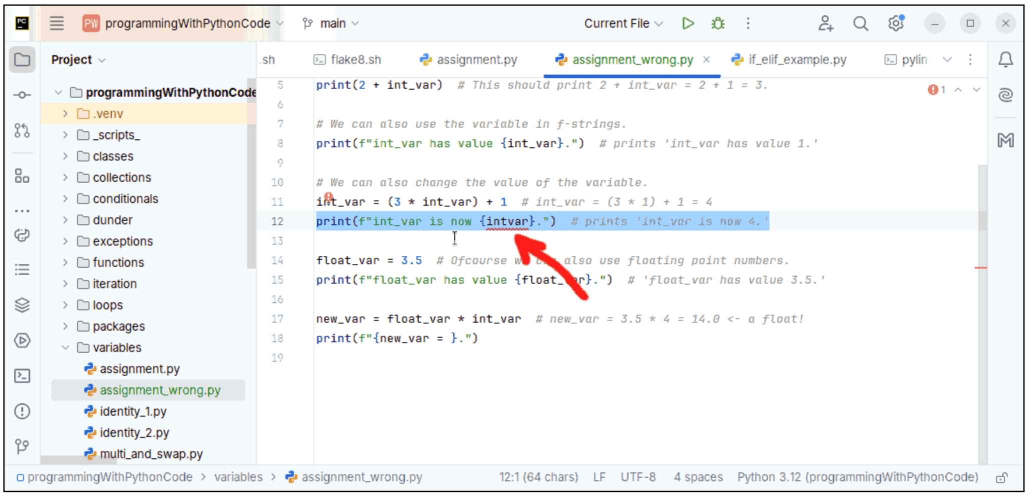
```
3
int_var has value 1.
Traceback (most recent call last):
  File "{...}/variables/assignment_wrong.py", line 12, in <module>
    print(f"int_var is now {intvar}.")  # prints 'int_var is now 4.'
                            ^^^^^^
NameError: name 'intvar' is not defined. Did you mean: 'int_var'?
#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



(4.5.1) We run the program `assignment_wrong.py` given in Listing 4.5 in the PyCharm IDE by clicking on the ▷ button or by pressing ⇧+F10.

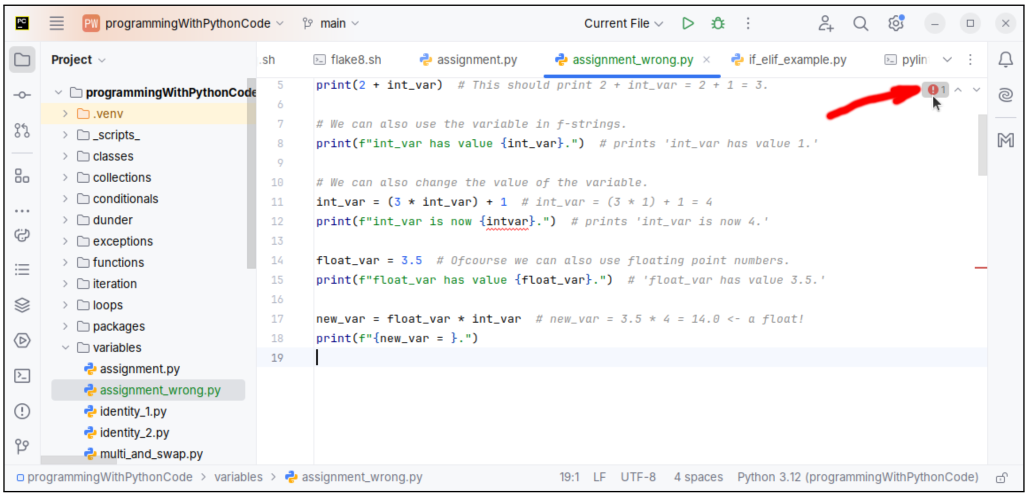


(4.5.2) The output in the run window is the same as given in Listing 4.6. It is the *first* way to find out what went wrong. It tells us what went wrong and even suggests how to fix it: *`NameError`: name '`intvar`' is not defined. Did you mean: '`int_var`'?* It also tells us the exact file and line where the error occurred, namely in line 12 of file `assignment_wrong.py`.

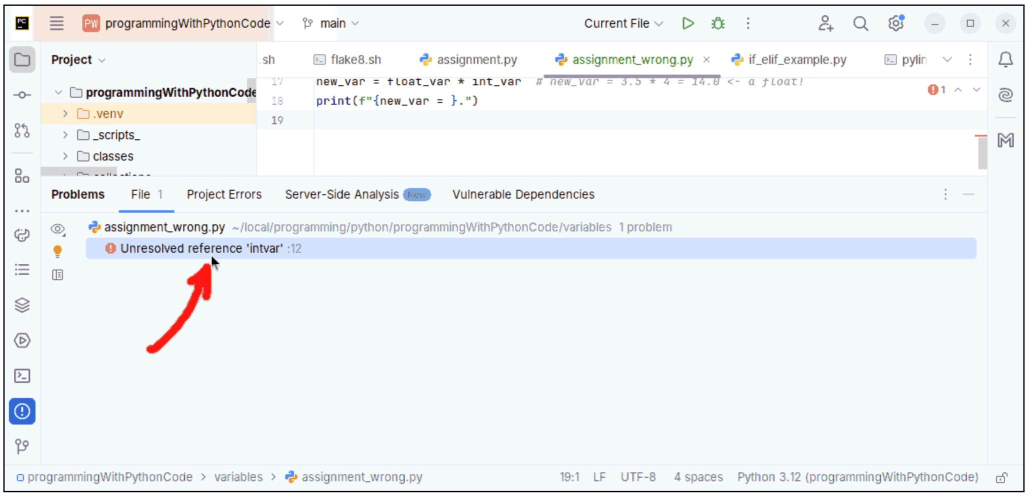


(4.5.3) If we click on the linked file location, it takes us to where the error occurred. The incorrectly typed word is (and always was) underlined with red color. Looking for underlined words is the *second* method to find errors in code!This should have told us already that something is fishy without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

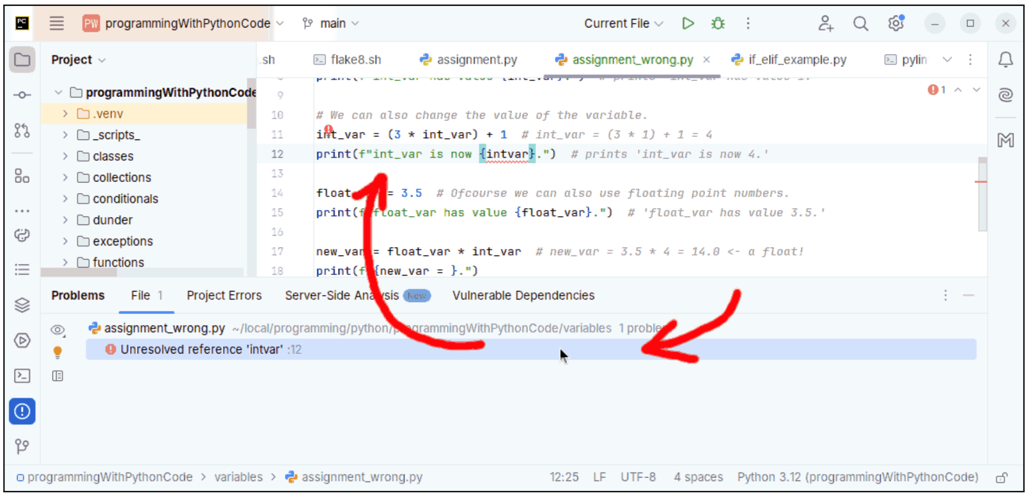
Figure 4.5: How the IDE can help us finding errors.



(4.5.4) 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. . .



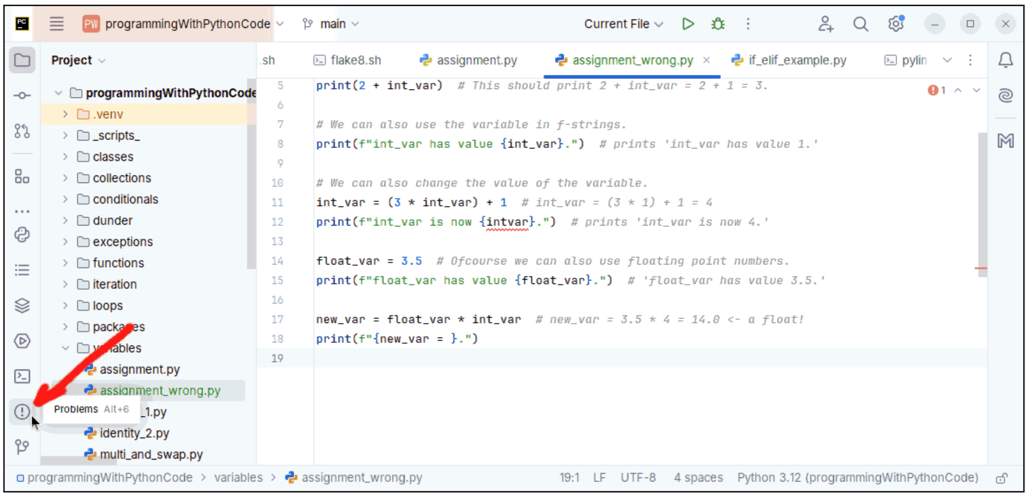
(4.5.5) . . . and it takes us to the list of potential errors that it has detected. Here, it tells us that there is an *Unresolved reference 'intvar'* at line 12 of the file. We click on this note. . .



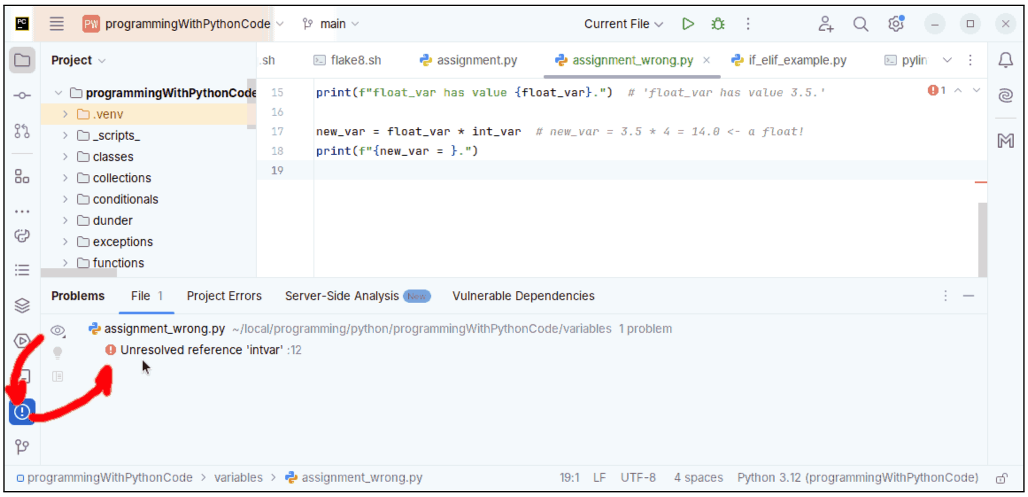
(4.5.6) . . . and it takes us again to the dodgy line.

Figure 4.5: How the IDE can help us finding errors.

(4.5.7) 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.



(4.5.8) 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.



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

Figure 4.5: How the IDE can help us finding errors.

of file `assignment_wrong.py`! With this information, we have a good chance of finding the mistake. The so-called `Exception` stack trace that it prints thus not just tells us the error, a probable cause, and the most-likely location 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 9 Always carefully *read* error messages. They often provide you very crucial information where to look for the mistake. Not reading error messages is wrong.

In the run console of `PyCharm`, we click on the linked file location. This takes us to where the error occurred in Figure 4.5.3. When looking at this line, 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 10 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 4.5.4. 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 4.5.5. 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 4.5.6

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 4.5.7. Holding the mouse cursor over these lines 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 4.5.8. This again takes us to the list of potential errors in Figure 4.5.9.

Useful Tool 1 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.

These tools make it much much easier to find errors. You can guess the importance of such features also by how many different ways `PyCharm` implements to get you to click and investigate its list of proposed errors and warnings. As mentioned in Best Practice 9 and 10, 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 cannot easily tell whether the output of a program is correct. And the output you see might actually 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 full 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 one hand, *you* might 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.

Listing 4.7: 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 4.8)

```
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 4.8: The stdout of the program `multi_and_swap.py` given in Listing 4.7.

```
a = 5, b = 10
a = 10, b = 5
x = 3, y = 2, z = 1
x = 1, y = 2, z = 3
```

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. In this case, we separate both the variable names and the values with commas. The first line (`a, b = 5, 10`) in Listing 4.7 assigns the values `5` and `10`, respectively, to the variables `a` and `b`, respectively. After this assignment step, `a == 5` and `b == 10` holds. `print(f"a=a, b=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 first storing `a` in a temporary variable `t`, then overwriting `a` with `b`, and finally copying the value of `t` to `t`. But it accomplishes this in single line of code instead of three. `print(f"a=a, b=b")` thus now prints *a=10, b=5*.

> **Best Practice 11** 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=x, y=y, z=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=x, y=y, z=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 in Chapter 3. We can obtain the type of an object stored in variable `var` by invoking `type(var)`. Listing 4.9 shows a program that does just that and the output of that program is given in Listing 4.9. 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.

Listing 4.9: An example of the types of variables. (stored in file `variable_types.py`; output in Listing 4.10)

```
int_var = 1 + 7  # Create an integer variable holding an integer number.
print(type(int_var))  # This prints "<class 'int'>".

float_var = 3.0  # 3.0 is a float and it is stored in float_var.
print(type(float_var))  # This prints "<class 'float'>".

str_var = f"f{float_var = }"  # Render an f-string into str_var.
print(type(str_var))  # This prints "<class 'str'>".

bool_var = (1 == 0)  # 1 == 0 is False, so a bool is stored in bool_var.
print(type(bool_var))  # This prints "<class 'bool'>".

none_var = None  # We create none_var which, well, holds None.
print(type(none_var))  # This prints "<class 'NoneType'>".
```

↓ `python3 variable_types.py` ↓

Listing 4.10: The stdout of the program `variable_types.py` given in Listing 4.9.

```
<class 'int'>
<class 'float'>
<class 'str'>
<class 'bool'>
<class 'NoneType'>
```

Listing 4.11: An example of the confusing variable types. (stored in file `variable_types_wrong.py`; output in Listing 4.12)

```
int_var = 1 + 7  # Create an integer variable holding an integer number.
print(type(int_var))  # This prints "<class 'int'>".

int_var = int_var / 3  # / returns a float, but we may expect an int?
print(type(int_var))  # This now prints "<class 'float'>".
```

↓ `python3 variable_types_wrong.py` ↓

Listing 4.12: The stdout of the program `variable_types_wrong.py` given in Listing 4.11.

```
<class 'int'>
<class 'float'>
```

4.4.2 Types and Confusion

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 permitted to only assign values that have exactly this type to the variable. In Python, you do not need to specify a type and you can assign whatever you want to a variable. This has the advantage that the code is shorter (because you do not need to write the type), looks more elegant, and programming becomes easier. At first glance. However, there are also problems. Let's take a look at Listing 4.11. 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`. This means that our variable `int_var` now contains a `float`, which is also visible in the output in Listing 4.12.

From the perspective of Python, this is totally fine. The program executes and the output appears without error. However, from the perspective of programming, Listing 4.11 is *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 find that a variable named `int_var` contains a `float`. This is not forbidden,

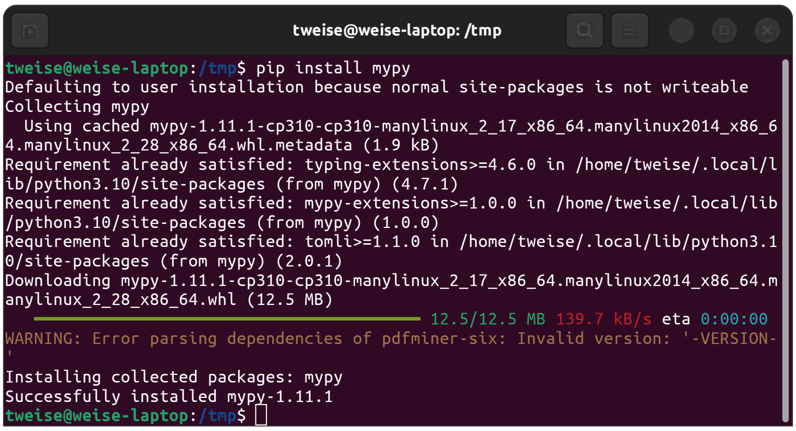


Figure 4.6: Installing `Mypy` in a `Ubuntu` terminal via `pip` (see Section 14.1 for a discussion of how packages can be installed).

but when reading the code, it must strike you as odd.

Indeed, there are at least two possible explanations for this: On the one hand, maybe, the original author of this code mistakenly mixed up the `/` operator for the `//` (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. Choosing this name, however, can be very dangerous: 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.

Best Practice 12 The names we use in program code should clearly reflect our intentions.

If the programmer followed Best Practice 12, then the code would clearly contain a bug, namely a mix-up between the `/` operator for the `//`. Regardless of what is true, 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 [151]. So here, the lenience of `Python` of allowing us to not specify types comes back to bite us.

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.

4.4.3 Static Type Checking

A first step to avoiding any type-related errors in programs is, ofcourse, careful programming. The second step is to use tools that check whether your program code contains ambiguities or errors. In languages like `C`, the compiler will take care of that for you.

In `Python`, which allows for dynamic typing and is an interpreted language, we will use a tool like `Mypy` [172]. 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 Enter. The `Mypy` tool will be installed as illustrated in Figure 4.6.

Listing 4.13: The results of static type checking with Mypy of the program `variable_types_wrong.py` given in Listing 4.11. (This is actually output generated by the script Listing 16.1 on page 327.)

```
$ mypy variable_types_wrong.py --no-strict-optional --check-untyped-defs
variable_types_wrong.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 1.15.0 failed with exit code 1.
```

Listing 4.14: The results of static type checking with Mypy of the program `variable_types.py` given in Listing 4.9. (This is actually output generated by the script Listing 16.1 on page 327.)

```
$ mypy variable_types.py --no-strict-optional --check-untyped-defs
Success: no issues found in 1 source file
# mypy 1.15.0 succeeded with exit code 0.
```

We can now apply the tool to the program from Listing 4.11. 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 4.13, 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. 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.

> **Useful Tool 2** Mypy [172] 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 4.6 on page 69. You can then apply Mypy using the command `mypy fileToScan.py`. We use the Bash script given in Listing 16.1 on page 327 to apply Mypy to the example programs in this book.

If we instead apply Mypy the completely fine (albeit useless) program Listing 4.9, it will tell us that there is no error, as illustrated in Listing 4.14. 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 what the programmer actually intended to do. But knowing that line 4 in Listing 4.11 is probably wrong will help the programmer to fix that error or oversight before passing the program on to someone else.

> **Best Practice 13** Every program should pass static type checking with tools such as Mypy (see Useful Tool 2). 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.

4.4.4 Type Hints

When we discussed Listing 4.11 in Section 4.4.2, we stated that there could be two reasons for the error in the code: Either, the author accidentally mixed-up two datatypes or operators (`/` vs. `//`) 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 find that line 4 four is probably wrong, because the variable `int_var`, which former contained an `int`, now gets a `float` value assigned to it.

Oddly enough, the problem of guessing the intention of the programmer does not exist in a statically typed language like pglsC. 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

Listing 4.15: Listing 4.11, but with the variable explicitly hinted as `int`. (stored in file `variable_types_wrong_hints_1.py`; output in Listing 4.16)

```python
int_var: int = 1 + 7  # Create a variable hinted as integer.
print(type(int_var))  # This prints "<class 'int'>".

int_var = int_var / 3  # / returns a float, but we may expect an int?
print(type(int_var))  # This now prints "<class 'float'>".
```

↓ `python3 variable_types_wrong_hints_1.py` ↓

Listing 4.16: The stdout of the program `variable_types_wrong_hints_1.py` given in Listing 4.15.

```
<class 'int'>
<class 'float'>
```

Listing 4.17: Listing 4.11, 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 4.18)

```python
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 4.18: The stdout of the program `variable_types_wrong_hints_2.py` given in Listing 4.17.

```
<class 'int'>
<class 'float'>
```

Listing 4.19: The results of static type checking the program `variable_types_wrong_hints_1.py` with Mypy of the program given in Listing 4.15.

```
$ 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 1.15.0 failed with exit code 1.
```

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 and could tell us that either line 4 is wrong or our initial assignment of an `int` to the variable in line 1.

This is where the dynamic typing and lenience 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 4.11. 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 [239, 301]. We can now declare the type of a variable if we want. This solves the above problem basically entirely and allows us to tell type checking tools our intention.

When we declare and assign a variable, `my_var = 1` for example, and want to define it as 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. And this makes a difference.

If the author of Listing 4.11 had used type hints, they could have written their program differently, as

Listing 4.20: The results of static type checking the program `variable_types_wrong_hints_2.py` with Mypy of the program given in Listing 4.17.

```
$ mypy variable_types_wrong_hints_2.py --no-strict-optional --check-untyped
   ↪ -defs
Success: no issues found in 1 source file
# mypy 1.15.0 succeeded with exit code 0.
```

illustrated in Listings 4.15 and 4.17. Both programs as well as the original one produce exactly the same output if we execute them with the Python interpreter, since type hints are ignored by the interpreter. However, if we type-check them with Mypy, we get two different results, namely Listings 4.19 and 4.20, respectively.

In Listing 4.15, 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 is in Listing 4.19 basically the same as it stated In Listing 4.13, namely that assigning a `float` to this variable is wrong.

However, if the intention of the author wanted 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 `ints` and `floats` are acceptable values for a variable [221].[1] This annotation is used in Listing 4.17. 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 4.20, which indicates that everything is OK.

Using type hints together with type checking would have prevented the problems in Listing 4.11 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 clearly communicated their intention by marking the variable to either hold an `int` or a `float`. 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 14 Always use type hints.

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 run-time.*
>
> — *Paul Jansen [142], 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 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 Listing 4.9 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"`, 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 4.21, which we can check with Mypy and obtain Listing 4.22.

It is very obvious at this point that including the type of the variable in the variable name is no needed. It is not helpful for any tool. If we want to convey our intention about the type to another

[1] Technically speaking, the type hint specification in PEP484 [301] 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 in some cases. Writing `int | float` gives me the opportunity to introduce the `|` operator ... so this is how we do it.

Listing 4.21: A variant of Listing 4.9 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 4.22: The results of static type checking with Mypy of the program `variable_types_hints.py` given in Listing 4.21.

```
$ mypy variable_types_hints.py --no-strict-optional --check-untyped-defs
Success: no issues found in 1 source file
# mypy 1.15.0 succeeded with exit code 0.
```

Listing 4.23: The results of static type checking the program `assignment_wrong.py` with Mypy given in Listing 4.5.

```
$ mypy assignment_wrong.py --no-strict-optional --check-untyped-defs
assignment_wrong.py:12: error: Name "intvar" is not defined  [name-defined]
Found 1 error in 1 file (checked 1 source file)
# mypy 1.15.0 failed with exit code 1.
```

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 14 *always* applies.

For the sake of completeness, we also apply Mypy to the program `assignment_wrong` given in Listing 4.5 that we used to illustrate the use of the PyCharm IDE in finding bugs. The output given in Listing 4.23 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.

You may ask why we emphasize that type hints are important and good for a programming language where they are originally not part of. Several important tools, like `psycopg` [305], the PostgreSQL Python adapter, are fully annotated with type hints based on the PEP484 [301] specification. Others use these annotations at least partially and/or try to ensure that code which is newly contributed to them is annotated, e.g., Matplotlib [134], NumPy [198], and Pandas [207]. The fact that many popular tools use it only *partially* use instead of being completely type-hinted is that they simply are older than PEP484 [301], which is from 2014. Scikit-learn and SciPy, for instance, to the best of our knowledge, do not adopt static typing at the time of this writing, because this would be very complicated with their existing codebases [4, 47]. The lesson we should learn from this is that

Best Practice 15 It is important to integrate type hints from the very start at 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 [312]. Injection attacks such as SQL injection attacks (SQL) 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 constants. Python supports the type `LiteralString` for string constants [268]. Implementations of the Python DB Application Programming Interface (API), such as `psycopg` [305], 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 [305, 333].

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.

And this can be done rather easily. Imagine that you buy a green jacket, let's call it A. Then you buy the exactly same jacket again, for a second time, and call the second jacket B. Now A and B are

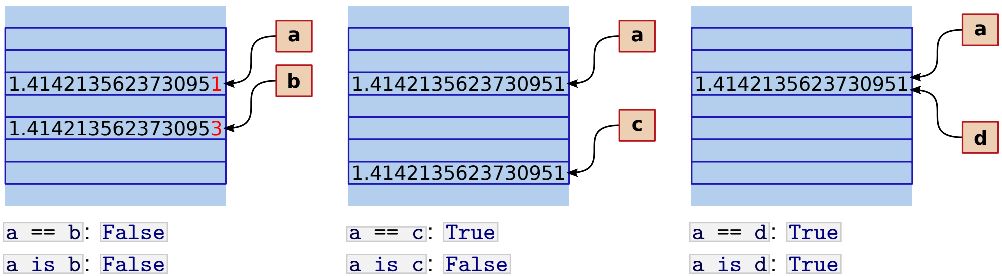


Figure 4.7: A illustration of the concepts of equality (`==`) and identity (`is`) from the perspective of variables.

Listing 4.24: An example of the difference between equality and identity. (stored in file `identity_1.py`; output in Listing 4.25)

```python
# 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 4.25: The stdout of the program `identity_1.py` given in Listing 4.24.

```
sqrt_2 = 1.4142135623730951
sqrt_2_computed = 1.4142135623730951
are they equal: True
are they the same object: False
```

Listing 4.26: An second example of the difference between equality and identity. (stored in file `identity_2.py`; output in Listing 4.27)

```
# 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 4.27: The stdout of the program `identity_2.py` given in Listing 4.26.

```
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
```

equal, but they are still two different objects.

Figure 4.7 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 Listing 4.24. 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 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 Listing 4.26. 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 4.26, 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 4.27 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.[2] It is obvious that we need small integer numbers again and again and again in our programs. Allocating a new objects whenever we use the values `1` or `0` would be wasteful. In Listing 4.26, 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.

4.6 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 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

[2] 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.

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 not depart from the interesting topic of variables.

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 variables to store instances (objects) of such datatypes. However, in many cases, we do not just want to store a single object. Often, we want to store and process *collections* of objects [42, 63, 72]. `Python` offers us four basic types of collections:

The first one, *lists*, are mutable sequences of objects. We can create a list `my_list` composed of the three strings `"a"`, `"b"`, and `"c"` by writing `my_list = ["a", "b", "c"]`. 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 `"a"`. 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 formed by *tuples*. 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 a tuple `my_tuple` containing 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 5.2.

The mathematical notion of sets is implemented in the `Python` datatype `set`. 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 5.3.

Finally, dictionaries 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. In other words, `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 5.4.

5.1 Lists

A `list` is a mutable sequence of objects which can be accessed via their index [248]. They work very similar to the strings we already discussed in Section 3.5, but instead of characters, they can contain any kind of objects and they can be modified.

5.1.1 Basic Functionality and Examples

In Listing 5.1, we provide some first examples for 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 hint `list[elementType]` where `elementType` is to be replaced with the type of the list elements [169]. `fruits: list[str] = ["apple", "pear", "orange"]` therefore

Listing 5.1: A first example for 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)

```python
"""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'].
```

creates the list `fruits` with the contents listed above. It also tells any automated type checking tool and other programmers that we intent that only `str` values should be stored inside the list.

The length of a list is 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 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

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)

```python
"""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 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 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.

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."""

l1: list[int] = [1, 2, 3, 4]  # create first list
l2: list[int] = [5, 6, 7]  # create second list
l3: list[int] = l1 + l2  # l3 = concatenation of l1 and l2.
print(f"l3 = l1 + l2 == {l3}")  # [1, 2, 3, 4, 5, 6, 7]

l4: list[int] = l2 * 3  # l4 = l2, repeated three times.
print(f"l4 = l2 * 3 == {l4}")  # [5, 6, 7, 5, 6, 7, 5, 6, 7]

l5: list[int] = l4[2:-2]  # l5 = l4 from index 2 to 3rd from end
print(f"l5 = l4[2:-2] == {l5}")  # [7, 5, 6, 7, 5]

l6: list[int] = l4[1::2]  # start at index 1, take every 2nd element
print(f"l6 = l4[1::2] == {l6}")  # [6, 5, 7, 6]

# Start copying l4 at last element, move backwards take every 2nd
# element, and stop right before index=3.
l7: list[int] = l4[-1:3:-2]
print(f"l7 = l4[-1:3:-2] == {l7}")  # [7, 5, 6]

l7[1] = 12  # Modify the slice l7 originally from l4.
print(f"{l4 = }, {l7 = }")  # Shows that l4 remains unchanged.

a, b, c = l2  # store the three elements of l2 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.

```
l3 = l1 + l2 == [1, 2, 3, 4, 5, 6, 7]
l4 = l2 * 3 == [5, 6, 7, 5, 6, 7, 5, 6, 7]
l5 = l4[2:-2] == [7, 5, 6, 7, 5]
l6 = l4[1::2] == [6, 5, 7, 6]
l7 = l4[-1:3:-2] == [7, 5, 6]
l4 = [5, 6, 7, 5, 6, 7, 5, 6, 7], l7 = [7, 12, 6]
a = 5, b = 6, c = 7
```

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 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 elements and indices start at 0, it returns `1`. Similarly, `numbers.index(7)` returns `3`, because `numbers[3] == 2`.

The `insert` method allows us insert an element at a specific index. The elements which are currently at that or higher indices are moved 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]`.

We can sort a list inplace 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 `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`.

In Listing 5.5, we continue 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]`.

In Section 3.5.1, we discussed string slicing. Lists can be sliced in pretty much the same way [248]. 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, the 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 `l4 = [5, 6, 7, 5, 6, 7, 5, 6, 7]`, then the slice `l4[2:-2]` will return a new list which contains all the elements of `l4` starting from index `2` and up to (excluding) the second-to-last element. The slice `l4[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 final example, consider the slice `l4[-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 `l7 = [7, 5, 6]`.

Notice that the slices we create are independent copies of ranges of the original lists. The list `l7` is a slice from the list `l4`. If we modify it, e.g., set `l7[1] = 12`, then we set the second element of `l7` to `12`. `l7` becomes `[7, 12, 6]`. Now, the second element of `l7` originally is the seventh element of `l4`, 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. `l4` remains unchanged by any operation on the independent copied slice `l7`.

An interesting functionality is also list unpacking. In Section 3.5.1, the list `l2` 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 = l2` creates and assigns values to three variables `a=5`, `b=6`, and `c=7` by unpacking the list `l2`.

5.1.2 An Example of Errors and a new Tool

Now, in the previous chapter, we learned that static code analysis tools can help us to discover subtle problems in our programs. Obviously, when dealing with more complex datastructures like lists, there are also more potential problems, more mistakes that one could make. Let us look at the very short example 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.

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 1.15.0 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).

However, upon closer inspection, we discover some issues. In a first step, we would apply `Mypy` (as Useful Tool 2) 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). As promised in the title of this section, we will also use another tool to analyze this program: `Ruff`.

> **Useful Tool 3** `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 [182]. It can be installed via `pip install ruff` as shown in Figure 5.1 on page 83. 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 329.

Let us apply `Ruff` to the program `lists_error.py` given in Listing 5.7, which produces the output Listing 5.10. `Ruff` finds two errors in this file: First, it complains that any python file should start with multi-line 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 5.10: The results of linting with Ruff of the program given in Listing 5.7. (We used the script given in Listing 16.2 on page 329 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
lists_error.py:1:1: D100 Missing docstring in public module
lists_error.py:1:22: C410 Unnecessary list literal passed to `list()` (
   ↪ remove the outer call to `list()`)
  |
1 | my_list: list[str] = list([1, 2, 3])
  |                      ^^^^^^^^^^^^^^^ C410
2 | print(my_list)
  |
  = help: Remove outer `list()` call

Found 2 errors.
No fixes available (1 hidden fix can be enabled with the `--unsafe-fixes`
   ↪ option).
# ruff 0.11.2 failed with exit code 1.
```

Best Practice 16 Each Python file should start with a string describing its purpose [108]. 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 `"""..."""` [108, 302].

Additionally, Ruff finds that writing `list([1, 2, 3])` is actually useless waste of speed and memory: It basically creates a list via `[1, 2, 3]` and then 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. 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.

Best Practice 17 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 where possible.

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]
```

Well, this was only a two-line program. But ask yourself: Did you spot the incorrect type hint when you read the program? 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.) Imagine that your job would be to work on a program with thousands of lines that was developed by a colleague. Wouldn't you love it if that colleague had thoroughly documented and type-hinted and checked their code? Be that colleague.

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 1.15.0 succeeded with exit code 0.
```

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 0.11.2 succeeded with exit code 0.
```

5.2 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, on a semantic level, lists are intended to hold objects of the same type. The type hint `list[int]` indicates that we want something to be 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. Since they cannot be changed, it will always be clear which element of which type is at which location. Three, tuples are defined using parentheses instead of square brackets, i.e., with `(...)`.

> **Best Practice 18** When you need to use an indexable sequence of objects, use a list if you intent to modify this sequence. If you do not intent 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]` [169]. 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"]`.

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.

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)

```python
"""An example of creating, indexing, and printing 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.

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()  # sort the list
print(f"sorted tuples list: {tuples}.")

a, b, c = mixed  # we unpack the tuple
print(f"a={a}, b={b}, c={c}")

mixed = "x", 4, 4.5
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)
```

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`.

In Listing 5.17 we explore tuples containing elements of multiple 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 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

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)

```python
"""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 and stderr as well as the 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 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 1.15.0 failed with exit code 1.
```

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. 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 the same, the comparison continues at the second element, 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 leave away 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])`.

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.

> **Best Practice 19** Only put immutable objects into tuples. Mutable objects inside tuples makes the tuples modifiable as well, while other programmers may assume that they are immutable. This can lead to strange errors down the line.

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.

5.3 Sets

Lists and tuples are classical sequence-based datastructures. Another classical datastructure 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, like tuples, 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 20** Only put immutable objects into sets.

Different from tuples, sets themselves are not immutable. Furthermore, sets are *unordered* collections [249]. Therefore, also different from tuples and lists, they cannot be indexed, i.e., the square-bracket notation does not work with sets.

> **Best Practice 21** 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 datastructures [66, 155, 257]. The operation for checking whether one object is contained in a set can be done in $\mathcal{O}(1)$ in average, whereas lists need $\mathcal{O}(n)$ in average, with n being the list length [6, 131, 194]. 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 linear in 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/or need to perform set operations such as those discussed in the following text, sets are the way to go.

In Listing 5.22, we provide a first example of creating and working with sets. Sets 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 [169]. The line `upper: set[str] = {"A", "G", "B", "T", "V"}` creates the variable `upper`, which points to 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.

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.

Listing 5.22: 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 21). (stored in file `sets_1.py`; output in Listing 5.23)

```
"""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 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 5.23: The stdout of the program `sets_1.py` given in Listing 5.22.

```
some uppercase letters are: {'T', 'B', 'V', 'A', 'G'}
some more uppercase letters are: {'T', 'B', 'V', 'A', 'G', 'Z'}
even uppercase letters are: {'Z', 'T', 'B', 'V', 'Q', 'A', 'G', 'K', 'W'}
some lowercase letters are: {'t', 'j', 'b', 'i', 'c'}
lowercase letters after deleting 'b': {'t', 'j', 'i', 'c'}
some letters are: {'Q', 't', 'A', 'G', 'Z', 'j', 'K', 'T', 'W', 'B', 'V', 'i', 'c'}
the sorted list of letters is: ['A', 'B', 'G', 'K', 'Q', 'T', 'V', 'W', 'Z', 'c', 'i', 'j', 't']
```

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 a sequence into the constructor `set`. To 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)`.

Listing 5.24: 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 21). (stored in file `sets_2.py`; output in Listing 5.25)

```python
"""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 5.25: The stdout of the program `sets_2.py` given in Listing 5.24.

```
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}
```

We can also convert the set `letters` to a list or tuple. `list(letters)` will achieve the former, `tuple(letters)` `tuple` the latter. However, since sets are unordered [249, 250] and the order in which the elements would appear in the created list or tuple is not defined. They may appear in a strange or unexpected order. Here we therefore use the function `sorted`, which accepts an arbitrary collection of objects as input and returns a sorted list. `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.

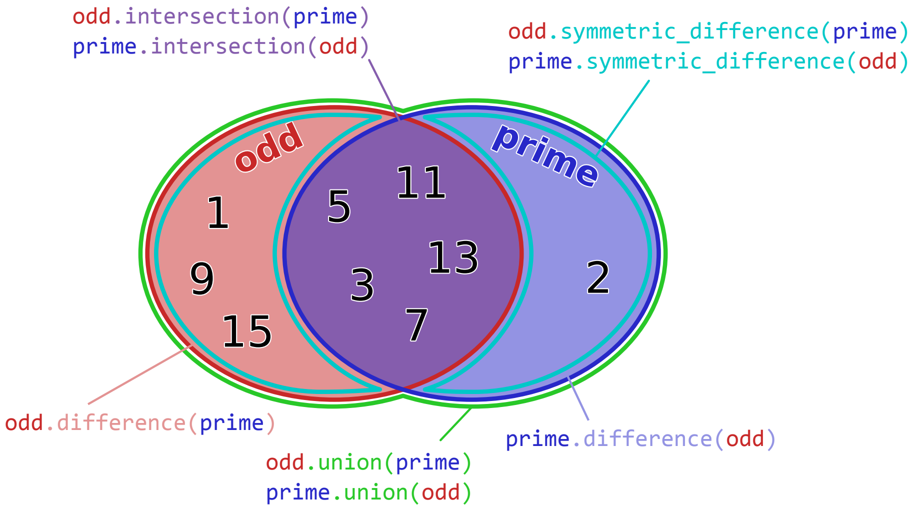


Figure 5.2: An illustration of the set operations in Python as used in Listing 5.24.

Best Practice 22 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 Listing 5.24 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 3.5.6). 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.

5.4 Dictionaries

Dictionaries in Python are containers 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

Listing 5.26: An example of using dictionaries in Python. (stored in file `dicts_1.py`; output in Listing 5.27)

```python
"""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 5.27: The stdout of the program `dicts_1.py` given in Listing 5.26.

```
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
```

languages or domains. Sets and dictionaries in Python are implemented using similar datastructures [6], namely hash tables [66, 155, 257], and thus exhibit similar performance.

Listing 5.26 shows 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 [169]. 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 5.26, 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 datastructures. Their elements appear in all sequenced versions always in the same order in which they were originally inserted into the dictionary [78].

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 obeye the same requirement as set elements:

> **Best Practice 23** Dictionary keys must be immutable.

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] = {}`.[1] 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.

5.5 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 datastructures that can store multiple values. You learned about mutable and immutable sequences of values, namely 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.

However, 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

[1] Using the `dict()` without arguments has the same effect, but several linters discourage this and encourage using `{}` instead.

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 3 (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 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 expressions that we already discussed back in Section 3.4. The syntax for this in Python is very simple:

```
if booleanExpression:
    conditional statement 1
    conditional statement 2
    # ...

normal statement 1
normal statement 2
# ...
```

The first line begins with `if` statement, 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 24 Blocks are indented by four spaces.

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 [280]. 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. Well, if that turns out to be `False`, the year could still be a leap year is it was divisible by 400. So the final condition to be checked is `(year % 400) == 0`. Now we piece all of this together using the logical `and` and `or` operators and get the condition `(((year % 4) == 0) and ((year % 100) != 0)) or ((year % 400) == 0)`.

All what is left to do is put an `if` in front of the condition and a `:` after it, and we got the conditional check. If it evaluates to `True`, we will print `f"{year}is a leap year."`. We copy this code

Listing 6.1: An example of using the `if` statement. (stored in file `if_example.py`; output in Listing 6.2)

```python
"""
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.
```

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 is 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 leads to confusion. So we are offered a better alternative: the `if ... else` statement, which looks like this:

```
if booleanExpression:
    conditional statement 1
    conditional statement 2
    # ...
else:
    alternative statement 1
    alternative statement 2
    # ...

normal statement 1
normal statement 2
# ...
```

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

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 will be executed as well.
else:  # Not leap year.
    print(f"{year} is not a leap year.")  # This line is 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.
else:  # Not 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 will, 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.
```

– is executed only if the Boolean expression evaluates to `True`. Then, the `else` line follows. Notice that 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`. 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, together with the fact that there can be multiple statements both in the `if` and the `else` block.

Well, if there can be multiple statements in these blocks and `if` and `if...else` are also both *statements*, then can we nest them? Can we place an `if` statement inside the body of another `if` statement?

Yes, we can, and we explore this in Listing 6.5. Here, we have three numbers `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 this 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, this case, either `b > a` or `b == a`, which means that it 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` and 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: `max` and `min`. 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)`.

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:
    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 maximum of a sequence
print(f"The minimum is {min(a, b, c)}.")  # min: get minimum of a sequence
```

↓ `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.
```

6.3 The if...elif...else Statement

In some cases, we need to query a sequence of alternatives in such a way that `else` blocks would be nested over `else` blocks over `else` blocks, and so on. Since everytime we nest a conditional statement into another one we have to add four spaces of indentation, 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` looks like this:

```
if booleanExpression 1:
    conditional 1 statement 1
    conditional 1 statement 2
    # ...
elif booleanExpression 2:  # such block can be placed arbitrarily often
    conditional 2 statement 1
    conditional 2 statement 2
    # ...
else:  # this, too, is optional
    alternative statement 1
    alternative statement 2
    # ...

normal statement 1
normal statement 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

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.
```

checked. And so on. Only if all the conditions of the `if` and all `elifs` 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` [221].) Anyway, we start with the first conditional and write `if age <= 3`. If this evaluates to `True`, we want to set `phase = "infancy"`, meaning that the person is a child in the earliest phase of life. If this is `True`, 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`. It 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 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 lift phase as `"middle childhood"`. 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.

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 3 back in Section 5.1.2. Of course, we should apply such tools to all of our code. If we apply `Ruff` to Listing 6.5, it will produce 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.

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
if_else_nested.py:12:1: PLR5501 [*] Use `elif` instead of `else` then `if`, to reduce indentation
   |
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.
   | |____^ PLR5501
14 |             print(f"{b} is the greatest number.")
15 |         else:  # This means that b >= a and c >= b.
   |
   = help: Convert to `elif`

Found 1 error.
[*] 1 fixable with the `--fix` option.
# ruff 0.11.2 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)

```python
"""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.
```

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 25 Prefer `elif` over nested `else ... if` constructs.

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

# Numbers with an absolute value less than ten are small.
# If their absolute value is larger than ten, they are large.
size: str
if abs(number) < 10:  # just a random threshold for this example...
    size = "small"
else:
    size = "large"

# Numbers can be positive, negative, or unsigned (0 is unsigned).
sign: str
if number < 0:
    sign = "negative"
elif number > 0:
    sign = "positive"
else:
    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.
```

6.4 The Inline `if...else` Statement

A very common use case of `if...else` statements is to 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 first store the number in an integer variable `number`. As 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:` and store the string `"small"` in the variable `size` if the condition evaluates to `True` and, otherwise, 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. 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`. Python offers us a more compact syntax for this, and Ruff can again give us some idea about that in Listing 6.14: The inline `if...then...else` statement, which has the following syntax:

```
variable = valueA if conditionForUsingValueA else valueB

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`

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
if_else_could_be_inline.py:8:1: SIM108 Use ternary operator ‘size = "small"
   ↪  if abs(number) < 10 else "large"‘ instead of ‘if‘-‘else‘-block
    |
 6 |    # If their absolute value is larger than ten, they are large.
 7 |    size: str
 8 | / if abs(number) < 10:  # just a random threshold for this example...
 9 | |      size = "small"
10 | | else:
11 | |      size = "large"
   | |__________________^ SIM108
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 0.11.2 failed with exit code 1.
```

evaluates to `True`. Otherwise, `valueB` is used if `conditionForUsingValueB` evaluates to `True` and, again otherwise, `valueC` is used.

Best Practice 26 When your `if...then...else` statement only assigns values to variables, use the inline variant discussed in Section 6.4, as it is more compact.

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.

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 A if the input meets a condition B and otherwise the result of a function C. This is already quite nice. For example, we can now implement and hard-code decision trees [242, 253] and Listing 6.7 is basically an example of that. Still, 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. . .

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

# Numbers with an absolute value less than ten are small.
# If their absolute value is larger than 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.
```

Chapter 7

Loops

When we are working with sequences of data, we do not just want to perform an action on one data element. We instead often want to apply the actions repetitively to all data elements. Loops allow us to do just that, to perform the same actions multiple times.

7.1 The `for` Loop Statement

The most basic such sequence in Python may be the `for` loop, which has the following pattern:

```
for loopVariable in sequence:
    loop body statement 1
    loop body statement 2
    # ...

normal statement 1
normal statement 2
# ...
```

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 in the following, indented block are executed for each of these values. 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.

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 3.5.1 and 5.1.1). `range(5)` will give us a sequence of integers starting with 0 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 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 Listing 7.1, we loop over exactly these ranges. In this listing, we try to create a dictionary (see Section 5.4) 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`.

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`, 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`.

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(4):  # i takes on the values 0, 1, 2, and 3 one by one.
    squares[i] = i * i  # Stores 0: 0, 1: 1, 2: 4, 3: 9.

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, 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!
```

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 27** If we do not care about the value of a variable (or parameter), we should name it `_` [157]. 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. However, nowhere in the loop body we care about the value of the loop variable. We thus simply call it `_`. 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 this program, we executed the same code five times. Instead of doing this, we can put this into a loop, which reduces the lines of code from over 25 to about 10 in Listing 7.3. The outputs in Listings 4.4 and 7.4 are exactly the same.

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 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 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 us π≈3.0.
12 edges, side length=0.5176380902050416 give us π≈3.1058285412302498.
24 edges, side length=0.2610523844401031 give us π≈3.132628613281237.
48 edges, side length=0.13080625846028635 give us π≈3.139350203046872.
96 edges, side length=0.0654381656435527 give us π≈3.14103195089053.
192 edges, side length=0.03272346325297234 give us π≈3.1414524722853443.
```

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 `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`. 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`. Instead of modifying the `range` over which we loop (which would be reasonable), we here want to use the `break` statement. We therefore wrap it into the conditional `if i > 10:`, meaning that it will only be executed if `i > 10`. As soon as `break` is reached, the loop will immediately be aborted. 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 by invoking the `continue` statement. This means that if `i == 5`, the `continue` statement lets the control directly return to the head of the loop, which will set `i = 6`. This will happen 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 we will never even reach this code 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 7.5: 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.
```

7.3 Nesting Loops

Like conditional statements, `for` loops can be arbitrarily nested. Let us explore this by computing the list of all prime numbers less than 200 in Listing 7.7.

Definition 4 (Prime Number) A prime number $p \in \mathbb{N}_1$ is a positive integer $p > 1$ (greater than one) that has no positive integer divisors other than 1 and p itself [67, 238, 317].

In our example program, we want to store all of these prime numbers 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 only containing the integer `2`. This will allow us to later on only consider the odd numbers in the range 3..200. 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..200, 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 and we know that 200 is not a prime number either, 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 set `is_prime = True` then will try to find a divisor, in which case we will set `is_prime = False`. If we cannot find a divisor of `candidate`, then `is_prime` will remain `True` and 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 `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 $a = b * c$ and $b > \sqrt{a}$, then it must be that $c < \sqrt{a}$. 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 c would also be a divisor of a that we would have found before reaching b in the loop since $c < \sqrt{a} < b$.

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

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)

```python
"""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 a 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].
```

square root of an integer number $\lfloor\sqrt{a}\rfloor$ can be computed using the `isqrt` function from the `math` module, which we `import` 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 <= 3`, then this loop will never be executed because no number `check` with $3 \leq$ `check` $<$ `candidate` exists. Then, `is_prime` will remain `True` and we will add `candidate` to `primes` further down the outer loop body. If `candidate > 3`, then `check` will go from `3`, $\dots$, $\lfloor\sqrt{\texttt{candidate}}\rfloor$.

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, because 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, if `is_prime` is still `True`, we add `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` itself. The output in Listing 7.8 tells us that, after performing 252 divisions, we could identify 46 prime numbers inside 2..200.

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

Listing 7.9: An example iterating over the elements in different collection datastructures 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={i}")  # We store "i=1", "i=2", "i=3", and "i=50"

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={f}")  # We store "f=7.6", "f=9.4", and "f=8.1"

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'", and "s='w'".

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='v', s='w', s='u', k = 1.1, k =
    ↪ 2.5, v = True, v = False, 1.1: True, 2.5: False
```

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={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={f}"`.

As third example, we create the set `st` containing the three strings `{"u", "v", "w"}`. We can do this again by simply writing `for s in st`, which 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 21), which 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 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 escape otherwise unprintable characters, see Section 3.5.4). Thus, we will add `"s='v'"`, `"s='u'"`, and `"s='w'"` to `txt` (in an arbitrary order).

As fourth and final example, we create a dictionary `dc` that maps floating point numbers to Boolean values, containing 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 datastructure `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 5.2, we learned about tuple unpacking. So instead, we write `for k, v in dc.items()`. This is a shortcut for writing `for t in dc.items()` followed by `k, v = t`. It directly unpacks the tuples in the sequence `dc.items`, meaning that we 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 to 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 `v` 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, f=9.4, ...`. 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 datastructures 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 the range `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.

Therefore, in our new and more efficient program Listing 7.11, we replace the head of the inner loop with `for check in primes[1:]:`. `primes[1:]` is a slice of the list `primes` containing all but the first element (see Section 5.1). This first element is `2`, and since all `candidates` are odd, we do not need to try whether they can be divided by `2`. The outer loop will step-by-step add prime numbers to `primes`. Actually, for each value of the loop variable `candidate`, `primes` will contain all prime numbers which are smaller than `candidate`. We only need to check those which are less than or equal to `isqrt(candidate)`, so 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.

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*.

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)

```python
"""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] = [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.
    limit: int = isqrt(candidate)  # Get the maximum possible divisor.

    for check in primes[1:]:  # 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 a 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_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].
```

7.5 Enumerating over Sequences when needed Indices

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 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`, because among the collection datastructures discussed so far, only `lists` are directly indexable. Therefore, the second option to achieve our goal would be to use an additional integer variable `i` which we would initialize as `i: int = 0` before the loop. We would then increment it as `i += 1` at the bottom of the loop body, where `i += 1` is equivalent to `i = i + 1`. This would work with all collection classes.

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.

Listing 7.13: Enumerating over the index-value pairs of a list (inefficiently). (stored in file `for_loop_no_enumerate.py`; output in Listing 7.14)

```
"""Enumerate the index-value pairs in a list."""

data: list[int] = [1, 2, 3, 5]
for i in range(len(data)):
    print(f"data[{i}]={data[i]}")
```

↓ `python3 for_loop_no_enumerate.py` ↓

Listing 7.14: The stdout of the program `for_loop_no_enumerate.py` given in Listing 7.13.

```
data[0]=1
data[1]=2
data[2]=3
data[3]=5
```

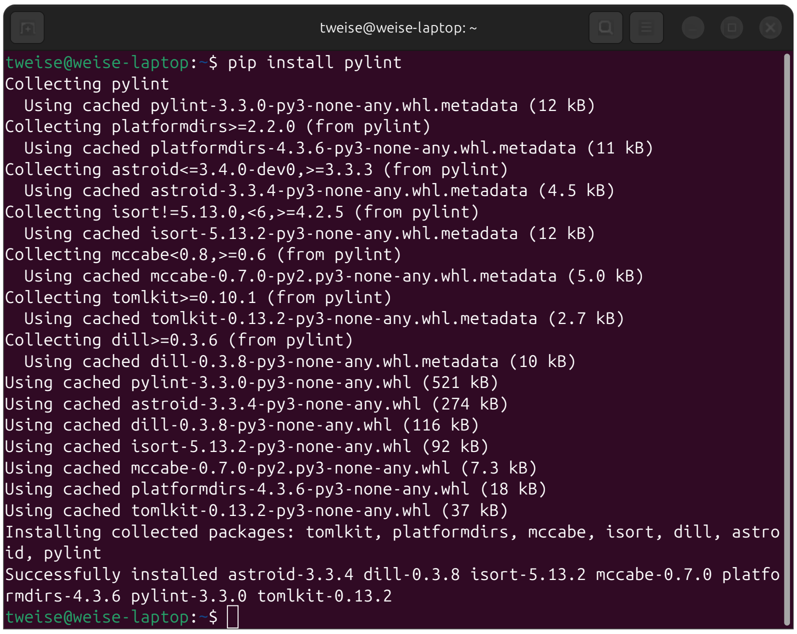


Figure 7.1: Installing `Pylint` in a `Ubuntu` terminal via pip (see Section 14.1 for a discussion of how packages can be installed).

Useful Tool 4 `Pylint` is a Python linter that analyzes code for style, potential errors, and possible improvements [226]. It can be installed via `pip install pylint` as shown in Figure 7.1 on page 115. 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 330.

In Listing 7.15, `Pylint` suggest that we should iterate over the list `data` using the `enumerate` function. `enumerate` accepts a collection as parameter and creates a sequence of index-value tuples. Since we can unpack such tuples while iterating over this sequence (see Listing 7.9), the head of our loop now takes on 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 Listing 7.16. And it would work exactly the same, regardless whether `data` was a list, set, tuple, or dictionary.

Now, above we used the `Pylint` linter to discover that we could improve our code by using this concept. At the time of this writing, `Ruff` and `Mypy` did not find this potential improvement. Maybe their future versions will. Regardless, it becomes clear that using more static analysis tools is always a good idea, as stated back in Best Practice 17.

Listing 7.15: The results of static code analysis using the Pylint linter for the program given in Listing 7.13.

```
$ pylint for_loop_no_enumerate.py --disable=C0103,C0302,C0325,R0801,R0901,
    ↪ R0902,R0903,R0911,R0912,R0913,R0914,R0915,R1702,R1728,W0212,W0238,
    ↪ W0703
************* Module for_loop_no_enumerate
for_loop_no_enumerate.py:4:0: C0200: Consider using enumerate instead of
    ↪ iterating with range and len (consider-using-enumerate)

-----------------------------------
Your code has been rated at 6.67/10

# pylint 3.3.6 failed with exit code 16.
```

Listing 7.16: Enumerating over the index-value pairs of a list using the `enumerate` function. (stored in file `for_loop_enumerate.py`; output in Listing 7.17)

```python
"""Enumerate the index-value pairs in a list using enumerate."""

data: list[int] = [1, 2, 3, 5]
for i, v in enumerate(data):
    print(f"data[{i}]={v}")
```

↓ `python3 for_loop_enumerate.py` ↓

Listing 7.17: The stdout of the program `for_loop_enumerate.py` given in Listing 7.16.

```
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 [100, 246]. The mathematician Hero(n) of Alexandria, who lived in the first century CE, specified an abstract algorithm for computing the square root of numbers which, today, is known as Heron's Method [158, 246].

Given a number a and aiming to find the square root $\sqrt{a}$ of a, this algorithm starts with a guess x_0, let's say $x_0 = 1$. In each iteration i, it will compute a new guess as follows [158, 246]:

$$x_{i+1} = \frac{1}{2}\left(x_i + \frac{a}{x_i}\right) \tag{7.1}$$

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 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 [246].

If we want to implement this algorithm, we will naturally need a loop of some sort. Clearly, we perform the same computation again and again. However, a `for` loop will not do: We do not know the number of steps that we will need in advance. Of course, we could try to pick a very very huge number and then `break` the loop when the guesses converg ... but that is just ugly. The `while` comes to rescue:

```
while␣booleanExpression:
␣␣␣␣loop␣body␣statement␣1
␣␣␣␣loop␣body␣statement␣2
␣␣␣␣#␣...

normal␣statement␣1
normal␣statement␣2
#␣...
```

In the `while` loop, the body 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 in Listing 7.18 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` be the current guess of what $\sqrt{\texttt{number}}$ could be and `old_guess` be the previous guess.

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`. Of course, if we could represent real numbers at infinite precision, we would never reach `guess == old_guess` for any `number` with an irrational square root. However, the `float` datatype has limited precision (see Section 3.3.1). Therefore, at least in our examples, `guess != old_guess` indeed eventually becomes `False`.

> **Best Practice 28** Do not use (un)equality comparisons of `floats` as loop termination criteria, as they may lead to endless loops. There can always be inputs that cause endless oscillations between values or the appearance of `nan` values (see Section 3.3.5).

The only reason why we did it here is that it looks nice and is easy to read as a functional and yet brief example of a `while` loop.[1] Anyway, the loop condition necessitates us to store some value different from `1.0` in `old_guess` initially (and we picked `0.0`). In the loop, first the current guess becomes the

[1] To enforce that no oscillation between two neighboring `float` values could occur, one could use the condition `abs(old_guess - guess)/ max(old_guess, guess)>= 1e-15` instead. This makes the loop stop when the relative difference between `guess` and `old_guess` becomes less than $10^{-}15$.

Listing 7.18: We compute the square root of a number using Heron's Method [158, 246] implemented as py `while` loop. (stored in file `while_loop_sqrt.py`; output in Listing 7.19)

```python
"""Using a `while` loop to implement Heron's Method."""

from math import sqrt  # math.sqrt is as exact with float as possible.

for number in [0.5, 2.0, 3.0]:  # The three numbers we want to test.
    guess: float = 1.0  # This will hold the current guess.
    old_guess: float = 0.0  # 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 = 0.5 * (guess + number / guess)  # The new guess.

    actual: float = sqrt(number)  # The "actual" computed sqrt value.
    print(f"\u221A{number}\u2248{guess}, sqrt({number})={actual}")
```

↓ `python3 while_loop_sqrt.py` ↓

Listing 7.19: The stdout of the program `while_loop_sqrt.py` given in Listing 7.18.

```
√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
```

old guess via `old_guess = guess`. Then we update the guess as specified in Equation 7.1, by setting `guess = 0.5 * (guess + number / guess)`.

Finally, we print the result of the computation, and 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 the Unicode escape method from Section 3.5.6 to represent the characters $\sqrt{\cdot}$ and $\approx$ as `\u221A` and `\u2248` to get them 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 *after* the loop if the loop has completed normally, i.e., if `break` was not invoked. We can do this by declaring a Boolean variable denoting whether the loop has completed normally before the loop and initializing it with `True`. If we invoke `break`, then we would first set this variable to `False`. After the loop, we could place an `if` to check if the variable is still `True` and then execute the action. This is totally fine, but Python offers us a much less verbose method: Using the `else` statement at the bottom of the loop, which works for both `for` and `while` loops:

```
for loopVariable in sequence:
    loop body statement 1
    loop body statement 2
    # ...
else:
    code executed if break not invoked 1
    # ...
normal statement 1
normal statement 2
# ...
```

```
while booleanExpression:
    loop body statement 1
    loop body statement 2
    # ...
else:
    code executed if break not invoked 1
    # ...
normal statement 1
normal statement 2
# ...
```

We now use this construct to implement a binary search [21, 117, 155]. Binary search works 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. Now, the whole sequence `data` and, hence, also the segment S, are sorted. If M is bigger than E, then E must be in the 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 must be in the upper half, i.e., in the sub-segment starting right after M and reaching until the end of S. Otherwise, we must have found the element. This means in one step we have effectively halved the size of S. If $n =$ `len(data)`, then we can do this at most $\log_2 n$ times and the time complexity of binary search is in $\mathcal{O}(\log n)$ [21, 117, 155].

In Listing 7.20, we want to find the indices of some characters in the alphabetically sorted string `data = "abdfjlmoqsuvwyz"`. Of course, there is the `find` method that can do this, but we want to take advantage of the fact that the characters in `data` are sorted. 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. To search them anyway, we let a variable `search` iterate over the list `["a", "c", "o", "p", "w", "z"]`.

Listing 7.20: We implement binary search [21, 117, 155] using a `while` loop with a `break` statement. (stored in file `while_loop_search.py`; output in Listing 7.21)

```
"""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.
    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 7.21: The stdout of the program `while_loop_search.py` given in Listing 7.20.

```
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'.
```

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 in which `search` could be contained. It is therefore initialized with `0`. `upper` is the exclusive upper end of the segment 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 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 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 is when the `else` statement at the bottom of the loop is executed. Then and

[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` [117].

only then we print that we did not find the string.

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 of 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!

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 output to the `sqrt` function from the `math` module which computes the square root. Now you will learn how to make your own functions.

8.1 Defining and Calling Functions

The syntax for defining our own functions in `Python` is as follows:

```
def my_function(param_1: type, param_2: type, ...) -> 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
normal statement 2
my_function(argument_1, argument_2)  # we can 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 29 Function names should be lower case, with underscores separating multiple words if need be [302].

Then follows an opening and a closing parenthesis, i.e., `(...)`. A function can have parameters through which we can pass values to it. Inside the function, these parameters act like variables. The values of these variables can be passed in when we call (invoke, execute) the function.

Definition 5 (Parameter) A function *parameter* is a variable defined inside the function that receives its value when the function is called.

Notice that, just like variables, all such parameters should be annotated with type hints (see Section 4.4) [324]. 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 is specified via the type hint `-> result_type`. We refer to the parameters and the return type of a function as its signature. The function header ends with a colon (`:`).

Best Practice 30 All parameters and the return value of a function should be annotated with type hints [324]. From my perspective: *A function without type hints is wrong.*

Then, indented by four spaces, follows the function body.

Best Practice 31 The body of a function is indented with four spaces.

This can be an arbitrary block of code, which may contain all the things we already learned. The function can, at any point, be left 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, ...)`, where `value_1` is passed in as value of `param_1`, `value_2` is passed in as value of `param_2`, and so on. This follows the same pattern of function calls that we already used in many of our examples.

Definition 6 (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 need to place a so-called docstring, which is a multi-line string (see Section 3.5.5). This string consists of a title line shortly describing what the function does. After an empty line, we can (but not necessarily need to) place a paragraph of text providing a more detailled 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 32 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 33 After the function and its body are defined, leave *two* blank lines before writing the next code [302].

After all of this long introduction, let us finally come to some example. Let's implement the factorial function as, well, function. The factorial is defined as follows [50, 84]

$$a! = \begin{cases} 1 & \text{if } a = 0 \\ \prod_{i=1}^{a} i & \text{otherwise, i.e., if } a > 0 \end{cases} \tag{8.1}$$

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. It should take a single parameter `a` as parameter. `a` will be type-hinted as integer and the result of our function will be an integer as well.

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 integers less than or equal to `a`. We want to multiply these values to `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

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)

```python
"""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.
```

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 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 `print`. 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. [180].

Functions can have a more than one parameter or no parameter at all. They can return one value or return nothing at all. Functions can also be called from other functions. 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 [36, 88, 102], going back to Euclid of Alexandria ($E\acute{\upsilon}\kappa\lambda\varepsilon\acute{\iota}\delta\eta\varsigma$) who flourished about 300 BCE. 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 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 $a = ig$ for some $i \in \mathbb{N}_1$ and $b = jg$ for some $j \in \mathbb{N}_1$. If we assume, without loss of generality, that $a > b$. Then, $c = a - b = (i - j)g$ and it will be clear that

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)

```python
"""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.
```

$c \bmod g = (a - b) \bmod g = (i - j)g \bmod g = 0$ as well, i.e., that `gcd`$(a, b) =$ `gcd`$(a - b, b) = g$. Similarly, $d = a \bmod b = ig \bmod (jg) = ig - \lfloor i/j \rfloor * jg = g(i - j\lfloor i/j \rfloor)$ is still divisible by g without remainder as $d \bmod g = 0$. This means that `gcd`$(a \bmod b, b) =$ `gcd`$(a, b) = g$, too.

Since both d and c are less than a, we could replace a with either of them. In particular, d will be less than both a and b, so we could store b in a and replace b with d. We would repeat the computation 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 this step. If $a = b$, then $a \bmod b = 0$ and we would immediately terminate after the first step and return a as the greatest common divisor.

This algorithm is implemented in Listing 8.3 as function `gcd`. `gcd` has two integer parameters, `a` and `b`, and returns another `int`. Its 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 tuple-unpacking command `a, b = b, a %b` (see Section 5.1). 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 `a` by `b` is stored in `b`. Clearly, `b` will become smaller in each iteration and since it can never become negative, it will eventually reach `0` and the loop will terminate. Similarly, the `gcd` is never "lost" during the loop and will thus be the value in `a` at the end. And this value is returned.

So this was a function with two parameters and one return value. Let us now implement a second function, this time with no return value. `print_gcd` accepts again two parameters `a` and `b` and returns nothing. Instead, it will print the `gcd` nicely using `print` and an f-string. Since the `math` module also provides a function names `gcd` for computing, well, the greatest common divisor, 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 [36].

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 7.18, and `factorial`, copied from Listing 8.1. (src)

```python
"""A module with mathematics routines."""


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 old_guess != guess:  # Repeat until nothing changes anymore.
        old_guess = guess  # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

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. And we can distribute code over multiple files. This clearly is a concern for any application larger than a simple script: 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* [281].

For all intents and purposes within this book, a *module* is a Python file and a *package* is a directory wherein the file is located. As described in [281], 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 7.18, 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.

Best Practice 34 Package and module names should be short and lowercase. Underscores can be used to improve readability. [302]

Our new module `my_math` 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.

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 [139] (we will eventually discuss this later). The `my_math` module is found because it is in the same directory as the program `use_my_math`.

If we had 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` instead, 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 modul 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 LIU Hui's 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.

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)

```python
"""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.
```

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.

But dividing code into functions and placing it into modules has yet another advantage: We can test these separate functions separately. Imagine that you have a big application using lots of components, code for reading and writing files, code for making mathematical computations, and so on. It is not eay to figure out how to check that the big application behaves exactly as it is supposed to. There will be many combinations of inputs and environments and conditions that we would need to test. Even if we would find that the application behaves, maybe only in some cases, a bit strange, it would be hard to figure out which of its many components has an error.

However, we can also do something else: We can test each component separately, too. 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 7 (Unit Testing) *Unit Testing* is a software testing technique where separate components or functions of an application are tested in isolation.

The goal of unit testing is to ensure that each unit of the software performs as expected. Since tests can be developed along (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 testing is especially useful if an application is

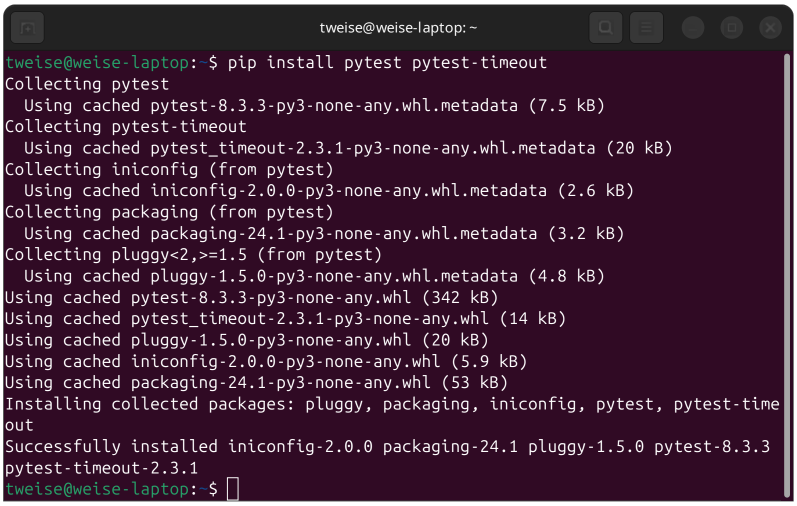


Figure 8.1: Installing `pytest` in a `Ubuntu` terminal via `pip` (see Section 14.1 for a discussion of how packages can be installed).

developed, improved, and maintained over a long time: If test cases are preserved and re-tested every time the software changes, then we can discover if a change that we applied to an older piece of code breaks a unit test, i.e., may lead to an unwanted change of behavior our program that we did not anticipate. Especially with its increased automation, unit testing in software development has steadily gained importance during the past decades [241, 286, 320] and is an important cornerstone of `Python` software development [77, 203, 206].

> **Useful Tool 5** `pytest` is a `Python` is a `Python` framework for writing and executing software tests [160]. It can be installed via `pip install pytest pytest-timeout` as shown in Figure 8.1 on page 128. 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 331. See also Useful Tool 7 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 same of clarity, if the actual code is in a file `my_code.py`, then we could the code for the tests into a file called `test_my_code.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:

```
assert booleanExpr  # raises AssertionError if not booleanExpr
```

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 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 fails or 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 $0! = 1$. So it makes sense to define `assert factorial(0)== 1`. The Boolean expression

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! == 4790'016'00
    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.
```

here is `factorial(0)== 1`, which is obviously only `True` if `factorial(0)` evaluates to `1`. We 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. 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!. We have computed this value using an online tool that offers arbitrary precision. The result of this function 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 expect 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, which both can be imported from the `math` module and which were discussed in Section 3.3.5: `inf` stands for positive infinity $+\infty$ and `nan` stands basically for "undefined", a value 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` for any value 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.

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:28: in sqrt
    guess = 0.5 * (guess + number / guess)  # The new guess.
E   ZeroDivisionError: float division by zero
======================== short test summary info =======================
FAILED test_my_math.py::test_sqrt - ZeroDivisionError: float division by
   ↪ zero
====================== 1 failed, 1 passed in 0.03s =====================
# pytest 8.3.5 with pytest-timeout 2.3.1 failed with exit code 1.
```

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.

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)`.

When we look back at our `my_math` module in Listing 8.6, we find that our initial guess for the square root is `1`. 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`. 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, and `guess` will thus eventually become `0.0`. In the next iteration, 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 Listing 8.11. If this new conditional evaluates 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. Here, we would ideally `raise` an error by ourselves, but we will learn only later how to do that.[1]

We apply the same tests to the new version of the `sqrt` function in Listing 8.11. The output of the test case, provided in Listing 8.12, now indicates another error: We a timeout!

You see, when invoking pytest with the option `--timeout=10` – which only works if the package `pytest-timeout` is installed – we limit 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.

> **Best Practice 35** 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 continuous integration systems from clogging.

The time limit protected us here. Our `sqrt` function goes into an endless loop if `number = inf`. Initially, we set `guess = 1.0`. In the first iteration of the loop in `sqrt`, `guess = 0.5 * (guess + number / guess)` makes `guess` become `inf`. However, in the second iteration, we have `number / guess` becoming `inf / inf` which yields `nan`. From now on, all calculations yields `nan`. Since `nan != nan` is `True`, the loop never ends.

[1] We learn it in Chapter 9 and there we will revisit our `sqrt` function in Listing 9.1, too.

Listing 8.10: An improved variant of Listing 8.5 dealing with the failing test case `0.0` discovered in Listing 8.9. (src)

```python
"""A second version of our module with mathematics routines."""

# 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 old_guess != guess:  # Repeat until nothing changes anymore.
        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)

```python
"""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.
```

We finally solve this problem 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 and `False` if it is `inf`, `-inf`, or `nan`. This means that condition can only become `True` 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 `nan` if the input is `nan`. 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 certain assumed that all of our functions were implemented correctly. However, `pytest` has helped us to spot two erros. We then fixed these errors.

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:18: in test_sqrt
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
my_math_2.py:18: in sqrt
    while old_guess != guess:  # Repeat until nothing changes anymore.
E   Failed: Timeout >10.0s
======================== short test summary info =========================
FAILED test_my_math_2.py::test_sqrt - Failed: Timeout >10.0s
========================== 1 failed in 10.02s ============================
# pytest 8.3.5 with pytest-timeout 2.3.1 failed with exit code 1.
```

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 isfinite

# 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 old_guess != guess:  # Repeat until nothing changes anymore.
        old_guess = guess  # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Best Practice 36 A function which is not unit tested is *wrong*.

Best Practice 37 A good unit test 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.

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  # 1e10² = 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 8.3.5 with pytest-timeout 2.3.1 succeeded with exit code 0.
```

Best Practice 38 A good unit test 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 testing is. 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` [128]. 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, I/O errors, and out-of-memory errors. Due to its reliability, `SQLite` is often used in mission-critical applications such as flight software [127]
>
> — *Kevin P. Gaffney, Martin Prammer, Laurence C. Brasfield, D. Richard Hipp, Dan R. Kennedy, and Jignesh M. Patel [103], 2022*

`SQLite` is the most used Structured Query Language (SQL) DB in the world. It is installed in nearly every smartphone, computer, web browser, television, and automobile [51, 103, 323]. And its core developers mark reliability, shown by thorough tests, as one of the four reasons for that.

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
```

8.4 Function Arguments: Default Values, Passing them by Name, and Constructing them

After the brief introduction into unit testing, let us now come to a lighter topic: passing arguments to functions.We have already seen examples for this. Our `gcd` function from back in Listing 8.3 has two parameters `a` and `b` and we can invoke it by writing the values of these parameters in parentheses. `gcd(12, 4)` will invoke `gcd` and assign `12` to `a` and `4` to `b`.

We can also let parameters have so-called *default values*. If a parameter has a default value, then we can either specify the value of the parameter when calling the function *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 probability density function (PDF) of the normal distribution. You may 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 arithmetic mean, i.e., the expected value, of the distribution and σ is its standard deviation (making σ^2 its variance). x is the variable of this function. This function describes typical bell-shaped curve of the normal distribution as sketched in Figure 8.2. Implementing this function as a, well, function in Python is straightforward. Listing 8.16 offers the function `pdf` with three parameters: `x`, `mu`, and `sigma`, which represent x, μ, and σ.

Now, the two parameters μ and σ of f (respectively `mu` and `sigma` of `pdf`) 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. 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 write `x: float, mu: float = 0.0, sigma: float = 1.0`.

In Listing 8.17, we import our new `pdf` function and use it inside a program. 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)` if 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.

Best Practice 39 Default parameter values must always be immutable.

The default value of a function must always be immutable. 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.

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
```

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.

Back to business. 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 default value, which 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 provides also another interesting way to call a function in Python. As we have established by now, the parameters of a function have names. If we write something like `mu=8.0, x=0.0, sigma=1.5` to assign arguments, this looks very similar to the way we created dictionary constants back in Section 5.4 and Listing 5.26. Calling `pdf(-2.0, sigma=3.0)` is equivalent to writing `pdf(x=-2.0, sigma=3.0)`.

We can create a dictionary with the values `{"x": -2.0, "sigma": 3.0}`. Let's call this dictionary `args_dict`. Can we now somehow pass in these values to `pdf`? We indeed 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 `*` (called wildcard, star, or asterisk) *before* the dictionary, i.e., the `**` is telling Python to unpack the dictionary this way. Second, default argument values still apply here, i.e., `mu` will have value `0.0` in this function call.

Similarly, maybe we do not care about the parameter names but want pass them in by position,

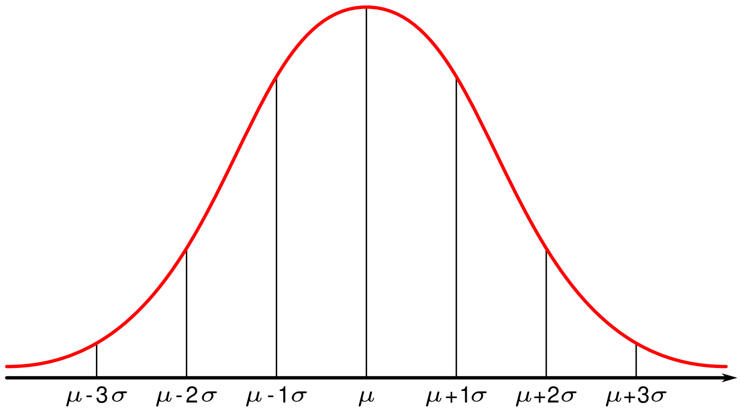


Figure 8.2: A sketch of the probability density function of the normal distribution given in Equation 8.2.

as we have always done 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, invoking `pdf(*args_tuple)` will basically fill in the three values in their into the parameters, i.e., will be equivalent to `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 a sequence 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` 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 the parameters in a dictionary and in the `if` branch just add the eleventh parameter if need be, your code will become much simpler.

8.5 Functions as Parameters or Variables, `Callable`, and `lambdas`

We have 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. References to objects can be stored by variables or passed in as function arguments. Functions are objects too. This means that we can also store functions in variables or pass them as argument 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 have learned about integration and differentiation. A *definite integral* is the

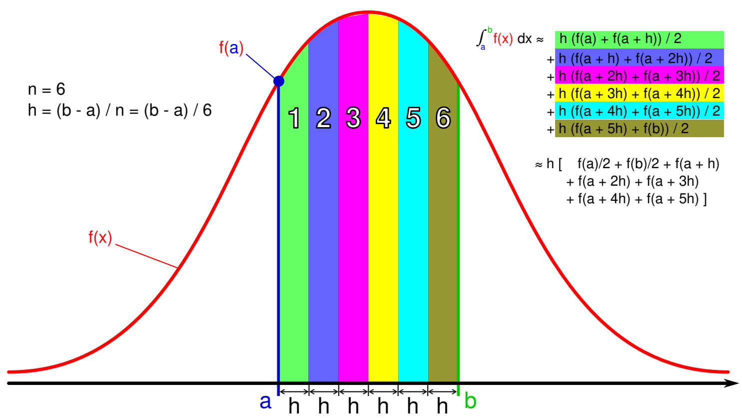


Figure 8.3: A sketch of the trapezoid method for approximating definite integrals.

formal calculation of the area under a function, using infinitesimal stripes of the region. The second fundamental theorem of calculus states: 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 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. If we have a function $f(x)$, which would obviously be a parameter of our program, as well as the interval limits a and b, we could divide the range $[a, b]$ into n strips. For each strip, we could approximate the area beneath $f(x)$, 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).

The trapezoid method is one very simple implementation of this idea [87]. As illustrated in Figure 8.3, it treats the strips as right trapezoids. The baseline of the trapezoid 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 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 8.3.

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. Our implementation `integrate` of this approach, given in Listing 8.19, does exactly that.

The interesting part of our `integrate` method, however, is not the approximation of the integral. It is the 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 [8]. Like the type hint for lists and tuples, it can be parameterized following the scheme [186]:

```
Callable[[parameterType1, parameterType2, ...], resultType]
```

In other words, inside the `Callable[...]`, we first provide the list of parameter types, then a comma `,` followed by the return type. `f: Callable[[float], float |int]` thus states that our function expects, as first parameter, *another function* `f`. `f` must accept one parameter of type `float`. Its return type is `float |int`, i.e., it should return either a `float` or `int` [221]. Notice that the type `Callable` is provided by the module `typing`[2], so we need to import it first if we want to use it.

[2][8] states that this import is now deprecated and we should use `collections.abc.Callable` instead. In the

Listing 8.19: Using the trapezoid method to numerically approximate definite integrals, as sketched in Figure 8.3. (stored in file `integral.py`; output in Listing 8.20)

```python
"""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
```

Additionally, our `integrate` function accepts two parameters `a` and `b` which must be of type `float | int`, i.e., can either be an `float` or `int`. Their default values are `0.0` and `1.0`, meaning that we will integrate over $[0, 1]$ if they are not specified. As final parameter, we expect an integer `n`, by default `100`, which corresponds to the number of trapezoids that we shall use for the approximation.

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:

```python
def const_1(x: float) -> float:
    return 1.0
```

or

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 in Listing 8.19.

```
$ mypy integral.py --no-strict-optional --check-untyped-defs
Success: no issues found in 1 source file
# mypy 1.15.0 succeeded with exit code 0.
```

```
def const_1(_: float) -> float:
    return 1.0
```

where the `_` indicates that we are actually going to ignore this parameter (see Best Practice 27). 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` [164], which have the structure

```
lambda param1, param2, ...: 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. Either way, since we need one parameter for our 1-returning-function and since we do not care about the value of this parameter, we can simply write:

```
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 1 over the range $[0, 1]$ is also 1. 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 (and we use the unicode escape `"\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 method returns `-3.3332`, which is fairly close to $-3.\overline{3}$.

Let us now integrate the sinus 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.

Let us, as final example, also integrate the probability density function (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 values are used, the function computes the PDF of the *standard* normal distribution.

Interestingly, although we demand that the function supplied to `integrate` should only have one parameter, we can still pass in `pdf` due to its second and third parameter being 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 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%.

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%. 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, like Simpson's rule, are much better [87]. Still, as an example of functions that work on other functions, it was suitable.

8.6 Summary

Functions are the central building block 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 code with clearly defined input and output that exists in isolation from other code, then we can *test* this code in isolation as well. 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 unit tested these two functions and 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 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 testing 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 3.5.1. The attempt to modify an element of a `tuple` is punished with a `TypeError` in Section 5.2. 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) [215]. 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 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` [105, 185, 252].

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 an `TypeError`, for example. Personally, I am also a fan of this approach.

If we would 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 will lead to some crash down the line. 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.

Input sanitization could cover some error that happened earlier. This could even cause more errors, because it will allow other programmers to call our code with wrong values. Input sanitization encourages sloppy programming.

This leaves raising an `Exception`. But what does that actually mean? Raising an `Exception` means two things:

1. We store information about the error and information about the current execution state (current line of code and the function call hierarchy) 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 handles 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 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, this the raised exception takes down 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?" 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 40 Errors should not be ignored and input should be sanitized. Instead, the input of our functions should be checked for validity (where reasonable). `Exceptions` should be raised as early as possible if any unexpected situation occurs.

9.2 Raising Exceptions

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 that `sqrt` we implemented back in Listing 8.10 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 to pass 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 that a square root is. Or the negative input came as the result from 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 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, returning `0.0` is not a good idea.

Instead, we want to signal this error explicitly. Our function should raise an exception. We therefore create a new implementation of `sqrt` in Listing 9.1. Before writing the actual `Python` code, we will express the new behavior in the docstring of our function:

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)

```python
"""A `sqrt` function raising an error if its result is not finite."""

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; 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 old_guess != guess:  # Repeat until nothing changes anymore.
        old_guess = guess  # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess
```

Best Practice 41 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.

In our case, any input x for which $\sqrt{x}$ would be either undefined or not finite should lead to an error. Since this error is of arithmetic nature, we will raise an `ArithmeticError`. 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.

In the new `sqrt` implementation Listing 9.1 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 either `inf`, `-inf`, or `nan`. The complete expression becomes `True` if the input of the function is not finite or negative. We thus check whether the output of the `sqrt` function would not be a finite number. If this is indeed the case, we `raise ArithmeticError(f"sqrt({number})is not permitted.")`.

`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 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 either caught (which we discuss later) or the process itself is terminated.

The latter can be observed in 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 stdout using f-string and `print`.[1] The first five numbers are fine and `sqrt` returns the proper

[1] The `flush=True` parameter passed to `print` forces the output to be immediately written and not cached. This is only useful to keep the output in order in this example, because the stdout and stderr are captured and presented in Listing 9.3 and the order of the text could otherwise get mixed up.

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)

```python
"""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 and stderr as well as the exit code of the program `use_sqrt_raise.py` given in Listing 9.2.

```
√0.0≈0.0
√1.0≈1.0
√2.0≈1.4142135623730951
√4.0≈2.0
√10.0≈3.1622776601683795
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 15, 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.
```

results. However, the latter three numbers, `inf`, `nan`, and `-1.0` all would cause an error.

We collect the output of the program in Listing 9.3. 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 an the so-called stack trace is written to the output.

It begins with the line `Traceback (most recent call last):`. In the first line following this text, the source code file 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 as the culprit Listing 9.2. We already took a glimps on how useful the stack trace is back in Section 4.2. Indeed, 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 given, and 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 using an f-string 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 42 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 9.

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 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.

9.3 Handling Exceptions

Of course, we do not want that all possible unexpected error conditions will immediately terminate our process. For example, maybe we programmed a program for painting a picture, the user painted a picture and wants to store it, but enters a wrong destination. It would be annoying if the program would immediately crash. In this section, we discuss how errors can be handled. We use examples that are specifically constructed to cause certain errors to illustrate how we can deal with them.

9.3.1 The `try...except` Block

The `try`-`except` clause exists as primary approach to recover from 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 assume that a block could raise an `ArithmeticError`, we could write `except ArithmeticError as ae:`. The code inside this `except` block would be executed if and only if indeed 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 different types of `Exceptions` may be raised, so we can have multiple `except` blocks. This looks like this:

```
try:  # Begin the try-except block.
    # Code that may raise an exception or that
    # calls a function that may rise one.
except ExceptionType1 as ex1:  # One exception type that can be caught.
    # Code the handles exceptions of type ExceptionType1.
except ExceptionType2 as ex2:  # Another exception type (optional).
    # Code the handles exceptions of type ExceptionType2 that are not
    # instances of ExceptionType1.

next statement  # Executed only if there are no uncaught Exceptions.
```

Let us now try the `try`-`except` block. Back in Section 3.5.1, 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 a very simply operation `r.index(s)`, which works 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.

In Listing 9.4, we explore this behavior. 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.

Listing 9.4: 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 9.5)

```
"""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 9.5: The stdout of the program `try_except_str_index.py` given in Listing 9.4.

```
'Hello' is at index 0.
Error: substring not found
The program is now finished.
```

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 9.5 shows that this is indeed the case: 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.`

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.

While we can catch and handle `Exceptions`s, 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` would occur 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. So 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, to prevent more problems later. A crashed program is the clearest indicator to the user that something is wrong and that actions on their side are required, after all.

Best Practice 43 Only `Exceptions` should be caught by `except` blocks that we can meaningfully handle. 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 9.6, 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`. We first declare the variable as a `float`. Then, in a `try`-`except` block, we perform the actual computation: `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` error. As you can see, we can have two independent `except` clauses.

So, which one will be executed? Certainly, $\frac{1}{0}$ is not finite, so `sqrt` will raise an exception. Then again, $\frac{1}{0}$ cannot be computed at all, so maybe we get a `ZeroDivisionError` error instead? We find that the `except` block for `ZeroDivisionError` is executed. The reason is that in order to invoke

Listing 9.6: 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 9.7)

```python
"""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 as de:  # A division by zero?
    print(f"We got a division-by-zero error: {de}.", 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 9.7: The stdout and stderr as well as the exit code of the program `try_multi_except.py` given in Listing 9.6.

```
We got a division-by-zero error: division by zero.
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.
```

`sqrt(1 / 0)`, the Python interpreter must first compute the result of `1 / 0`. This computation raises `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`? The answer is: Nothing. 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 value of `sqrt_of_1_div_0` is undefined. The variable `sqrt_of_1_div_0` never really received any value at all. Trying to access it will `raise` a `NameError`. While we did declare it, the Python interpreter does not know this variable yet, as it does not have a value. Our program will terminate, because the `NameError` is never caught anywhere. Instead, the stack trace and error information will be printed in Listing 9.7.

Best Practice 44 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.

9.3.2 Exceptions in Exception Handlers

What happens if an exception is raised *inside* an `except` block? In Listing 9.8, we try to compute `sqrt_of_1_div_0 = sqrt(1 / 0)` twice. First in the `try` block, exactly as in Listing 9.6 and then again

Listing 9.8: An example of `Exceptions` being raised inside an `except` block. (stored in file `try_except_nested_1.py`; output in Listing 9.9)

```
"""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 9.9: The stdout and stderr as well as the exit code of the program `try_except_nested_1.py` given in Listing 9.8.

```
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.
```

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 9.9: 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. Therefore, like any other Python code blocks, we can also nest `try`-`except` blocks. Listing 9.10 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` error.

Listing 9.10: An example of nested `try`-`except` blocks, as an improvement over Listing 9.8. (stored in file `try_except_nested_2.py`; output in Listing 9.11)

```
"""Demonstrate nested ‘try...except‘ blocks."""

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)
    try:  # Nesting try-except blocks is totally fine.
        sqrt_of_1_div_0 = sqrt(1 / 0)  # Let's try it again!
    except ZeroDivisionError as de:  # Another ZeroDivisionError?
        print(f"Yet another division-by-zero error: {de}.", flush=True)

print("The program is now finished.")  # This time, we do get here.
```

↓ `python3 try_except_nested_2.py` ↓

Listing 9.11: The stdout of the program `try_except_nested_2.py` given in Listing 9.10.

```
We got a division-by-zero error: division by zero.
Yet another division-by-zero error: division by zero.
The program is now finished.
```

9.3.3 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.

```
try:  # Begin the try-except block.
    # Code that may raise an exception or that
    # calls a function that may rise one.
except ExceptionType1 as ex1:  # One exception type that can be caught.
    # Code the handles exceptions of type ExceptionType1.
# ... maybe more ‘except‘ blocks
else:
    # Code executed if and only if no Exception occurred.

next statement  # Executed only if there are no uncaught Exceptions.
```

In Listing 9.12, we present an example of the `try`-`except`-`else` block. The example is structured a bit similar to Listing 9.6. We again use our own `sqrt` function, this time attempting to compute $\sqrt{\frac{1}{0}}$ and then $\sqrt{3}$, each within its own block. Back in Listing 9.6, we got a `NameError` because we wanted to access the value of a variable that we assigned 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. These 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 9.12 shows that this program can properly finish.

Listing 9.12: An example of the `try`-`except`-`else` block which is structured a bit similar to Listing 9.6 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 9.13)

```
"""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:  # We catch and print the 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 ArithmeticError as ae:  # Or an ArithmeticError?
    print(f"We got an arithmetic error: {ae}.", flush=True)  # Not done.
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 9.13: The stdout of the program `try_except_else.py` given in Listing 9.12.

```
We got a division-by-zero error: division by zero.
√3≈1.7320508075688772
The program is now finished.
```

9.3.4 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 be perform some necessary actions beforehand. 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, and suddenly some unexpected error occurs. If we terminate immediately without closing the file, then the contents of the complete file could be lost. If we close the file before terminating, then at least the data that was successfully written so far will be preserved. By terminating the program, 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 add a `finally` block that contains the code that should always be executed. While we definitely need a `try` block that contains that code that may cause an error, we can optionally add `except` blocks to handle certain `Exceptions` and an `else` block to be executed if no error occurs.

```python
try:  # Begin the try-except block.
    # Code that may raise an exception or that
    # calls a function that may rise one.
except ExceptionType1 as ex1:  # The **optional** except blocks.
    # Code the handles exceptions of type ExceptionType1.
# ... maybe more `except` blocks
else:  # The **optional** else block.
    # Code executed if and only if no Exception occurred.
finally:  # The **optional** finally block.
    # Code will always be executed, even if exceptions are uncaught.

next statement  # Executed only if there are no uncaught Exceptions.
```

Let us explore this in another artificial example in Listing 9.14. 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 `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 for 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 9.15, we show the `stdout` and `stderr` for invoking this function with several different arguments. For `divide_and_print(10, 5)`, both 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 completed 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 strict requirements (as in 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 and so is the `print` at the bottom of our function, since the `TypeError` was properly handled. By the way, had we applied the static type checker `Mypy` to Listing 9.14, it would have informed us that we here try to call `divide_and_print` with an invalid argument, as shown in Listing 9.16.

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, none of our `except` blocks are reached, and 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 9.14: An example for the `try`-`except`-`else`-`finally` block. (stored in file `try_except_else_finally.py`; output in Listing 9.15)

```python
"""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 9.15: The stdout and stderr as well as the exit code of the program `try_except_else_finally.py` given in Listing 9.14.

```
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 9.16: The results of static type checking with Mypy of the program given in Listing 9.14.

```
$ 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 1.15.0 failed with exit code 1.
```

9.3.5 The `with` Block and Context Managers

The `try`-`finally` block allows us to make sure that one action will always be performed, regardless if some other intermediate code fails (and raises exceptions). A use case for this is handling resources that need to be explicitly closed after some time. Typical examples for this are network connections or file input/output. We will discuss both of these topics eventually much later.

Still, let us use file I/O as an example here anyway in Listing 9.17. In this file, we will create and open a text file `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. At the end, we will delete the file to not leave it laying around.

For that purpose, we first import the function `remove` from the module `os`. We also import the type `IO` from the `typing` module. is the basic type for text-based input/output streams and will later use it as type hint.

We open 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." The parameter `encoding` is set to `"UTF-8"`, which defines 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 [138, 330]. This is of no importance here and we will discuss this eventually later in the book.

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. 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 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 stream again, this time for reading. This works exactly as open-

Listing 9.17: Using `try`-`finally` for closing files after writing to and reading from them. (stored in file `file_try_finally.py`; output in Listing 9.18)

```python
"""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 9.18: The stdout of the program `file_try_finally.py` given in Listing 9.17.

```
Hello world!
```

Listing 9.19: The output of Ruff when applied to Listing 9.17: 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
file_try_finally.py:7:18: SIM115 Use a context manager for opening files
  |
6 | # We open the text file "example.txt" for writing.
7 | stream_out: IO = open("example.txt", mode="w", encoding="UTF-8")
  |                  ^^^^ SIM115
8 | try:  # If we succeed opening the file for writing, then...
9 |     stream_out.write("Hello world!")  # ...we write one line of text
  |

file_try_finally.py:14:17: SIM115 Use a context manager for opening files
   |
13 | # We now open the text file "example.txt" for reading.
14 | stream_in: IO = open("example.txt", encoding="UTF-8")
   |                 ^^^^ SIM115
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 0.11.2 failed with exit code 1.
```

Listing 9.20: The output of Pylint when applied to Listing 9.17: 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 3.3.6 failed with exit code 8.
```

ing for writing. 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 via `print`.

At the end of our program, we delete the `example.txt` by calling `remove("example.txt")`. We had imported the `remove` function from the module `os` exactly for this purpose.

In summary, the output of our program, given in Listing 9.17 looks exactly as expected. Since we are diligent programmers, we will, of course, also perform static code analysis by using tools such as

Listing 9.21: Using a `with` block for closing files after writing to and reading from them. (stored in file `file_with.py`; output in Listing 9.22)

```python
"""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 9.22: The stdout of the program `file_with.py` given in Listing 9.21.

```
Hello world!
```

Ruff and Pylint. Their output can be found in Listings 9.19 and 9.20. Oddly enough, they complain: Ruff suggests to use 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.
>
> — *[325], 2001*

The `with` statement has the following syntax, where the `expression` is supposed to return a context manager [65, 285, 300, 325].

```
with expression as variable:
    # block of code that works with variable

# or

with expression:
    # block of code
```

Right now, we are still lacking some background knowledge needed to discuss what a context manager exactly is or how it works. We will therefore do this later in Section 13.5. 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 whether an exception occurred inside the indented block of code directly under `with`.

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 looks more elegant.

Many of Python's resource-related APIs are realized as context managers. This also holds for the file input/output API. We now rewrite Listing 9.17 using a `with` block as Listing 9.21. 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

Listing 9.23: A unit test checking that our new variant of the `sqrt` function given in Listing 9.1 properly raises an `ArithmeticError` if its input is non-finite or negative. (src)

```python
"""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 import sqrt  # Import our new sqrt function.


def test_sqrt() -> None:
    """Test the new `sqrt` function that raises errors."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    # ... We skip the other tests here for space reasons.

    for v in [-1.0, inf, -inf, nan]:
        with raises(ArithmeticError, match="sqrt.* is not permitted."):
            sqrt(v)  # The square root of v is not permitted
```

Listing 9.24: The output of the unit tests in Listing 9.23: The `ArithmeticErrors` are correctly raised.

```
$ pytest --timeout=10 --no-header --tb=short test_sqrt_raise.py
========================== test session starts ==========================
collected 1 item

test_sqrt_raise.py .                                                [100%]

=========================== 1 passed in 0.01s ===========================
# pytest 8.3.5 with pytest-timeout 2.3.1 succeeded with exit code 0.
```

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.

9.4 Testing Exceptions

Back in Section 8.3, we introduced the concept of unit tests and show how the tool pytest can be used to test our functions. We stated in Best Practice 38 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 `Exceptions` and extension handling [177].

If our function is supposed to `raise` a certain exception in some cases, 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.

Let us expand our Listing 8.11 for testing the `sqrt` function to now also test for `ArithmeticErrors`. The `pytest` module offers a so-called context manager [65] named `raises`. If we want to check whether a function indeed raises a certain exception of type `ExpectedExceptionType` for a given input, then we can wrap the corresponding function call into `with raises(ExpectedExceptionType):`. This is block tells the pytest system that, in the following, indented, block, we expect that a certain exception must be raised. If such an exception is not raised, the test fails. If it is raised, the text succeeds.

In particular, we can provide the exception class as well as a regular expression (regex) via the parameter `match` to `raises`. If the block does not raise an exception of the expected type, the test fails. If it does raise a corresponding exception, then it will be converted to a string. The string is then matched against the regex and if it indeed matches, the test succeeds. If it does not match, then the test fails.

But what is a regex? We will learn this later as well, in a part of the book that is not yet written.

Listing 9.25: 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 pytest import raises


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 old_guess != guess:  # Repeat until nothing changes anymore.
        old_guess = guess  # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess


def test_sqrt_1() -> None:
    """Test the ‘sqrt‘ variant that does not itself raise Exceptions."""
    with raises(ArithmeticError):  # We can also test without ‘match‘.
        sqrt(-1.0)  # This is not permitted, but no Exception is raised.


def test_sqrt_2() -> None:
    """Test the ‘sqrt‘ variant that does not itself raise Exceptions."""
    with raises(ArithmeticError, match="sqrt.* is not permitted."):
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors.
```

For now, let's say that a regex is a string pattern that can be compared with another string. For example, the `"sqrt.* is not permitted."` that we use in Listing 9.23 is such a regex. All the "normal" characters in this string must match exactly gainst characters in the text it is compared against. The only thing that makes it different from a normal string is the `.*` that it contains. The dot (`.`) can be replaced with any other character and the star (`*`) says that arbitrarily many such arbitrary characters are allowed. `"sqrt.* is not permitted."` would thus match against `"sqrt is not permitted."`, `"sqrt(1)is not permitted."`, `"sqrt(inf)is not permitted."`, but not `"sqrt is wrong."`. Therefore, our produced error message must follow this pattern for the test to work. And, as you can find in the pytest output in Listing 9.24, for the values `v in [-1.0, inf, -inf, nan]`, no error is reported. For these values, our `sqrt` function does indeed raise the expected `Exceptions`, with the expected error messages. The unit test succeeds.

In Listing 9.25, we write a similar test. This time, we use basically the original version of our `sqrt` function which does not by itself raise any exception. This function is directly implemented in the listing, in the same way as we did in Listing 8.10, and again called `sqrt`. We also write two unit tests into the new file `test_sqrt.py`. Both are intentionally designed to show what happens if either an expected exception is not raised (`test_sqrt_1`) or if its error message does not match the `match` argument (`test_sqrt_2`).

`test_sqrt_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 in Listing 9.26 shows us that this test fails. If the code inside the `with raises(ArithmeticError):` does not actually raise an `ArithmeticError`, the test fails.

The second test case, `test_sqrt_2`, is a bit more interesting. It specifies `with raises(ArithmeticError, match="sqrt.* is not permitted.")`, just like the tests in List-

Listing 9.26: The output of the unit tests in Listing 9.25: No error was raised in it, so the first fails and 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 2 items

test_sqrt.py FF                                                   [100%]

=============================== FAILURES ===============================
______________________________ test_sqrt_1 _____________________________
test_sqrt.py:25: in test_sqrt_1
    with raises(ArithmeticError):  # We can also test without ‘match‘.
E   Failed: DID NOT RAISE <class 'ArithmeticError'>
______________________________ test_sqrt_2 _____________________________
test_sqrt.py:32: in test_sqrt_2
    sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors.
test_sqrt.py:19: in sqrt
    guess = 0.5 * (guess + number / guess)  # The new guess.
E   OverflowError: int too large to convert to float

During handling of the above exception, another exception occurred:
test_sqrt.py:31: in test_sqrt_2
    with raises(ArithmeticError, match="sqrt.* is not permitted."):
E   AssertionError: Regex pattern did not match.
E    Regex: 'sqrt.* is not permitted.'
E    Input: 'int too large to convert to float'
======================= short test summary info ========================
FAILED test_sqrt.py::test_sqrt_1 - Failed: DID NOT RAISE <class '
    ↪ ArithmeticError'>
FAILED test_sqrt.py::test_sqrt_2 - AssertionError: Regex pattern did not
    ↪ match.
 Regex: 'sqrt.* is not permitted.'
 Input: 'int too large to convert to float'
=========================== 2 failed in 0.02s ===========================
# pytest 8.3.5 with pytest-timeout 2.3.1 failed with exit code 1.
```

ing 9.23 for the `sqrt` function that does raise `ArithmeticErrors`. No we do know that our original `sqrt` from Listing 8.10 does not raise such errors … by itself. However, we try to compute `sqrt(10 ** 320)`. This very large integer values will eventually be converted to a `float` inside our function. This will fail, because it is too large, as we just discussed back in Section 9.3.4. This failure results in an `OverflowError`. `OverflowErrors` are a special case of `ArithmeticError` (see later in Figure 9.1). So indeed, this does fulfill our testing requirement: If an `OverflowErrors` is also a `ArithmeticError`, so the right type of exception was raised. However, its error message will not fit to our `match` argument. Therefore, the unit test still fails. The output of Listing 9.26 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 immediately signaled to them by `Exceptions`). Notice that good unit testing goes hand in hand with good documentation in docstrings, because good docstrings reduce the chance of such misunderstandings.

Best Practice 45 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.

9.5 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-existing file. Therefore, different types of exceptions are raised: The former problem causes a `MemoryError` while the later raises an `FileNotFoundError`. The hierarchy of the different problem types is illustrated in Figure 9.1 [40]. There, you can also see why an `except` block catching `ArithmeticErrors` would also catch an `OverflowErrors`, because the latter is a special case of the former.

Depending on the operations that your code tries to perform, you would wrap it into `except` blocks for the errors from this list that you could reasonably expect to be able to recover from. Of course, the documentation of the `Python` functions that you use will tell you which exceptions it could raise. And so should the docstrings of your own code as well as library functions you rely on.

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` 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. For these scenarios, the `except` blocks exist. They allow us to catch errors which we can reasonably expect and that are no 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.

- `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 3.5.1
      - `KeyError` — a dictionary key is not found, see, e.g., Section 5.4
    - `MemoryError` — when we are out of memory
    - `NameError` — e.g., when reading an unassigned variable, see, e.g., Listings 4.5 and 9.6
      - `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 5.2
    - `ValueError` — a parameter has the right type but an inappropriate value, see, e.g., Listing 9.4
      - `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 `Generate` or coroutine terminate; 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 [40].

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 testing capabilities that check whether `Exceptions`s 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 datastructures whose complete content exists in memory at any given time. In Python, we can also iterate over sequences where the items that are constructed at the time when they are actually needed. A good example for this is the `range` datatype. We can iterate over all the 1'000'000'000 `int` elements of `range(1_000_000_000)` in a loop. These many integers do not all exist in memory at the same time. Instead, they are provided one-by-one as needed. From the perspective of a programmer, we can iterate over a `range` and a `list` in exactly the same way. Matter of fact, many objects in Python support iteration.

Vice versa, we can also create container 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 in instance of the corresponding datatype. In Section 5.1.1, we learned that `[1, 2, 2, 3]` creates a list 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 we modify several datastructures in place by combining them with other containers. Invoking `l.extend({1, 2, 3})` will append the elements `1`, `2`, and `3` to a list `l`, for example.

Much later – in a chapter that I have not yet written – we learn that we also can iterate over the contents of a file. 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. In this chapter, we will investigate those that we did not yet already discuss in Section 7.5 and Chapter 5.

10.1 `Iterable`s and `Iterator`s

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 an `typing.Iterator` [328]. 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 Figure 10.2.1. We can use this list `x` in a `for xi in x`-kind of loop arbitrarily often. `x` is an instance of `list` and every list is also an `Iterable`.

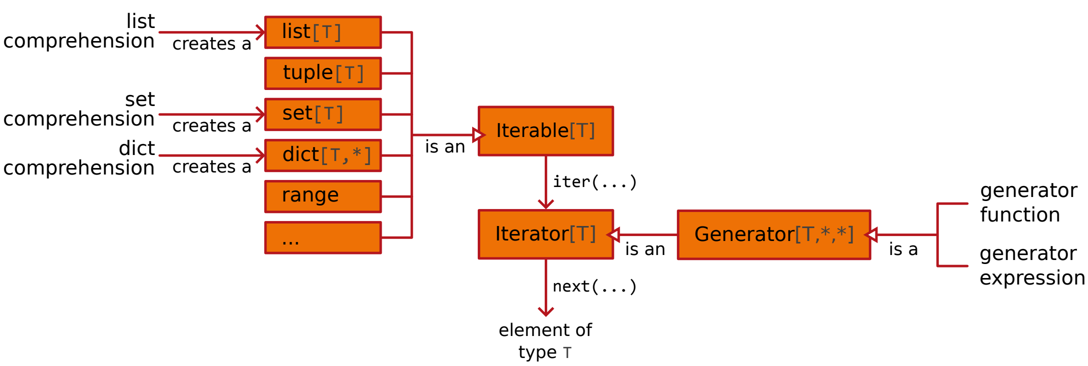
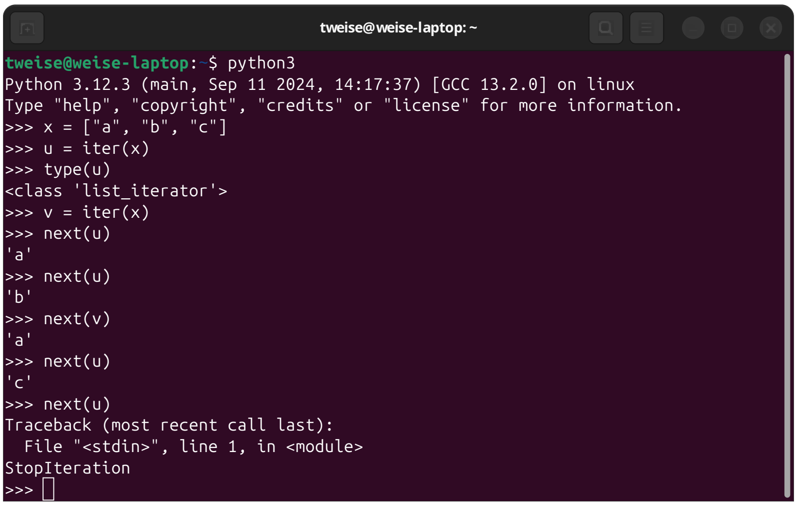
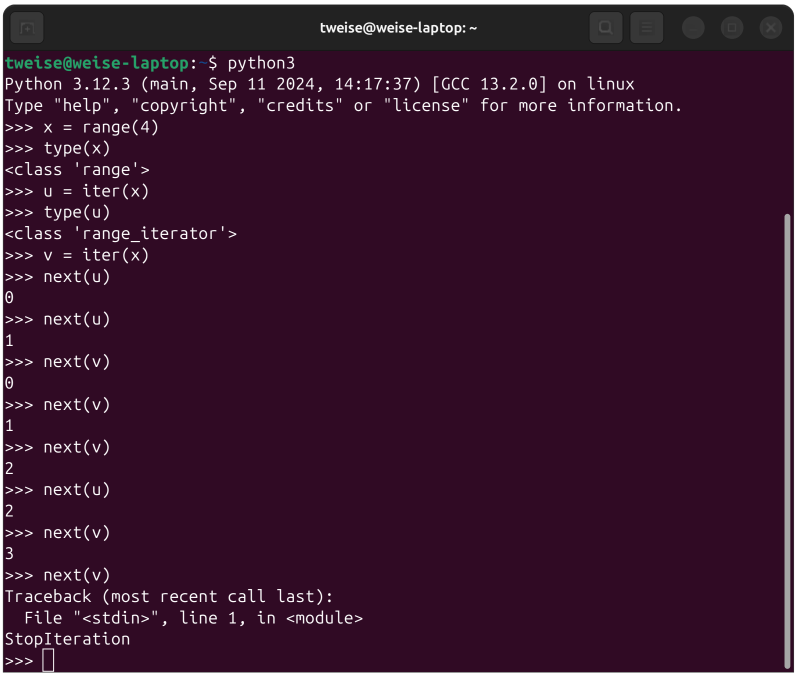


Figure 10.1: The concepts of comprehension, `Iterables`, `Iterators`, and `Generators` in Python.



(10.2.1) Manually iterating over a `list`.



(10.2.2) Manually iterating over a `range`.

Figure 10.2: Manually iterating over a `list` and a `range` in the Python console.

Every time we do loop over `x`, an `Iterator` instance is created internally by (doing something like) invoking `y = iter(x)`. In principle, this `Iterator` object only has to remember its current position in the list, allowing us to query the next item by invoking `next(y)`. The `for`-loop basically does this internally. However, we can also do it "by hand." In Figure 10.2.1, we perform `u = iter(x)` and `v = iter(x)`. This creates two independent `Iterators`, which we can use to step over the list separately. 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 shows us why there is a distinction between `Iterable` and `Iterator`. The former is the object that holds or can generate the data sequence. The latter marks one independent iteration over that 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. This is not an error in the strict sense. This instead is how the end of an iteration sequence is signaled. A `for` loop will, for instance, stop when it encounters this exception.

Listing 10.1: Some simple examples for list comprehension. (stored in file `simple_list_comprehension.py`; output in Listing 10.2)

```python
"""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.

even_numbers: list[int] = [k for k in range(10) if k % 2 == 0]
print(f"even numbers: {even_numbers}")

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 10.2: The stdout of the program `simple_list_comprehension.py` given in Listing 10.1.

```
 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]
letter combinations: ['ax', 'ay', 'bx', 'by', 'cx', 'cy']
letter combinations as tuples: [('a', 'x'), ('a', 'y'), ('b', 'x'), ('b', 'y'), ('c', 'x'), ('c', 'y')]
```

This approach to iterate over collections by first creating an iterator using the `iter` function and then applying `next` to that iterator 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 21 we already clarified that `sets` are unordered data structures. Interestingly, we can also iterate over `dicts` like this. This iteration *only* returns 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.

Figure 10.2.2 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 implementing the `Iterable` functionality. Because of this, the `for y in x`-type of loops can be applied to any `Iterable` or `Iterator` instance `x`.

10.2 List Comprehension

We can create a list by writing it down as a literal, e.g., `[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 `[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 [311]:

```python
# Create a list from the items in a sequence; the condition is optional
[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` from another `sequence`. For example, `[i for i in range(10)]` creates a list with the

Listing 10.3: The output of `Ruff` for the examples for list comprehension in Listing 10.1: 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
simple_list_comprehension.py:5:5: PERF401 Use a list comprehension to
    ↪ create a transformed list
  |
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.
  |     ^^^^^^^^^^^^^^^^^^^^^^^^ PERF401
6 | print(f"result of construction: {squares_1}")  # Print the result.
  |
  = help: Replace for loop with list comprehension

Found 1 error.
# ruff 0.11.2 failed with exit code 1.
```

integer numbers `0` to `9`. The comprehension `[i ** 2 for i in range(10)]` instead creates a list with their squares (where "squaring" is the before-mentioned expression). Additionally, 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`.

In Listing 10.1 we provide some examples for 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 by initially creating an empty list `squares_1`. In a `for` loop letting a variable `i` iterate over the `range(0, 11)`, we would then add `i ** 2` to the list `squares_1` by invoking `squares_1.append(i ** 2)`. This will occupy at least three lines of code. 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 we want to insert into our list by using an `if` statement during the list comprehension. In Listing 10.1, 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. 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. Assume that the first sequence by `"abc"` and the second one be `"xy"`. How can we achieve this? By simply writing two `for` statements! In Listing 10.1, 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 all the characters in the string `"abc"`. 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 extrapolated 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 pro-

Listing 10.4: 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 10.5)

```python
"""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(f"runtime/call: {1000 * time_in_s:.3} ms.")  # Print the result.
```

↓ `python3 list_of_numbers_append.py` ↓

Listing 10.5: The `stdout` of the program `list_of_numbers_append.py` given in Listing 10.4.

```
runtime/call: 57.0 ms.
```

duces `[(’a’, ’x’), (’a’, ’y’), (’b’, ’x’), (’b’, ’y’), (’c’, ’x’), (’c’, ’y’)]`.

In Listing 10.3, we present the output that Ruff, our Useful Tool 3, produces when we apply it to Listing 10.1. Interestingly, Ruff considers the construction of a list via the `append` function as a *performance issue*, signified by the error prefix `PERF`. This is, of course, only the case if the list could as well be created via list comprehension, which is not always possible. But 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, preferable. But why would this also be an issue of performance, i.e., execution speed?

In order to investigate this issue, we could try to simply create two lists with the same contents, once by using the `append` method and once by using list comprehension. 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 interpreter 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. Clearly, it also depends on which version of the Python interpreter we use and our results could be different after each software update. So every runtime measurement is always fuzzy and imprecise. Whatever we would measure would have to take with a grain of salt, but we will try it anyway.

Useful Tool 6 `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 [214, 287].

`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 to 1’000’000.

In Listing 10.4 we therefore first implement a function `create_by_append` which constructs the list using the `append` method. In a loop over the `range(1_000_001)`, it appends all the even numbers to

Listing 10.6: 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 10.7)

```
"""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(f"runtime/call: {1000 * time_in_s:.3} ms.")  # Print the result.
```

↓ `python3 list_of_numbers_comprehension.py` ↓

Listing 10.7: The `stdout` of the program `list_of_numbers_comprehension.py` given in Listing 10.6.

```
runtime/call: 51.3 ms.
```

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 and measure the consumed runtime (which will be in seconds and stored in a `float`). However, due to the above factors which may influence the runtime, a single measurement is not very reliable [214]. We thus instruct `repeat` to take 90 such measurements. All of the measured runtimes are then return as a `list[float]`. The documentation of `timeit` [287] 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.
>
> — *[287], 2001*

Best Practice 46 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 [287]. 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. We print the result of `min` applied to the returned list. 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 10.5.

Now the PDF of this book is built automatically using a GitHub Action [52]. This action executes all the Python example programs and weaves their output into the book. Everytime 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 10.5. 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 10.6. This program is very similar to Listing 10.4. It defines a function `create_by_comprehension` which creates the very same list as `create_by_append` does in Listing 10.4. However, it uses list comprehension. We measure the runtime of this function in exactly the same way as before. In Listing 10.7, 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.

10.3 Interlude: doctests

As last example for list comprehension, let us consider the following scenario: Let's say that you have several different lists. You want to create a new list that contains all the elements of each of these existing lists. In Listing 10.8, 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 `[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. 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

Listing 10.8: A function that flattens `lists` and other `Iterables` using list comprehension. (src)

```python
"""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 10.9: The output of pytest executing the doctests for the examples for list comprehension in Listing 10.8: 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 8.3.5 with pytest-timeout 2.3.1 succeeded with exit code 0.
```

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 present its output. This time, we do something else: We present doctests for `flatten`.

Useful Tool 7 A doctest is a unit test written directly into the docstring of a function or module. We therefore insert small snippets of Python code and their 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 by modules like `doctest` [80] or tools such as pytest [159] (Useful Tool 5). 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 a 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 10.8.

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. 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. Passing a list-of-lists-of-lists to `flatten` as in the third test (`flatten([[[1], [2], [3]], [], [[4], [5], [6]]])`) 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.

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 output in Listing 10.9 shows that our function indeed fulfills the requirements imposed by its doctests.

Best Practice 47 Where ever possible, the docstrings of functions 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 36).

While we here 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

Set comprehension works very much the same as list comprehension. The corresponding syntax is as follows:

Listing 10.10: Some simple examples for set comprehension. (stored in file `simple_set_comprehension.py`; output in Listing 10.11)

```python
"""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, k) 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: {primes}")
```

↓ `python3 simple_set_comprehension.py` ↓

Listing 10.11: The stdout of the program `simple_set_comprehension.py` given in Listing 10.10.

```
 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: {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}
```

```
# Create a set from the items in a sequence; the condition is optional
{expression for item in sequence if condition}
```

In Listing 10.10, we provide some simple examples for set comprehension and in Listing 10.11 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 first do this by starting with an empty set `roots_1` and then iteratively append the 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 `{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 almost the same.

Let us now try do something more interesting: We want to create the set `primes` of the prime numbers [67, 238, 317] 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 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 `k - 1`. 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. However, we are doing set comprehension, so each value can occur at most once.

This extremely computationally inefficient 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 it 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).

10.5 Dictionary Comprehension

Dictionary comprehension works almost the same as set and list comprehension [310]. 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:

```
#␣Create␣a␣dict␣from␣the␣items␣in␣a␣sequence;␣the␣condition␣is␣optional
{expression1:␣expression2␣for␣item␣in␣sequence␣if␣condition}
```

In Listing 10.12 we provide some examples for dictionary comprehension. We again start by "manually" creating a dictionary and then show how dictionary comprehension is much more concise. Similar to Listing 10.1, where we discussed list comprehension, we want to construct a datastructure 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 10.13 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.

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 10.10 when constructing the set of none-prime numbers.

Listing 10.12: Some simple examples for set comprehension. (stored in file `simple_dict_comprehension.py`; output in Listing 10.13)

```python
"""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 numbers which are not prime.
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 of non-primes: {maxdiv}")
```

↓ `python3 simple_dict_comprehension.py` ↓

Listing 10.13: The stdout of the program `simple_dict_comprehension.py` given in Listing 10.12.

```
 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 of non-primes: {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}
```

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 be several `m`. However, a dictionary allows each key to appear at most once. In the dictionary comprehension, the last `m` value that meets the condition prevails. 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 in the dictionary `maxdiv`. Finally, `maxdiv` is printed and indeed contains the largest divisors for the numbers $k \leq 20$.

10.6 Generator Expressions

List comprehension gives us the ability to elegantly solidify a sequence of data into an instance of `list`. A list datastructure 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 such 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 datastructure that can store the elements and that lets us access them in an efficient way.

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. It accepts an `Iterator` as input. 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 that the elements be provided via the `next` function of the `Iterator` it receives as input.

To allow us to create "lazy" sequences that return their elements as needed (instead of keeping them all in memory all the time), generator expressions exist [124]. The syntax for generator expressions is basically the same as for list expressions, except that we use parenthesis instead of square brackets:

```
# A generator with the items in a sequence; the condition is optional
(expression for item in sequence if condition)

# If generator expressions are single function parameters, then the
# parentheses are unnecessary.
function(expression for item in sequence if condition)
```

First, let us investigate how generator expressions work internally. In Listing 10.14, 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 will create the next value `i` and compute `i ** 2` only when they are actually needed. The proper type hint for generator expressions is `Generator[type, None, None]`, where `type` is the type of elements (here `int`) and `Generator` is imported from the `typing` module. Anyway, we store the generator expression into a variable `gen`. As the output shows, the `type` of `gen` is `<class 'generator'>`, not `list` and neither `tuple`. We can obtain the first element of the sequence, `0`, by invoking `next(gen)`. The second invocation of `next(gen)` yields `1` and the third call to `next(gen)` gives us `4`. Finally, as expected, `next(gen)` again returns `9`. In our example program, we stop here. This means that the 999'999'996 elements are never created.

To better understand this, let us try to construct a generator expression which explicitly tells us when an element is created. For this, we first create the function `as_str` in Listing 10.16,. `as_str`

Listing 10.14: An investigation of how generator expressions work by using the `next` function. (stored in file `generator_expressions_next_1.py`; output in Listing 10.15)

```
"""The iteration behavior of generator expressions and `next`."""

from typing import Generator  # Import the generator type hint.

gen: Generator[int, None, None] = (i ** 2 for i in range(1_000_000_000))
print(f"{type(gen) = }")  # Type is `generator`, not `tuple`
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
# We stop using the generator here, so the remaining 999999996 elements
# are never generated.
```

↓ `python3 generator_expressions_next_1.py` ↓

Listing 10.15: The stdout of the program `generator_expressions_next_1.py` given in Listing 10.14.

```
type(gen) = <class 'generator'>
next(gen) = 0
next(gen) = 1
next(gen) = 4
next(gen) = 9
```

Listing 10.16: 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 10.17)

```python
"""The iteration behavior of generator expressions and `next`."""

from typing import Generator  # Import the generator type hint.


def as_str(a: int) -> str:
    """
    A simple function printing `a` and returning it as `str`.

    :param a: the input number
    :return: the output

    >>> as_str(5)
    input is 5
    '5'
    """
    print(f"input is {a}")  # Will show that generator is lazy-evaluated
    return str(a)


lst: list[str] = [as_str(j) for j in range(3)]
print("list created")
print(f"{next(iter(lst)) = }\n")

gen: Generator[str, None, None] = (as_str(j) for j in range(3))
print("generator created")
print(f"{next(gen) = }")  # Prints input, then prints `"0"`
print(f"{next(gen) = }")  # Prints input, then prints `"1"`
print(f"{next(gen) = }", flush=True)  # Last print, `"2"`
print(f"{next(gen) = }", flush=True)  # Raises StopIteration
```

↓ `python3 generator_expressions_next_2.py` ↓

Listing 10.17: The stdout and stderr as well as the exit code of the program `generator_expressions_next_2.py` given in Listing 10.16.

```
input is 0
input is 1
input is 2
list created
next(iter(lst)) = '0'

generator created
input is 0
next(gen) = '0'
input is 1
next(gen) = '1'
input is 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.
```

accepts an integer parameter `a` as input. It immediately writes `f"input is {a}"` to the `stdout`. This means that whenever `as_str` is called, we will immediately see some output. Then, it returns simply 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. 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")` completes. In the next line, we print the value of `next(iter(lst))`. This creates an `Iterator` over `lst` (via `iter`) and returns its first element (via `next`). The corresponding output appears after the output written by our `as_str` function. This shows that the `list` has indeed be created in its entirety before we began processing it in the next line.

What happens if the use `as_str` in a generator expression instead? We set `gen = (as_str(j)for j in range(3))` – and nothing happens. The next line, `print("generator created")` prints its output. Obviously, `as_str` has not yet been invoked.

We now iteratively query `gen` with `next`. Indeed, the first time we do this, `input is 0` appears in `stdout` before the result of the interpolated f-string `f"next(gen): {next(gen)}"` is printed. When we query `next(gen)` again in the next line, `input is 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, because this `StopIteration` getting raised.

> **Best Practice 48** Generator expressions shall be preferred over list comprehension if the sequence of items only needs to be processed once. Generator expression 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 10.18 shows several examples for 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 write the generator expression `(sin(k)for k in range(-100, 100))` and pass it to `max` to get the maximum value as well as to `min` to get the minimum value of the sequence. Again it is not necessary to write double-parentheses.

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`. We pass it to the function `all`, which returns `True` if and only if all the elements in the sequence that it receives as argument are `True`. Otherwise it returns `False`. 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 will never evaluate `"e" in "are"`, because `"e" in "how"` is already `False` and it is clear, at this point, that `all` must return `False` regardless of the rest of sequence. 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 a part of the data (and stored it nowhere).

A complement to the `all` function is the `any` function. 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`. `any` returns `True` if at least one of the elements of

Listing 10.18: 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 10.19)

```python
"""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 10.19: The stdout of the program `generator_expressions_in_reduction.py` given in Listing 10.18.

```
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
```

its argument sequence also is `True`. It returns `False` if *all* of the elements in the argument sequence are also `False`. We immediately see that `"w" in "how"` is `True`, so 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.

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 Listing 10.20.

We create a generator `gen` which creates the tuples $(n, \sin n)$ for all $n \in 0..1'000'000'000$ by writing `((n, sin(n))for n in range(1_000_000_000))`. For some reason, we want to find the first n for which $\sin n < -0.99999$. We therefore iterate over this generator `gen` and directly unpack the tuples, 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 $\sin$ `n`. 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`. The output `sin(11)=-0.9999902065507035` tells us that only 12 loop steps were done (as n starts with 0). Using a generator 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 datastructure 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.

Finally, generator expressions can also be passed to the constructors of collection datastructures or other functions that create such datastructures. 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. In Listing 10.22, we first define a string `csv_text` with the value `"22,56,33,67,43,33,12"`.

Listing 10.20: The behavior of generator expressions in `for` loops. (stored in file `generator_expressions_loops.py`; output in Listing 10.21)

```
"""Generator expressions in `for` loops."""

from math import sin
from typing import Generator

# The generator expression produces tuples (n, sin(n)).
gen: Generator[tuple[int, float], None, None] = (
    (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:
    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.
```

↓ `python3 generator_expressions_loops.py` ↓

Listing 10.21: The stdout of the program `generator_expressions_loops.py` given in Listing 10.20.

```
sin(11)=-0.9999902065507035
```

Invoking `csv_text.split(",")` will split the string into a list of single strings based on the delimiter `","`. This will thus yield `["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(","))` is a generator that converts the strings in the list into integers. Passing this generator to the `list` constructor, i.e., calling `list(int(i)for i in csv_text.split(","))` will create a list with the contents of the generator. 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 to the `sorted` function mentioned back in Section 5.3. 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 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 and the values are their integer representation. Notice that this dictionary also does not contain duplicates.

If we apply the trusty tool Ruff to Listing 10.22, it tells us that using generators to create lists, sets, and dictionaries is unnecessary, since we could use the corresponding comprehension instead. There is no tuple comprehension, though, and also passing the generator expression to the `sorted` function is without better alternative.

Listing 10.22: Using generator expressions when creating collection datastructures. (stored in file `generator_expressions_to_collection.py`; output in Listing 10.23)

```
"""Using generator expressions to create collections."""

csv_text: str = "22,56,33,67,43,33,12"

as_list: list[int] = list(int(i) for i in csv_text.split(","))
print(f"a generator converted to a list: {as_list}.")

as_tuple: tuple[int, ...] = tuple(int(i) for i in csv_text.split(","))
print(f"a generator converted to a tuple: {as_tuple}.")

as_set: set[int] = set(int(i) for i in csv_text.split(","))
print(f"a generator converted to a set: {as_set}.")

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}.")

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 10.23: The stdout of the program `generator_expressions_to_collection.py` given in Listing 10.22.

```
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}.
```

Listing 10.24: The output of Ruff for collection creation-expressions in Listing 10.22: 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
generator_expressions_to_collection.py:5:22: C400 Unnecessary generator (rewrite as a list comprehension)
  |
3 | csv_text: str = "22,56,33,67,43,33,12"
4 |
5 | as_list: list[int] = list(int(i) for i in csv_text.split(","))
  |                      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ C400
6 | print(f"a generator converted to a list: {as_list}.")
  |
  = help: Rewrite as a list comprehension

generator_expressions_to_collection.py:11:20: C401 Unnecessary generator (rewrite as a set comprehension)
   |
 9 | print(f"a generator converted to a tuple: {as_tuple}.")
10 |
11 | as_set: set[int] = set(int(i) for i in csv_text.split(","))
   |                    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ C401
12 | print(f"a generator converted to a set: {as_set}.")
   |
   = help: Rewrite as a set comprehension

generator_expressions_to_collection.py:17:27: C402 Unnecessary generator (rewrite as a dict comprehension)
   |
15 | print(f"a generator converted to a sorted list: {as_sorted_list}.")
16 |
17 | as_dict: dict[str, int] = dict((i, int(i)) for i in csv_text.split(","))
   |                           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ C402
18 | print(f"a generator of tuples converted to a dict: {as_dict}.")
   |
   = help: Rewrite as a dict comprehension

Found 3 errors.
No fixes available (3 hidden fixes can be enabled with the `--unsafe-fixes` option).
# ruff 0.11.2 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* [245]. From the perspective of the user of a generator function, it is a function that returns 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 a comprehension or pass it to the constructor of a collection, if we want to.

From the perspective of the implementor, however, it 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 has the feel of 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 10.25, 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 as `Generator[int, None, None]`, 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 iterate over the `Generator` like over any nor-

Listing 10.25: A very simple generator function yielding the numbers 1, 2, and 3. (stored in file `simple_generator_function_1.py`; output in Listing 10.26)

```
"""A simple example for generator functions."""

from typing import Generator  # The type hint for generators.


def generator_123() -> Generator[int, None, None]:
    """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.

as_list: list[int] = list(generator_123())  # Create a list.
print(f"{as_list = }")  # The list is [1, 2, 3].

gen: Generator[int, None, None] = generator_123()  # Use directly.
print(f"{next(gen) = }")  # First time next: get 1.
print(f"{next(gen) = }")  # Second time next: get 2.
print(f"{next(gen) = }", flush=True)  # Third time next: get 3.
print(f"{next(gen) = }", flush=True)  # raises StopIteration
```

↓ `python3 simple_generator_function_1.py` ↓

Listing 10.26: The stdout and stderr as well as the exit code of the program `simple_generator_function_1.py` given in Listing 10.25.

```
as_list = [1, 2, 3]
next(gen) = 1
next(gen) = 2
next(gen) = 3
Traceback (most recent call last):
  File "{...}/iteration/simple_generator_function_1.py", line 19, in <module>
    print(f"{next(gen) = }", flush=True)  # raises StopIteration
             ^^^^^^^^^
StopIteration
# 'python3 simple_generator_function_1.py' failed with exit code 1.
```

Listing 10.27: A generator function yielding the infinite sequence of Fibonacci numbers [256, 315]. (stored in file `simple_generator_function_2.py`; output in Listing 10.28)

```
"""A simple example for generator functions."""

from typing import Generator  # The type hint for generators.


def fibonacci() -> Generator[int, None, None]:
    """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 simple_generator_function_2.py` ↓

Listing 10.28: The stdout of the program `simple_generator_function_2.py` given in Listing 10.27.

```
0
1
1
2
3
5
8
13
21
34
```

mal `Iterator`. We first set `gen = generator_123()`. The 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` and resumed when `next` is called (unless the sequence was finished, that is). This becomes visible when we create a generator function that returns an infinite sequence.

In Listing 10.27, we implement a generator function producing the Fibonacci numbers [256, 315]. These numbers follow the sequence $F_n = F_{n-1} + F_{n-2}$ where $F_0 = 0$ and $F_1 = 1$. We therefore define the function `fibonacci`, which is annotated to return a `Generator`. It begins by setting `i` $= F_0 = 0$ and `j` $= F_1 = 1$. In a `while` loop which will never stop (as the loop condition is imply set to `True`), it always yields `i`. Then, assign `i` and `j` to `j` and `i + j`, respectively. This stores the old value of `j` in `i`. It also stores the sum of the old `i` and `j` values in `j`. In the next loop iteration, `yield` will then produce the next Fibonacci number.

We can now loop over the `Generator` returned by `fibonacci()` in a normal `for` loop. This would result in an endless loop, unless we insert some termination condition. In our loop, we print the Fibonacci numbers `a` we get, but stop the iteration via `break` once `a > 300`.

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 final example for generator functions, let us wrap our code for enumerating prime numbers [67, Listing 10.29: A generator function yielding the infinite sequence of prime numbers [67, 238, 317]. (src)

```python
"""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, None, None]:
    """
    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.
```

238, 317] from back in Section 7.4 into a generator function in Listing 10.29. We used nested `for` loop to produce the prime numbers in Listing 7.11, where the outer loop would iterate at most to 199. We do not need such limit anymore, as we can 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 difference between the old and new code is that, once we confirm a number to be prime, we do not just add it to the list `found` of odd primes, we also need to `yield` it.

We demonstrate how our generator 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 in the doctest should return `11`. You can use pytest by yourself to check whether the code works as expected. . .

Listing 10.30: An example for `takewhile` and `filter`. (stored in file `filter_takewhile.py`; output in Listing 10.31)

```python
"""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

less_than_50: list[int] = list(takewhile(lambda z: z < 50, primes()))
print(f"primes less than 50: {less_than_50}")


def is_sqr_plus_1(x: int) -> bool:
    """
    Check whether `x` is of the form `y^2+1`.

    :param x: the number to check
    :return: `True` if there is an integer `y` such that `x=y^2+1`.
    """
    y: int = isqrt(x)
    return x == (y ** 2) + 1


sqrs_plus_1: list[int] = list(filter(is_sqr_plus_1, takewhile(
    lambda z: z < 1000, primes())))
print(f"primes less than 1000 and of the form x^2+1: {sqrs_plus_1}")
```

↓ `python3 filter_takewhile.py` ↓

Listing 10.31: The stdout of the program `filter_takewhile.py` given in Listing 10.30.

```
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 x^2+1: [2, 5, 17, 37, 101, 197, 257,
    ↪ 401, 577, 677]
```

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 [245]. 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` [140]. 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 Listing 7.11. 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 Listing 10.30.

`takewhile` is a function with two parameters: The second parameter is an `Iterator`. Let's say that it 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. Back in Section 8.5, we learned that we can also pass functions or `lambdas` as arguments to other functions. This is a practical example of that. Basically, `takewhile` constructs a new `Iterator` which returns the elements from the original `Iterator` as long as the predicate function returns `True` for them. As soon as the predicate returns `False`, it will stop the iteration. Therefore, the answer to "How can I extract all the numbers less than 50 from the prime sequence?" is simply to call `akewhile(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. It, too, accepts a predicate and an `Iterator` as

Listing 10.32: An example for the function `map`. (stored in file `map.py`; output in Listing 10.33)

```python
"""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 10.33: The stdout of the program `map.py` given in Listing 10.32.

```
found value 23.
found value 445.
csv_text_sqrs: {36, 198025, 9, 144, 529, 25}
longest word length: 5
```

input. Different from `takewhile`, the new `Iterator` created by `filter` does not stop if the predicate returns `False`. However, it only returns only those elements for which the predicate returned `True`. In Listing 10.30, we use this to select prime numbers x for which an integer y exists such that $x = y^2 + 1$. This time, we implement the predicate as function `is_sqr_plus_1` and pass this function to `filter`. Since there again probably infinitely many such prime numbers, we return only those that are less than 1000, for which we use `takewhile`. The results can be seen in Listing 10.31: 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 function `map`. We explore its use in Listing 10.32. Back in Listing 10.22, we used a generator expression to process data that we exracted 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 sequence passed in as its second argument. The result of `map` is a new sequence with the return values of this function. In Listing 10.32, we map the string `csv_text` split at all `","` to `ints` and then `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 unique squares of the values in the CSV data, i.e., we discard all duplicate squares. 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 provide a `lambda` that does the transformation. Now we want to retain only the unique values. 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 10.33, 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 for `map`, we have a list of words `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`.

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.

Listing 10.34: An example for 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 10.35: The output of pytest executing the doctests for the `zip` example from Listing 10.34.

```
$ 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 8.3.5 with pytest-timeout 2.3.1 succeeded with exit code 0.
```

As last example for sequence processing we play a bit with the `zip` function. This function accepts several `Iterabless` as argument and returns a new `Iterator` which steps through all of input iterables in synch, returning tuples of with one value of each of them. For example, `zip([1, 2, 3], ["a", "b", "c"])` returns an `Iterator` that produces the 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` [38].

In Listing 10.34, 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

$$\texttt{distance(p1, p2)} = \sqrt{\sum_{i=1}^{n} (\texttt{p1}_i - \texttt{p2}_i)^2} \qquad (10.1)$$

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 expansion 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 10.35 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` [140] 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 dict 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, allow other code outside to process the result, and then resume with the generation of the next element. Finally, sequences of data can be processed by aggregating and transforming functions. These functions can process containers, comprehensions, generator expressions, and generators alike.

Part III

Classes

Chapter 11

Basics of Classes

We learned about simple datatypes in Chapter 3 and about collections in Chapter 5. However, in many situations, we deal with data that cannot satisfyingly represented by either of them alone. Often, datatypes and the operations on them 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]`, but this comes with several drawbacks. On 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"`.

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. This allows us to define a `class` for complex numbers which has attributes with the name `real_part` and `imaginary_part`. We can define operators 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. 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 [104, 176], Microsoft Word [81, 188], and Adobe PDF [211, 318]. On the input side, you would like to provide the user with a uniform way to create documents, to add text paragraphs, to insert graphics, to set the font of texts, and so on. 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 ability to divide a text into chapters which can be divided into paragraphs which can be divided into runs of text with different fonts. Obviously, these operations will be implemented differently for the different output formats. We could try to solve this by making different modules with functions of the same signatures for the different output formats. But this would be a huge hassle, in particular since 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 tool like `Ruff` could tell is if some module is no longer synchronized due 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 implementation-specific stuff would all be invisible to the user, exactly as it should be. So the second important use case for classes are that they offer us a very

[1] `Python` already offers the datatype `complex`, but for the sake of the example, assume it does not.

nice abstraction for defining and implementing APIs.

Classes therefore can solve two important problems where the basic datatypes 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 operators under 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:

```
class MyClass:   # or `class MyClass(MyBaseClass)`
    """Docstring of the class."""

    def __init__(self) -> None:
        """Docstring of the initializer __init__."""
        # initialize all attributes
        #: Meaning of attribute 1
        self.attribute_1: type hint = initial value
        # ...

    def my_method(self, param1, param2) -> result:
        """Docstring of my_method."""
        # compute something using the attributes
```

Best Practice 49 Class names should follow the CapWords convention (often also called camel case), i.e., look like `MyClass` or `UniversityDepartment` (not `my_class` or `university_department`) [302].

11.1 A Simple Immutable Class for Points in the Two-Dimensional Euclidean Plane

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. We create the `class Point` inside a Python file `point.py`. This is done relatively easily: All we have to do is to begin a block with, well, `class Point:`. Inside this block, we will place all the *methods* of the `class`. Methods are like functions, 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. Of course, first, we always place a docstring.

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)
```

Best Practice 50 At the beginning of a `class`, a docstring is placed which describes what the class is good for. This docstring and include doctests to demonstrate the class usage, or such doctests can be placed in the docstring of the module.

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 at-

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)

```python
"""Examples for using our class :class:`Point`."""

from point import Point

p1: Point = Point(3, 5)  # Create a first point.
print(f"{p1.x = }, {p1.y = }")  # p1.x = 3, p1.y = 5
print(f"{isinstance(p1, Point) = }")  # True

p2: Point = Point(7, 8)  # Create a second point.
print(f"{p2.x = }, {p2.y = }")  # p2.x = 7, p2.y = 8
print(f"{isinstance(p2, Point) = }")  # True

print(f"{isinstance(5, Point) = }")  # False

print(f"{p1 is p2 = }")  # False

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(4)]
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
isinstance(p1, Point) = True
p2.x = 7, p2.y = 8
isinstance(p2, Point) = True
isinstance(5, Point) = False
p1 is p2 = False
p1.distance(p2) = 5.0
p2.distance(p1) = 5.0
(0, 0), (0, 1), (0, 2), (0, 3), (1, 0), (1, 1), (1, 2), (1, 3), (2, 0), (2, 1), (2, 2), (2, 3)
```

tributes. 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 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, even after the execution of the method is completed.

> **Best Practice 51** Object attributes must only be created inside the initializer `__init__`. 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`. Otherwise, we set `self.x: Final[int |float] = x` and `self.y: Final[int |float] = y`.[2] These lines create the attributes `self.x` and `self.y` of the object which was passed in via the parameter `self`. Additionally, the type hint `Final` from the `typing` module annotates a variable as immutable [274]. In other words, we do not allow the coordinates of our `Points` to change after object creation.

Best Practice 52 Every attribute of an object must be annotated with a type hint and a documentation comment when created in the initializer `__init__` [173]. type hints work as with normal variables.

Best Practice 53 The type hint `Final` marks an attribute as immutable. All attributes that you do not intend to change should be annotated with `Final`.

Best Practice 54 An attribute is documented in the line *above* the attribute initialization by writing a *comment* starting with `#:`, which explains the meaning of the attribute [266]. (Sometimes, the documentation is given as string directly below the attribute definition [109], 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 is allocated. Then, the initializer is invoked as `__init__(p, 1, 2)`. As a result, `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 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 `class`es also allow us to define operations on such points in form of methods. As an example, we implement a method `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` and, therefore, not yet available as type *inside* its definition. Using the string here is therefore just a crutch with no real other effect.

Best Practice 55 All methods of `class`es must be annotated with docstrings and type hints.

[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.

Best Practice 56 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.

In Listing 11.2, we begin by creating 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 resulting new instance of `Point` 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 can check if an object `o` is an instance of our class `Point` by writing `isinstance(o, Point)`. For `p1`, this returns `True`.

We now create a second instance, `p2`, of the class `Point`. We assign `7` to `p2.x` and `8` to `p2.y` via the `__init__` initializer, which is automatically invoked when we write `Point(7, 8)`. We can again print the values of these attributes using an f-string. While `isinstance(p2, Point)` is again `True`, `isinstance(5, Point)` returns `False`.

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 `Point` (and the appropriate type hint would then by `list[Point]`). We can even create such a list with list comprehension, exactly as we learned back in Section 10.2. And 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 generates a sequence of strings of the form `"(x, y)"`, which 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.

The objects of our `Point` class are *immutable*, i.e., once created, the attributes cannot be changed anymore.[3] The x- and y-attributes are initialized by the `__init__` initializer. They are marked with the type hint `Final` and thus changing them is an error. 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 [27], 2008*

Definition 8 (Immutable) After their creation, the attributes of *immutable* objects cannot be modified anymore.

Creating classes whose instances are immutable has several benefits: 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 they 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. This is particularly 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.

[3] Well, in `Python`, type hints are just hints and not enforced by the interpreter [274]. So the x- and y-attributes of instances of `Point` can actually be changed. However, tools like `Mypy` will detect such mistakes and report them [274].

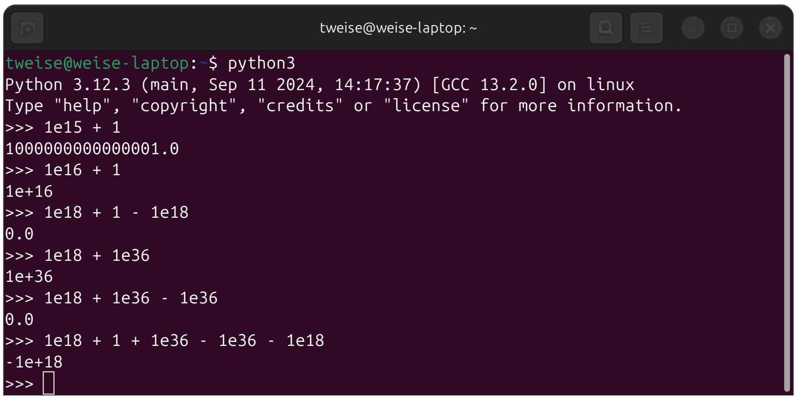


Figure 11.1: Errors occurring when adding floating point numbers due to the limited precision of the datatype `float`.

11.2 Encapsulation and Accurately Adding Floating Point Numbers

We now want to implement our first maybe actually useful piece of code, something that can be used in a real productive system.[4] To do such a cool thing, let us first revisit the limitations of the datatype `float` from way back in Section 3.3. We learned that the datatype `float` can represent numbers accurately to about fifteen to sixteen digits. This is illustrated in Figure 11.1: If we add `1` to 10^{15} = `1e15`, the correct result is `1000000000000001.0`. However, if we add `1` to 10^{16} = `1e16`, the result is still `1e16`. It is not possible to represent the number $1+10^{16}$ correctly with the 64 bit double precision floating point numbers that Python offers [98]. Usually, that itself is totally fine: There are few application in "everyday programming" where we really need more than fifteen digits of precision.

However, what happens if we try to compute $10^{18}+1$ and then subtract 10^{18} from this sum? Obviously, the result should be 1, a number that we can correctly and accurately represent as a `float`. The actual result of this computation in Python is 0. The reason is that `1e18+1 == 1e18` and `1e18 - 1e18 == 0`.

Similarly, computing `1e18 + 1 + 1e36 - 1e36 - 1e18` yields `-1e18`, while the correct result would again be `1`. The reason is that first, `1e18 + 1 == 1e18` is computed. Then `1e18 + 1e36` yields `1e36`, from which we subtract `1e36` and get `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`. It is easy to see why this does not work out, because the designers of the floating point number arithmetic unit in CPUs had to choose some limited number of bits per `float` and concluded that fifteen digits are reasonable whereas 36 are probably never actually needed. Well, never, unless you try to add up large and small numbers.

Luckily, in the 1960s, Kahan [149] and Babuška [12] had an idea how to fix that: 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?

We start with `sum = 0` and `cs = 0`. For every number `value` that we want to add, we first compute `t = sum + value`. `t` thus is the sum as precisely as a `float` can represent it. We then calculate `error = (sum - t)+ value`. Since `t = sum + value`, we would assume that `error` should be 0, because effectively, it is `(sum - (sum + value))+ value`. However, some digits could have been "lost" when computing `sum + value`. We then set `sum = t` and accumulate the errors in `cs`, i.e., set `cs += error`.

Back to the example `1e18 + 1 - 1e18`. Obviously, in the first step, we get `t = sum + value`, i.e., `0 + 1e18`, meaning that `t = 1e18`. `error = (sum - t)+ value` becomes `error = (0 - 1e18)+ 1e18` which gives us `0.0`. We thus then set `sum = 1e18` and `cs`, which was `0`, becomes `0 + 0`, which still is `0`.

The second step is more interesting: We compute `t = sum + value`, but now this

[4]Yes, we did implement the method of LIU Hui (刘徽) for approximating π and Heron's Method for approximating the square root. These implementations were not more accurate than the constant and function Python already has built in. But they are nice, though, I did enjoy doing them.

Algorithm 1: The second-order Kahan-Babuška-Neumaier summation algorithm [12, 149, 152, 191] over an array x as defined by Klein in [152].

```
sum ← 0;   cs ← 0;   ccs ← 0;
for i ∈ 0..n − 1 do                                          ▷ For each number to add.
    t ← sum + x[i];  ▷ Compute the sum and (below) the first-order error term [12, 149].
    if |sum| ≥ |x[i]| then c ← (sum − t) + x[i];          ▷ the tweak by Neumaier [191]
    else c ← (x[i] − t) + sum;
    sum ← t;
    t ← cs + c;                         ▷ The second-order error summation by Klein [152] begins.
    if |cs| ≥ |c| then cc ← (cs − t) + c;                ▷ the tweak by Neumaier [191] again
    else c ← (c − t) + cs;
    cs ← t;   ccs ← ccs + cc;
    return sum + cs + ccs
```

mean `t = 1e18 + 1` and this results in `t = 1e18`. The `1` is lost. Well, almost: `error = (sum - t)+ value` becomes `error = (1e18 - 1e18)+ 1`, i.e., `error = 1`. So after this step, `sum = 1e18` and `cs = 0 + 1` becomes `1`.

We now subtract `1e18`, which is equivalent to adding `-1e18`. This means that we first compute `t = sum + value`, i.e., `t = 1e18 + -1e18`, i.e., `t = 0`. For the error term, we get `error = (1e18 - 0)+ -1e18`, which is `0`. We now got `sum = 0` and `cs = 1 + 0`, which still means that `cs = 1`.

The final result will be `sum + cs` and this indeed gives us `0 + 1`, i.e., `1`! Although we could not represent the intermediate results of the addition exactly, we still got the right result. The trick was to keep track of the total error in the variable `cs`.

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 [191]. Klein [152] 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 most advanced and accurate algorithm is given in Algorithm 1. And we are now going to implement this in Listing 11.4.

What we have learned so far allows us to already implement proper and useful algorithms. This no longer is child's play, this is the real deal.

We now create a class whose instances each maintain one such running sum. The running sum will initially be 0. With a method `add` we can add one number to it. The method `result` should return the current result of the sum. We prefer using such a class over making a function that simply sums up a sequence because it allows for a much greater flexibility. We can use instances of our class to sum up sequences, but we can also use them to sum up numbers in any other way that pleases us.

The attributes of our class should be *hidden*. As we have discussed before, we will have a current running sum value `sum` and the error sum value `cs`. Since we implement Algorithm 1, we also need the second-order error sum term `ccs`. Now these values only make sense "inside" our object's logic. 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 object only via the methods `add` and `result`.

Definition 9 (Encapsulation) *Encapsulation* means that all the attributes of an object can only be accessed by methods. It is impossible to access them directly or to modify them in any other way. Encapsulation therefore allows the programmer to ensure that the state of an object can only be modified in a consistent and correct way.

This is going to be the interface for our new class `KahanSum`, which, in turn, is based on [152].

In the initializer `__init__`, we create the three attributes `__sum`, `__cs`, and `__ccs` and initialize them to `0`. These names are directly taken from Algorithm 1. Notice the double leading underscores in front of the names.

Best Practice 57 Names of attributes and attributes that start with a double leading underscore (`__`) are to be treated as *private* methods and attributes. 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).

With that out of the way, a new instance of `KahanSum` will start with all of its three summation variables initialized to `0`. Let us now 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 gets the parameter `value` which we want to add. And then, we just have to write down the loop body Algorithm 1. 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 [191] 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. And that's already it.

We now can place the last line of the algorithm that returns the final result of the summation into the new method `result`. And we are done. We now have translated a non-trivial mathematical algorithm into a piece of `Python` code.

In Listing 11.5, 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 summation function `fsum` from the `math` module. We find that all three summation 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. Sadly, if we throw in 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 $1 + 10^{-16}$, which, however, cannot be represented with the datatype `float`.

So it seems that `fsum` gives us the most exact result, while our `KahanSum` object is better than the built-in `sum`. The reason for this is in the implementation of `fsum`. It uses the algorithm by Shewchuk [123, 254] which dynamically allocates a list of helper variables to obtain a provably exact sum (within the range of `float`). Our `KahanSum`, however, uses only three variables for the whole summation process. It does not dynamically allocate more memory. Also, the number of steps it performs are constant for each addition, whereas the algorithm by Shewchuk needs to do a number of steps that depends on the current length of its internal datastructure. Thus, `KahanSum` is indeed a very nice compromise solution, which offers higher precision than normal summing but retains the same 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, which 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. First, we can semantically group data and the operations on the data together. Second, we can create define an interface, i.e., an API, that consists of multiple operations and implement it in different ways. For the first case, we have seen two examples in this section.

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 it is generally a good idea to make objects *immutable*. If 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 their value. It cannot happen that one part of our program has a reference to a `Point` variable and "thinks" that it's coordinates are $(0, 1)$, but some other code changed the coordinates

Listing 11.4: 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

>>> sum([1e36, 1e18, 1, -1e36, -1e18])
0.0
>>> KahanSum().add(1e36).add(1e18).add(1).add(-1e36).add(-1e18).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) -> "KahanSum":
        """
        Add a value to the sum.

        :param value: the value to add
        :returns: the sum itself
        """
        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.
        return self  # Return `self` so that we can chain calls.

    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 11.5: Some use cases of our Kahan-summation class from Listing 11.4 and a comparison with Python's `sum` and the function `fsum` from the `math` module. (stored in file `kahan_user.py`; output in Listing 11.6)

```python
"""Examples for using our class :class:`KahanSum`."""

from math import fsum  # An (even more) exact summation algorithm.

from kahan_sum import KahanSum

# 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 11.6: The stdout of the program `kahan_user.py` given in Listing 11.5.

```
======= 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)  = 0.0
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
```

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 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 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.

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

12.1 A Hierarchy of Geometric Objects

We already learned that classes are a proper tool four grouping data and operations on the data together into one semantic unit. Classes also offer the concept of inheritance, which basically means "specialization". This concept is explored by a more elaborate example illustrated in Figure 12.1, which we will now discuss step-by-step. Imagine that we would wanted to represent all geometrical shapes in a two-dimensional plane. Each shape has an associated area as well as a perimeter. We could create a class `Shape` that provides two methods, `area` and `perimeter`, returning the area in in area units and the perimeter length in length units, respectively.

A circle is a special shape. It does have a center as well as a radius. Knowing these two attributes, we can compute the area and perimeter. If we wanted to express this in terms of classes, we could say that the class `Circle` is a special subclass of class `Shape`. It would have two attributes, `center` (which could be an instance of `Point`) and `radius`, which would be a `float`. The methods `area` and `perimeter` could then be implemented appropriately and use these attributes.

This is already how inheritance works: If we want that a new class `Circle` inherits from a class `Shape`, all we have to do is to write `class Circle(Shape):` instead of just `class Circle:` when declaring the new circle class. Of course, we first need to create the class `Shape`.

We do this in Listing 12.1. In this listing, we define `Shape` as a new class. We want that this

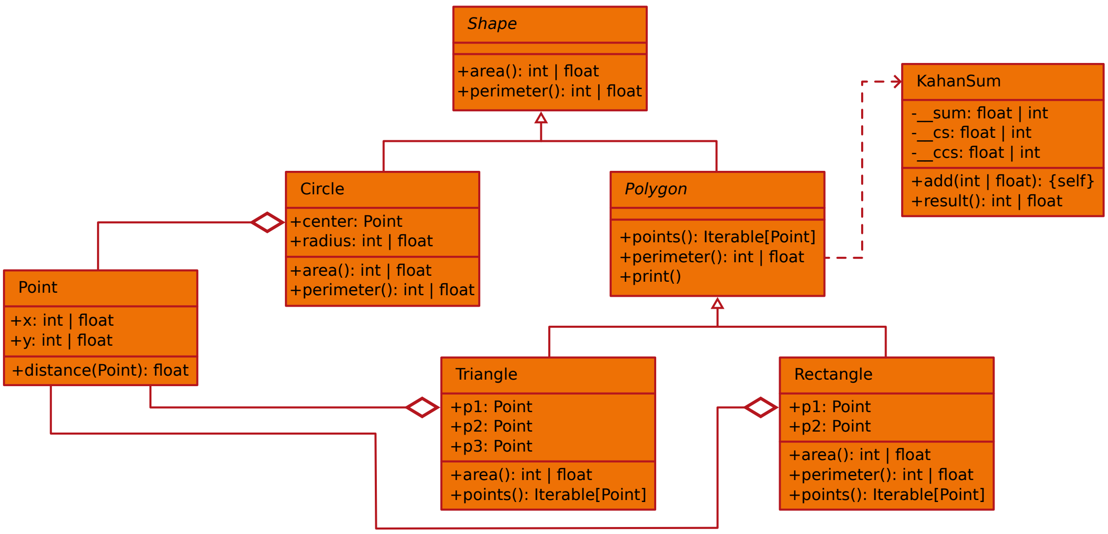


Figure 12.1: An example for class inheritance. The base class `Shape` offers two abstract methods, `area` and `perimeter`. The class `Circle` inherits from `Shape` and implements these methods. It also has an attribute `center`, which is an instance of `Point`, and an attribute `radius`. The class `Polygon` 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`.

Listing 12.1: A class for representing shapes in the two-dimensional Euclidean plane. (src)

```
"""A base class for shapes."""


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
        """
        raise NotImplementedError  # must be implemented by subclasses

    def perimeter(self) -> int | float:
        """
        Get the perimeter of this shape.

        :return: the perimeter of this shape
        """
        raise NotImplementedError  # must be implemented by subclasses
```

class has two methods, `area` and `perimeter`. The class `Shape` is not intended for being instantiated. Instead, we just want it to be the base class for "special" shapes. It does not have any attribute. 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.

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 Listing 12.2, we define the class `Circle`. By writing `class Circle(Shape)`, we declare it as a subclass of `Shape`. Its initializer `__init__`, we supply two parameters, `center`, which is an instance of our class `Point` from Listing 11.1, and `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. This ensures that we indeed 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 can now implement the function `area` to return π`radius`^{2} and `perimeter` to return 2π`radius`. With this, the complete interface defined by the superclass `Shape` is now filled with meaning.

In Listing 12.3, we explore how this new class can be used. We create the new instance `circ` of `Circle` by providing the `point=Point(2, 3)` and `radius=4`. These parameters are indeed reflected by the corresponding attributes. We also 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`.

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. In 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 datastructure for arbitrary polygons here. So the method `points` raises an `NotImplementedError` and thus must be implemented by the subclasses that we will develop later.

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 `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 distance 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. 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 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 11.4, 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, because 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.

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`.

A rectangle can be defined using its bottom-left and top-right corner point. In the initializer `__init__` of the class `Rectangle`, we pass in two points `p1` and `p2`. We raise a `ValueError` if these would form an empty rectangle. Otherwise, we store the minimum x- and y-coordinate in the bottom-left point attribute `p1` and 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`.

The area of the rectangle is easily computed. While we already inherit a perfectly fine method `perimeter` computing the perimeter of our rectangle from `Polygon` 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 length of the horizontal and vertical side length of our rectangle. The inherited `perimeter` method would instead iterate over all four points and compute Euclidean distances using `sqrt`, which is both slower and less accurate (especially if our coordinates would be `ints`).

The class `Triangle` is implemented in very much the same fashion. This time, we need to store all three corner points (and also return all of them in the `points` method implementation). There also 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 $A = x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2)$ you probably remember from high 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.

12.2 Summary

In this section, we have discussed how inheritance of 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 constructed 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 API would basically act as a blank canvas. It would probably provide a method to draw a rectangle with certain coordinates in a certain color. It would probably provide another method to draw a line of a certain width and color connecting two points. You could define this as a class with (at least) two methods `draw_line` and `draw_rectangle`. We could then go an implement these methods in a class in such a way that constructs an SVG graphic [71] in memory. Another implementation could instead construct an Adobe PDF [211, 318] graphic. While doing this would be more complex and exceed the space that we can reasonable use for an example in this book, the principle of the approach would not be very different from what you have already learned here.

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)

```python
"""Examples for using our class :class:`Circle`."""

from circle import Circle
from point import Point
from shape import Shape

circ: Circle = Circle(Point(2, 3), 5)
print(f"                  center: ({circ.center.x}, {circ.center.y})")
print(f"                  radius: {circ.radius}")
print(f"               perimeter: {circ.perimeter()}")
print(f"                    area: {circ.area()}")
print(f"isinstance(circ, Circle): {isinstance(circ, Circle)}")
print(f" isinstance(circ, Shape): {isinstance(circ, Shape)}")
```

↓ `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
```

Listing 12.5: Polygons are special shapes delimited by straight lines between corner points. (src)

```python
"""A polygon is a figure described by 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
        """
        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.
        return total.add(previous.distance(first)).result()  # Add last.

    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 is 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)

```python
"""
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
```

Chapter 13

Dunder Methods

In Python, *everything is an object* [141, 202]. 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`:

- `str(o)` should return a concise and brief representation of the object `o`. This representation is mainly for end users. `str(o)` invokes the `__str__` dunder method, if it has been implemented. Otherwise, `__repr__` is used instead.

- `repr(o)` should ideally return a string representation that contains all the information that is needed to re-create the object. 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 Listing 13.1. Here, we first create an integer variable `the_int` with value `123`. Both `str(the_int)` and `repr(the_int)` 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, which is equivalent to `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 are converted to strings using `repr`, not `str` [272]. Otherwise, we could not distinguish `l1` and `l2` in the output.

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)

```python
"""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) = '2025-04-01 06:16:22.930680+00:00'
     right_now   =  2025-04-01 06:16:22.930680+00:00
repr(right_now) = 'datetime.datetime(2025, 4, 1, 6, 16, 22, 930680, tzinfo=
   ↪ datetime.timezone.utc)'
     right_now   = datetime.datetime(2025, 4, 1, 6, 16, 22, 930680, tzinfo=
       ↪ datetime.timezone.utc)
```

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 `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 either do this using the format specifier `!s` or by printing the result of `str(o)`. The former variant is usually preferred. Anyway, we find that the simple string 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 using the format specifier `!r` or by printing the result of `repr(o)`. Doing this with a `datetime`

[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 0x7f1c25589c40>'
     p1   = <point.Point object at 0x7f1c25589c40>
repr(p1) = '<point.Point object at 0x7f1c25589c40>'
     p1   = <point.Point object at 0x7f1c25589c40>
(p1 is p2) = False
(p1 is p3) = False
(p1 == p2) = False
(p1 == p3) = False
(p1 != p2) = True
(p1 != p3) = True
(p1 == 5) = False
```

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 re-create the `right_now` object with the same data. 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, 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, 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., only identical instances are equal. 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.

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 [299]. However, this is not necessarily always the case[2]. Therefore, Python also allows us to implement an `__ne__` dunder method to realize inequality differently or, potentially, more efficiently, instead [299].

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. The default equality comparison only checks for identity. If implement `__eq__` by ourselves, this method should clearly return a value that makes `p1 == 5` become `False` as well. 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 Listing 13.5. The concise string representation returned by `__str__` will just be the point coordinates 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.
>
> — *[39], 2001*

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`.

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.

[2] In [299], 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 [136], maybe with the exception of signaling `nans`, which does not matter in Python. Maybe it was true for some Python implementations back then, as [309] indicates.

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 and 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 would like to construct a set of points. For this to be possible, two things are needed:

- A dunder method `__eq__` that compares two `Points` for equality. This, we already have done in Listing 13.5.
- A dunder method `__hash__` that returns the hash value of a `Point` in form of an `int`.

For these two methods, it must hold that [201]

$$\texttt{a.\_\_eq\_\_(b)} \Rightarrow \texttt{a.\_\_hash\_\_()} = \texttt{b.\_\_hash\_\_()} \tag{13.1}$$

This is equivalent [40, 201] 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 [107, 179]. Sets are basically the same, but only store the keys. The internal tables could be represented as 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 [66, 155, 257] are very fast. They have an element-wise access/update time complexity of $\mathcal{O}(1)$ [6, 131, 194]. As you know, lists can be indexed by integers. 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.

Luckily, `__hash__` does not need to compute a valid index into such a table. That would indeed not be possible: The `__hash__` method of an object `a` cannot know the length of the internal table underlying a dictionary or set where the object `a` will be stored. Instead, all it needs to do is to return one integer value that represents `a`. The dictionary/set implementation is then responsible for translating this integer into a valid index into its internal table. It can use modulo division for this. It will also do lots of other things, e.g., taking care of collisions, i.e., cases where two different objects have the same hashes (or different hashes that end up mapping to the same index) [107, 179]. None of this is important here.

What is important is that the hash values are needed to find the objects in the dictionaries and set. 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] It 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, just use key-value relationships, where the hash values of the keys are computed to find the right places in the internal tables. The details are (for the third time), unimportant here – but you can read them in these very interesting sources here [107, 179].

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 20 and 23).

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 constant `"123"`, but if we read the string `"123"` from a file, `"123" in s` could return `False`. That would make no sense at all.

With that out of the way, we can now make our `Point` class hashable. In Listing 13.8, we modify the `Point` class from Listing 13.5. We retain the implementation of `__eq__` and add the method `__hash__`.

[3] This works the same way in which `repr(a)` would invoke `a.__repr__()` if it is defined.

> ...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.
>
> — *[201], 2001*

Best Practice 58 For implementing `__eq__` and `__hash__`, the following rules hold [201]:

- 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.
- Then, 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.

That sounds complicated, but is actually very easy. 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)`.

One may be a bit scared regarding integers and floats. We permitted the coordinates of our points to be either `ints` or `floats`. Since `5.0 == 5` holds, we could feel anxious whether `hash(5.0)== hash(5)` would also hold. If *not*, then something like `hash((5.0, 3))!= hash((5, 3))` could happen, which could lead to `Point(5.0, 3).__hash__()!= Point(5, 3).__hash__()` while `Point(5.0, 3).__eq__(Point(5, 3))` is `True`. This would then violate Best Practice 58.

If that would happen, we could create sets that contain equal elements multiple times. Which, in turn, would violate the definition of sets. That would be a very counter-intuitive bug in our code. Actually, I secretly planned to use this as a very tricky example for learning how to use the debugger... 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`).
>
> — *[41], 2001*

Therefore, we can indeed implement `__hash__` with a single line of code in Section 13.2. And I will later find another example on how the debugger can be used to spot errors in code.

We use our new variant of the `Point` class in Listing 13.9. 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 `y`-coordinate of `p1` is an `int` and the one of `p2` is a `float` (but with the same value). When we create the set `points` as `{p1, p2, p3}`, it will have 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` 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 5.4 can be used without problems.

Listing 13.8: Our `Point` class, extended with the `__eq__` and `__hash__` dunder methods. (src)

```python
"""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`.

        :return: the hash code

        >>> hash(Point(4, 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 the 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.
```

↓ `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'}
```

13.3 Arithmetic Dunder and Ordering

Much of Python's syntactic behavior can be grounded on dunder methods. Indeed, even the arithmetic operators `+`, `-`, `*`, and `/`. This allows us to define new numerical types if we want.

Here, we will do exactly 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 $\frac{a}{b}$.

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` in our `Fraction` class. We want our numbers to be immutable, because like you cannot change the value of `5`, you should also not be able to change the value of `1/3`. The attributes will therefore receive the type hint `Final[int]` [274].

Our fractions should be *canonical*. It is totally possible that two fractions $\frac{a}{b} = \frac{c}{d}$ with $a \neq c$ or $b \neq c$. 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. 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 it should work, but it is better to verify that $\frac{0}{-9}$ is indeed mapped to $\frac{0}{1}$. 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 59** Constants are module-level variables which must be assign a value upon definition and which must be annotated with the type hint `Final` [274].

> **Best Practice 60** The names of constants contain only capital letters with underscores separating words. Examples include `MAX_OVERFLOW` and `TOTAL` [302].

> **Best Practice 61** Constants are documented by writing a comment starting with `#:` immediately above them [266].

[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` [101].

Listing 13.11: Part 1 of the `Fraction` class: The initializer `__init__` and global constants. (src)

```python
"""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)

```python
    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})"
```

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 such that it returns `self.a` as string if the denominator is one, i.e., if `self.b == 1`. 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)`, i.e., fractions that represent integer numbers will produce the same strings as 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 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`, meaning 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__` method. From primary school, we remember that $\frac{a}{b} + cd = \frac{a*d+c*b}{b*d}$. Therefore, if `other` is also an instance of `Fraction`, `__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`, because this would enable Python to look for other routes to perform addition with our objects.[5] The behavior of this method is again be tested with doctests. These check that $\frac{1}{3} + \frac{1}{2}$ actually yields $\frac{5}{6}$ and that $\frac{1}{2} + \frac{1}{2}$ really returns $\frac{1}{1}$. They

[5] Python would then look whether `other` provides an `__radd__` method that does not return `NotImplemented` ... but we will not implement all possible arithmetic dunder methods here so we skip this one.

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)
```

also check correct normalization by trying $\frac{21}{-12} + \frac{-33}{42} = \frac{882+396}{-504} = \frac{1278}{-504} = \frac{18*1278}{18*-28} = \frac{-71}{28}$.

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, $\frac{a}{b} - cd = \frac{a*d-c*b}{b*d}$. 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 a `__mul__`, if implemented, and that `/` operation uses `__truediv__`. Multiplying the fractions $\frac{a}{b}$ and $\frac{c}{d}$ yields $\frac{a*c}{b*d}$. Dividing $\frac{a}{b}$ by $\frac{c}{d}$ yields $\frac{a*d}{b*c}$. 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`. If present, `abs(x)` will invoke `x.__abs__()`. 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 by simply flipping the sign of it.

Finally, in Listing 13.15 we implement the six rich comparison dunder methods given in [299]:

- `__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 only that `x == y` 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 $\frac{a*d}{b*d} < \frac{c*b}{b*d}$, which 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 doctests. You should never omit such checks and tests, because in program code, you *do* have space. 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.

Listing 13.15: Part 5 of the `Fraction` class: All order-related dunder methods. (src)

```python
    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 0.02s ===========================
# pytest 8.3.5 with pytest-timeout 2.3.1 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. 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 like `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 Listing 13.17 we present 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 function `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 `negative` otherwise. We then make sure that `a` is positive by computing `abs(a)`. Then we also copy the denominator into a variable `b`. We will not change the local variable `b` in our function, so we mark it with the type hint `Final`.

> **Best Practice 62** Every time you declare a variable that you do not intend to change, mark it with the type hint `Final` [274]. On 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.

We now want to fill a list `digits` with the digits representing the fraction in a `while` loop. 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 add 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, so `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)` gives us 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 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, which was the case in our example. It also stop if we reach the maximum number of fractional digits, i.e., if `len(digits)` exceeds `max_frac`.

For example, 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. 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, 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.
```

we increment the last digit by 1.[6] 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 decimals. All we need to do is to convert these to a string and return them to the user.

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 "." 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]`.

[6] This is an error, as we will later see.

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 ________
038          '0.5'
039          >>> Fraction(1, 3).decimal_str(10)
040          '0.3333333333'
041          >>> Fraction(-101001, 100000000).decimal_str()
042          '-0.00101001'
043          >>> Fraction(1235, 1000).decimal_str(2)
044          '1.24'
045          >>> Fraction(99995, 100000).decimal_str(5)
046          '0.99995'
047          >>> Fraction(91995, 100000).decimal_str(3)
Expected:
    '0.92'
Got:
    '0.9110'

/home/runner/work/programmingWithPython/programmingWithPython/__git__/git/git__mwsh3eb/dunder/fraction_decimal_str_err.py:47: DocTestFailure
======================== short test summary info ========================
FAILED fraction_decimal_str_err.py::fraction_decimal_str_err.Fraction.decimal_str
=========================== 1 failed in 0.01s ===========================
# pytest 8.3.5 with pytest-timeout 2.3.1 failed with exit code 1.
```

If the fraction was `negative`, we place a minus sign before the string, 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 a generator that applies the `str` function to all the elements in `digits`. `str` applied to a string just returns the string itself. Applied to an integer, it converts it to a string.

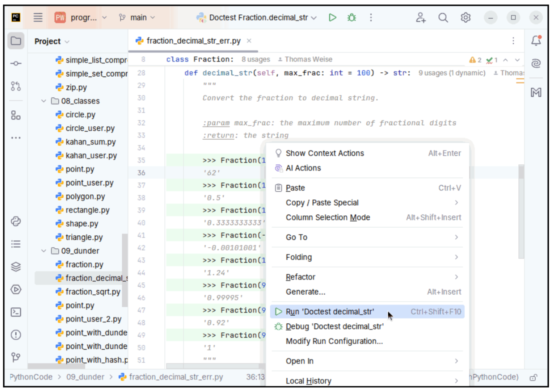
The method `join` of a string concatenates all elements of the sequence it receives as parameter and places the string itself as separator. For example, `"X".join(["a", "b", "c"])` would yield `"aXbXc"`. We use the empty string for joining, so for our example, we finally return `-11.1875`.

After finishing the implementation of `decimal_str`, what remains to do is to test it. We do this by adding several standard and 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`.

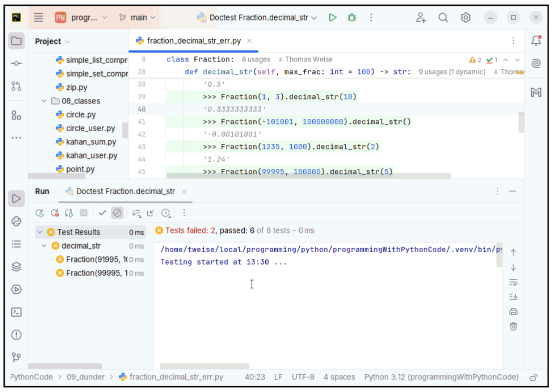
As fraction that cannot be exactly written down as finite decimal string, we choose $\frac{1}{3}$. $\frac{1}{3}$ to ten digits should yield `"0.3333333333"`. As example for negative fractions and also as an example for fractions with multiple leading zeros, we use $\frac{-101001}{100000000}$. This should 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 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"`.

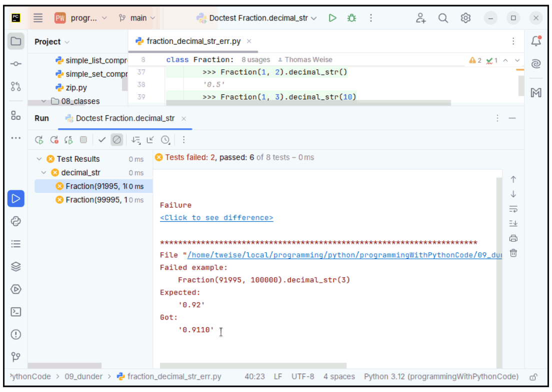
Computing `Fraction(91995, 100000).decimal_str(3)` means rounding 0.91995 to three decimals. The last digit, a 5, would be cut off. This means that we need to round-up, which would make



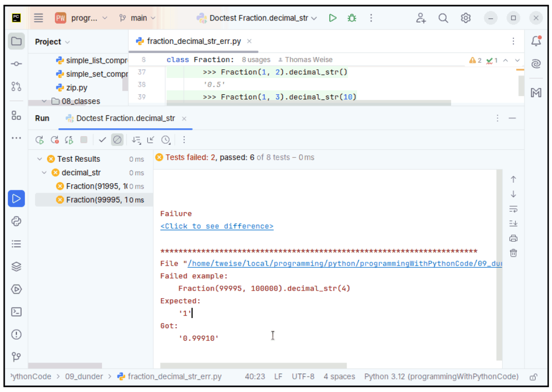
(13.1.1) We open a contact menu by right-clicking into out function. We left-click on Run 'Doctest decimal_str'.



(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`.

the second-to-last 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 toggle, finally resulting in a `"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 and scroll to our function `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 program would do. Would it not be nice if we could actually step-by-step execute the program for the test and see what it *actually* does? Luckily, we can do that! With a tool called debugger which ships with `Python` and `PyCharm`.

Useful Tool 8 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 [3, 240, 322]. A comprehensive example on how to use the debugger in `PyCharm` is given in Section 13.4.

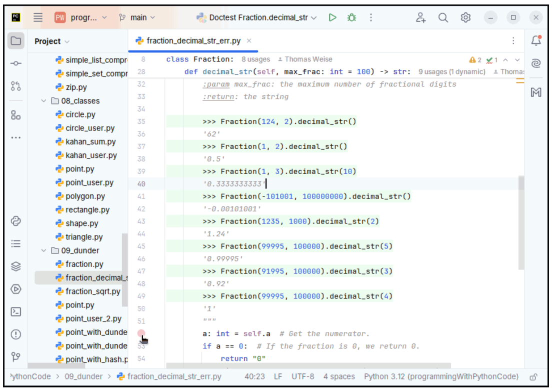
In `PyCharm`, we can apply the debugger to a complete program, but also to doctests. This is what we will do in Figure 13.2. First, we again open our program code in `PyCharm` and locate our function `decimal_str` (Figure 13.2.1).

On the left side of our code file, 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`, so be place the breakpoint there (Figure 13.2.2). The breakpoint is shown as a red ball over the line number.

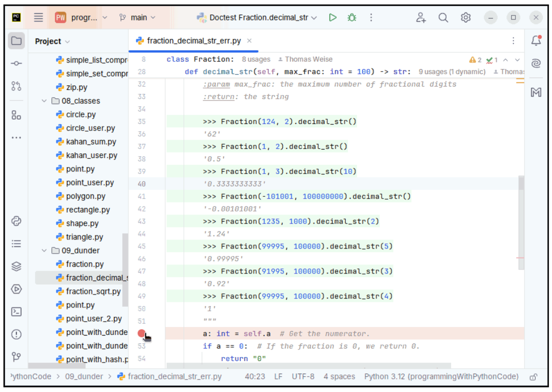
In order to begin the debugging process, we again open the context menu by right-clicking into the doctest. This time, instead of *running* the doctest, we click Debug 'Doctest decimal_str' (Figure 13.2.3). The doctests will now be executed, but instead of running them completely, the debugger kicks 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 right-click the top of this row and 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).

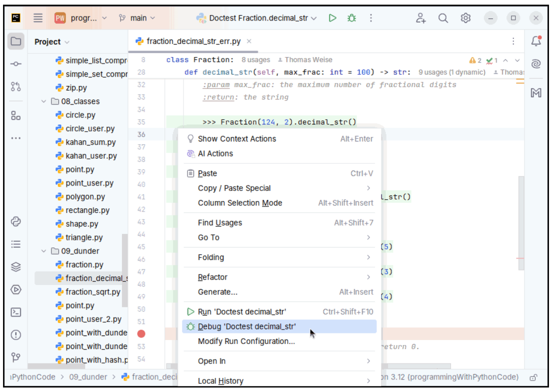
We see that `max_frac` has the value `100`. When clicking on the variable `self`, we see that the numerator `a` of the current fraction has value `62`, while the denominator `b` is `1`. This is exactly what we expect: Our first test case was `Fraction(124, 2).decimal_str()`, so the normalized fraction is indeed $\frac{62}{1}$.



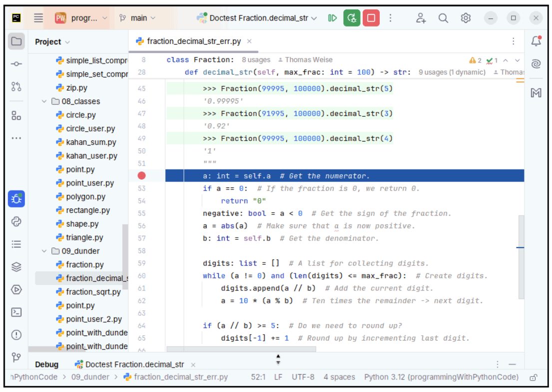
(13.2.1) We open our source code file in `PyCharm` and locate our function `decimal_str`.



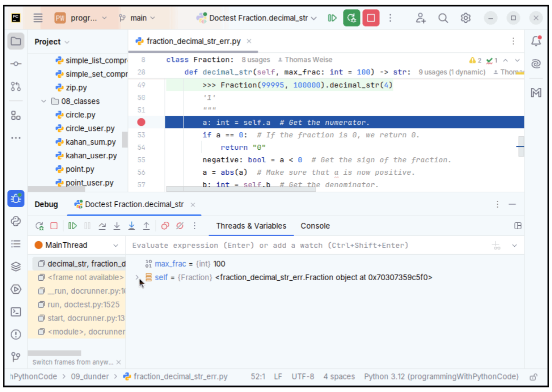
(13.2.2) We place a breakpoint at the first line of our function by right-clicking on the line number. This is denoted by the red ball over the line number. Later, the program execution will pause at this location.



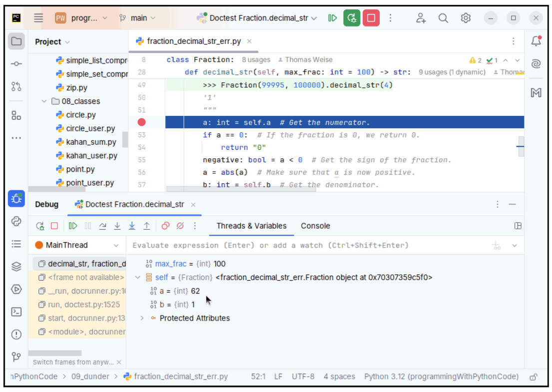
(13.2.3) We open the context menu by right-clicking into the doctest and click [Debug 'Doctest decimal_str'].



(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`.

This test case is already successful, so we are not interested in it. Among the symbols in Debug register, we click ⏵, 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, but 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)`. We again continue by clicking ⏵ or pressing F9. 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). 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), which we skip, too. 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.

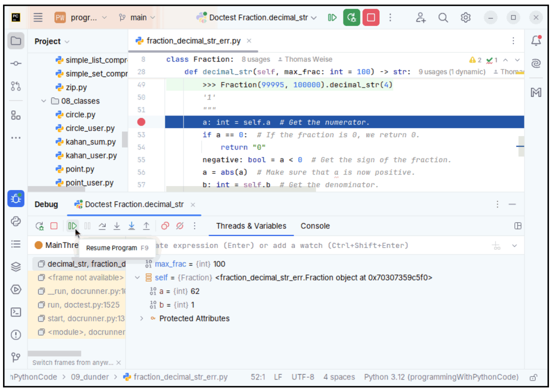
Figure 13.2.13 shows that we now arrived at the beginning of the failing doctest case `Fraction(91995, 100000).decimal_str(3)`. The fraction $\frac{91995}{100000}$ got normalized to $\frac{18399}{20000}$ in the initializer `__init__`. The parameter `max_frac` of `decimal_str` has the value `3`, as we 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 function. 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.

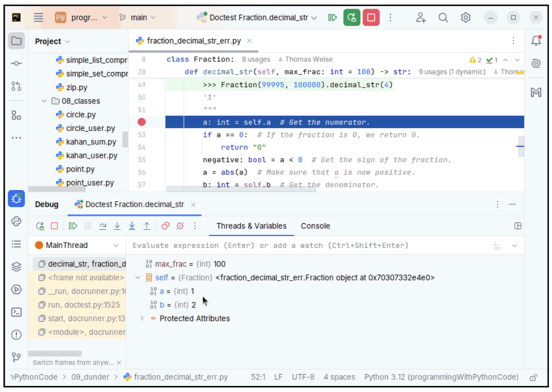
By pressing F8, the `if a == 0:` is executed. Since `a == 0` is not `True`, the body of the `if` is not executed. The program jumps right over it. The next line after the `if` is marked Figure 13.2.16. We execute it by pressing F8.

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 (Figure 13.2.17).

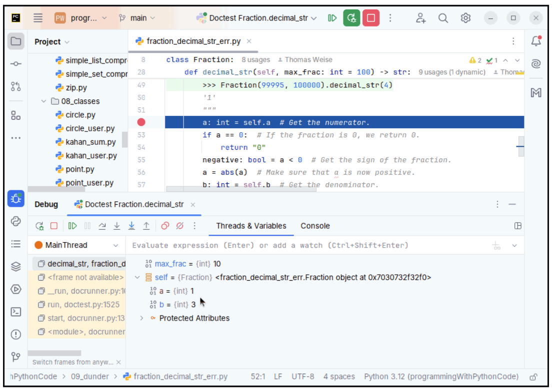
`a = abs(a)` has no effect, since `a` is already positive. We press F8 to continue (Figure 13.2.18).



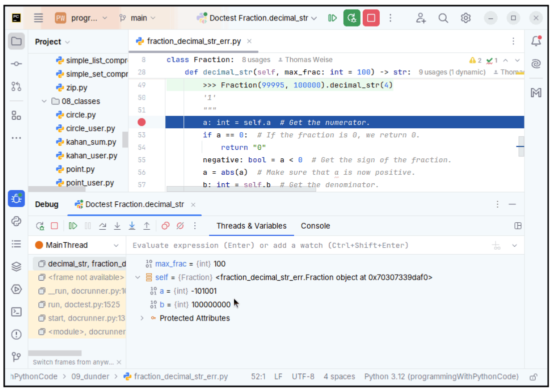
(13.2.7) The first doctest case $\frac{124}{2}$ is uninteresting, so we continue the program execution by clicking ▷ or hitting F9.



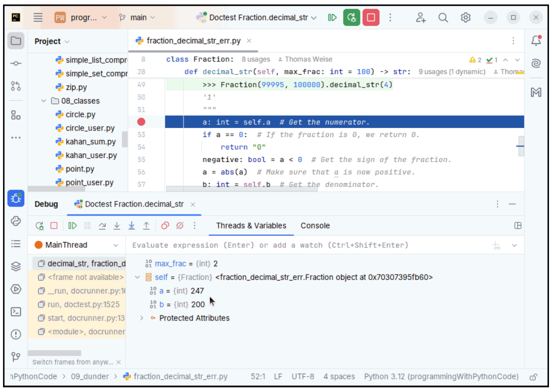
(13.2.8) The second doctest case $\frac{1}{2}$ is also uninteresting. We continue the program execution by clicking ▷.



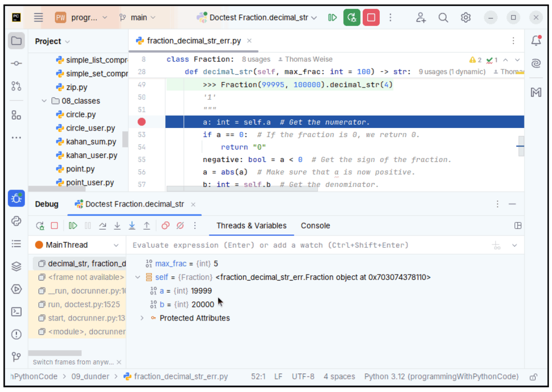
(13.2.9) The third doctest case, where the fraction is $\frac{1}{3}$ and `max_frac` is `10`. We again skip over it by pressing F9.



(13.2.10) The fourth doctest case $\frac{-101001}{100000000}$ can be skipped as well.

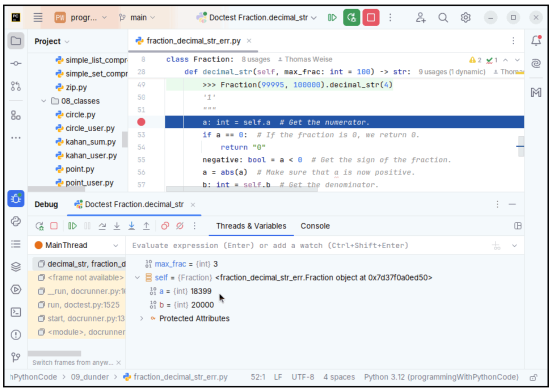


(13.2.11) The test case $\frac{1235}{1000}$, normalized to $\frac{247}{200}$, also was successful and can be skipped by pressing ▷.

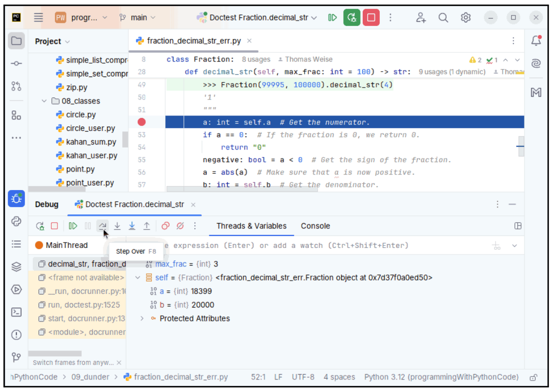


(13.2.12) The last one of the uninteresting doctest cases: `Fraction(99995, 100000)`. We again press F9.

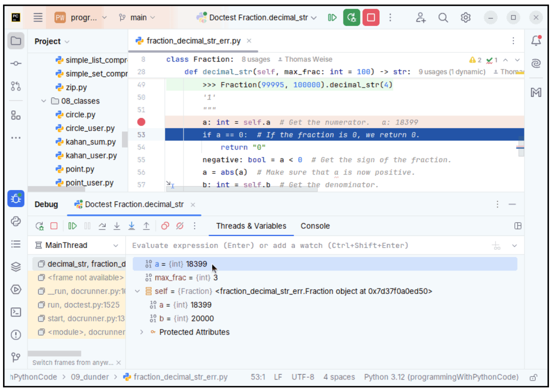
Figure 13.2: Using the debugger in `PyCharm` (continued).



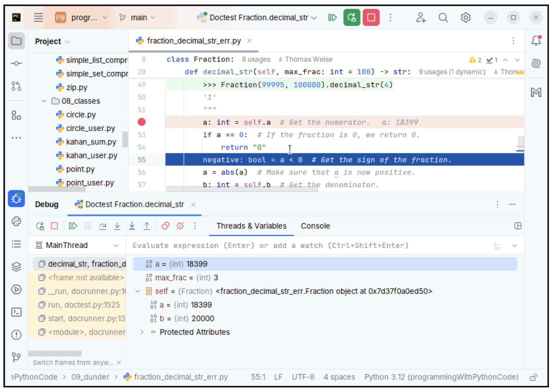
(13.2.13) We arrive at the beginning of the failing doctest case `Fraction(91995, 100000).decimal_str(3)`. The `max_frac` parameter has value `3`, `self.a` is `18399` and `self.b` is `20000`, because the fraction was normalized in `__init__`.



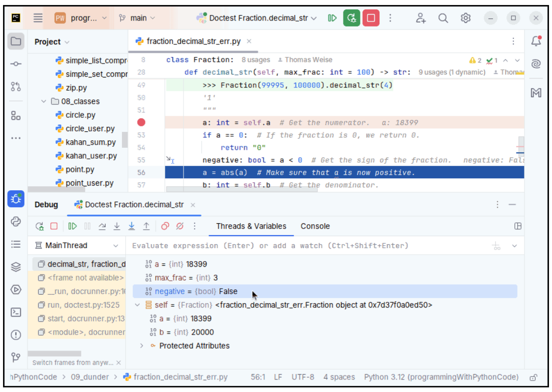
(13.2.14) We now execute the first line of the `decimal_str` function where the debugger has paused. This is done by either pressing the ⤼ button or by hitting F8.



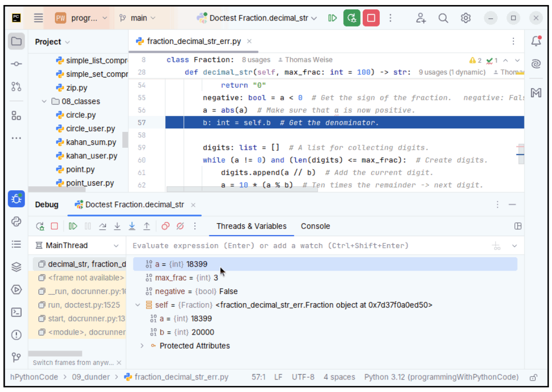
(13.2.15) The execution of the assignment `a = self.a` creates a new local variable `a` with value `18399`. We press F8 to continue the execution.



(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)` if 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, we again hit F8. 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`. Then, `9` gets appended to `digits` and `a` gets updated to `190000`.

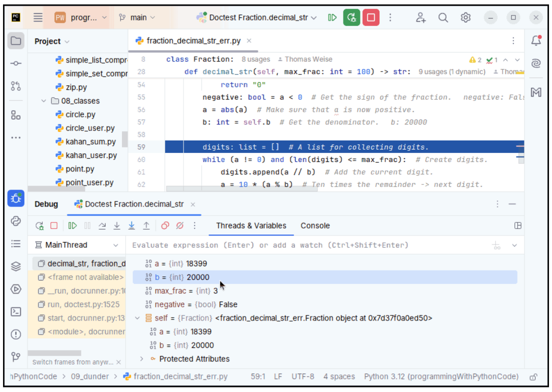
At this stage, `digits` has become `[0, 9, 1, 9]`. Since `max_frac` is `3`, `len(digits)<= max_frac` is no longer `True`. In Figure 13.2.32, we have arrived back at the head of the loop. When we hit F8, the loop condition is evaluated again, but this time it evaluates to `False`. The loop terminates and the cursor is placed on the next line of code after the loop.

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.

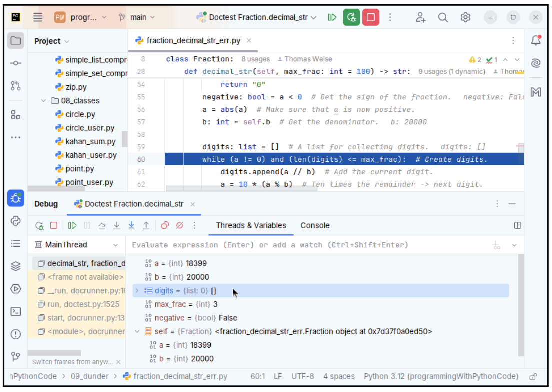
The next line of code, `if (a // b)>= 5`, is supposed to check whether we should round up the last digit. Since `a` is `190000` and `b` is still `20000`, `a // b` is `9`. So the condition should be met. In Figure 13.2.32, we press the ↷ button to find it out.

A look at Figure 13.2.35 reveals the bug 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]`.

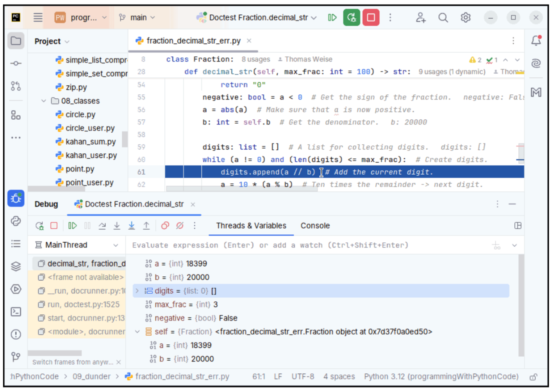
The strange trailing zeros in our output were not separate digits. They were the zeros of a ten. We did not consider that, when rounding up, we do not just have simple cases like 1.25 which we can round to 1.3 by only incrementing one digit. We can have cases like 0.9999, which rounds up to 1, even if we want three fractional digits of precision. We can stop the debugging here and go back to our code.



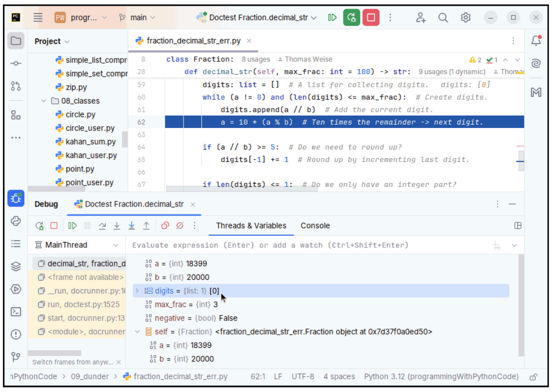
(13.2.19) After executing `b = self.b`, the new local variable `b` with value `20000` comes into existence. We continue debugging by pressing F8.



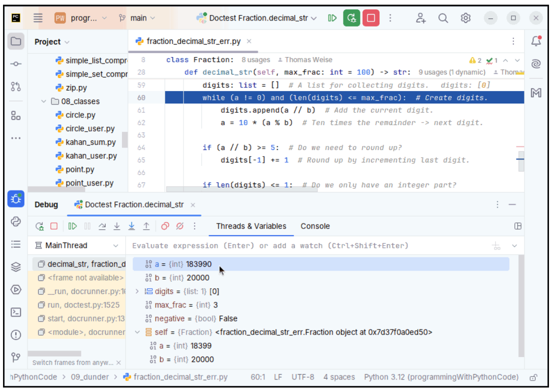
(13.2.20) The empty list `digits` has been created. By pressing F8, the executing `while` loop will begin by checking its condition.



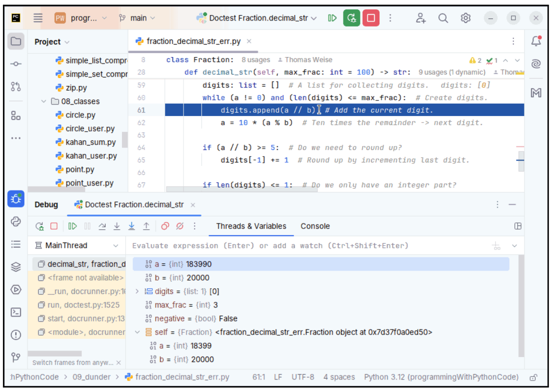
(13.2.21) The condition of the `while` loop is met. The first line of the loop's body is marked. We press ⏎ to execute it.



(13.2.22) `digits` now contains the result of `a // b`, i.e., is `[0]`. We press F8.

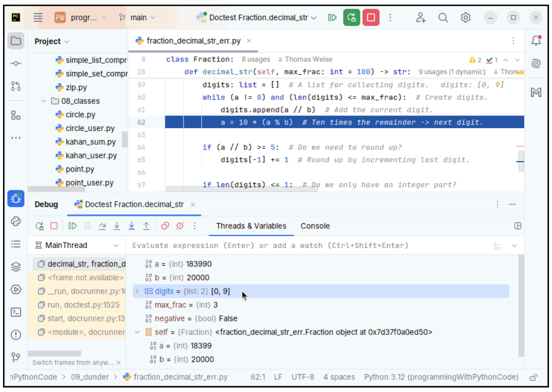


(13.2.23) `a` now is `183990`. We press ⏎ to continue.

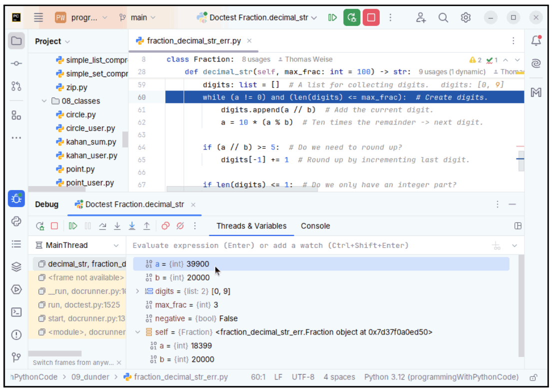


(13.2.24) We hit the F8 key to continue. The loop condition is still met, so the first line of the loop body is marked again.

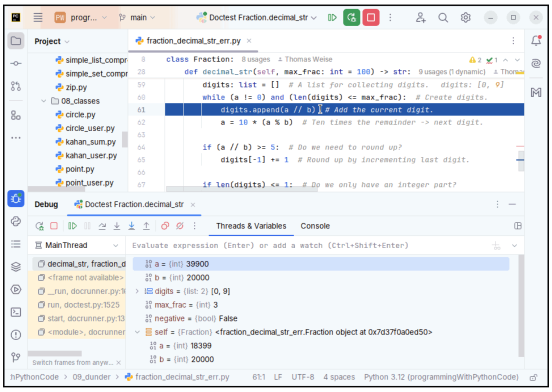
Figure 13.2: Using the debugger in `PyCharm` (continued).



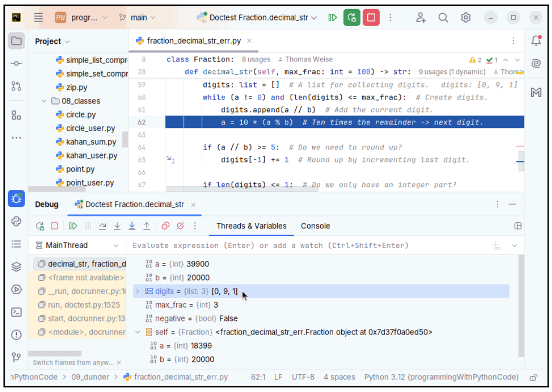
(13.2.25) `9` gets appended to `digits`.



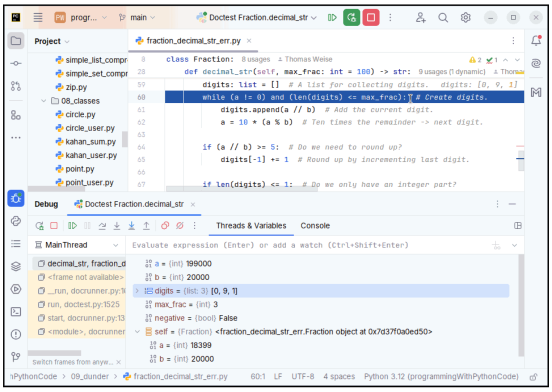
(13.2.26) `a` gets updated to `39900`.



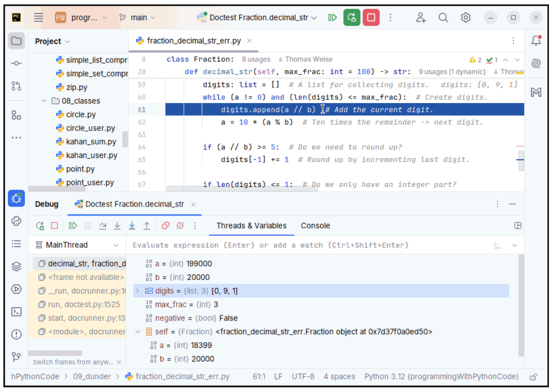
(13.2.27) The loop condition is still met.



(13.2.28) `1` gets appended to `digits`.

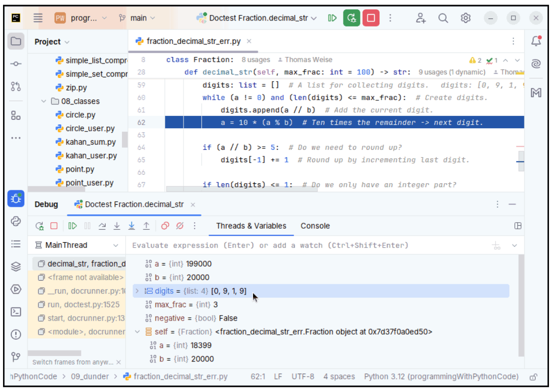


(13.2.29) `a` gets updated to `199000`.

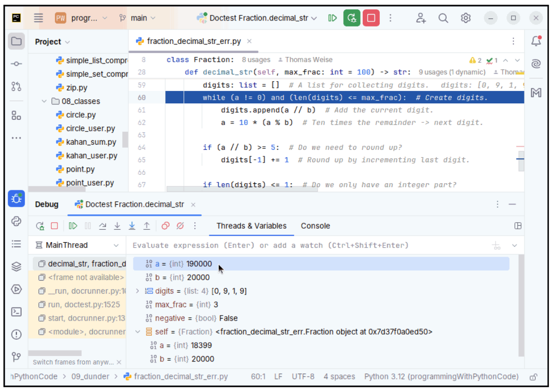


(13.2.30) The loop condition is still met.

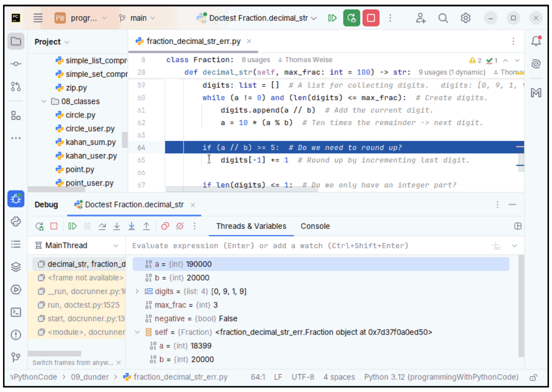
Figure 13.2: Using the debugger in PyCharm (continued).



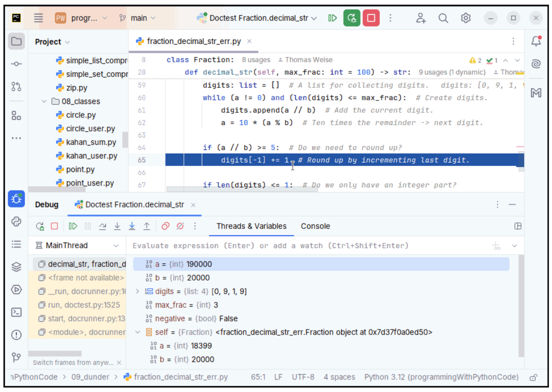
(13.2.31) `9` gets appended to `digits`.



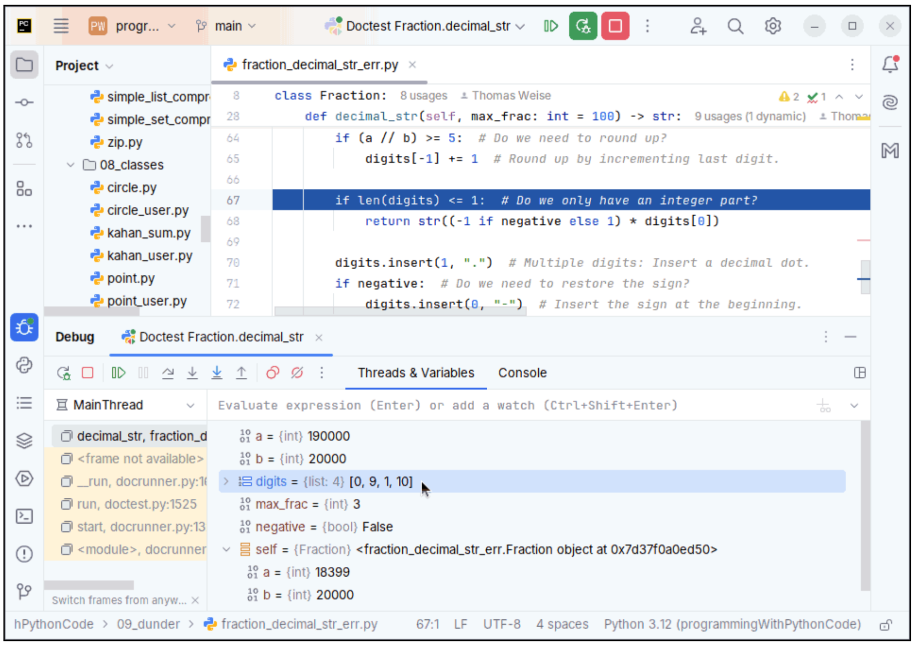
(13.2.32) `a` gets updated to `190000`.



(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, we just copy the code of the `sqrt` function from this file. 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, i.e., we cannot represent the exactly with an instance of `Fraction`. In the original code, due to the limited range of `floats`, this is not a problem and the function will eventually stop (when all of the 15 to 16 digits of precision are exhausted). However, using `Fraction`, the same code could loop forever, trying to find better and better approximations. So we need to limit the iterations with an additional parameter `max_steps: int = 10` limiting the number of cycles for our loop. 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 numbers like `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 woring with instances of `int` or `float`, but did not do it (again for the sake of brevity). Either way, all the numerical constants in Listing 13.20 are now instances of `Fraction`.

Our new function begins with an initial check whether the input number is negative and, if so, raises an `ArithmeticError`. Then it has basically the same loop and body as our original variant in Listing 8.5. The only difference is that we decrease `max_steps` in each iteration and `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 to a decimal string with *750 fractional digits*. We take the correct value from [190, 261]. This means 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, so the value of the fraction might not actually be 2. However, when rendered to 100 digits of precision, it should return `"2"`.

Finally, we want to compute the golden ratio ϕ [45, 89, 260], 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 [93].

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?

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)

```python
"""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)
    '1.4142135623730950488016887242096980785696718753769480731766797379\
9073247846210703885038753432764157273501384623091229702492483605585073\
2126441214970999358314132226659275055927557999505011527820605714701095\
9971605970274534596862014728517418640889198609552329230484308714321450\
3976260362799525140798968725339654633180882964062061525835239505474575\
2877599617298355752203375318570113543746034084988471603868999706990048\
5030544027790316454247823068492936918621580578463111596668713013015618\
6898723723528850926486124949771542183342042856860601468247207714358548\
4155657069677653720226485447015858801620758474922657226002085584466521\
5839889394437092659180031138824646815708263010059485870400318648034219\
8972782906410450726368813137398552561173220402 5'
    >>> sqrt(Fraction(4, 1)).decimal_str()
    '2'
    >>> (ONE_HALF * (ONE + sqrt(Fraction(5, 1)))).decimal_str(420)
    '1.6180339887498948482045868343656381177203091798057628621354486227\
0526046281890244970720720418939113748475408807538689175212663386222353\
9317931800607667263544333890865959395829056383226613199282902678806752\
8766892501711696207032221043216269548626296313614438149758701220340805\
8795445474924618569536486444924104432077134494704956584678850987433944\
2125448770664780915884607499887124007652170575179788341662562494075890 7'
    """
    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 8.3.5 with pytest-timeout 2.3.1 succeeded with exit code 0.
```

13.5 Implementing Context Managers for the with Statement Revisited

In Section 9.3.5, we introduced the with block, which is often used to ensure that resources are properly closed after using them. We used text files as 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 example, let us create a simple API that allows us to write output in a subset of the Extensible Markup Language (XML) format [35, 62, 156]. XML is a format for data interchange which was predominant in distributed systems the 2000s. After that, it began to fade out in favor [7] of JavaScript Object Notation (JSON) [34, 282] and YAML Ain't Markup Language™ (YAML) [61, 82, 156]. It is still very relevant today, for example, as foundation of several document formats such as those used in LibreOffice [104, 176] and Microsoft Word [81, 188], or as the basis for the Scalable Vector Graphics (SVG) format [71]. If you are a bit familiar with web design, then you will find that XML looks a bit like Hyper Text Markup Language (HTML) [126].

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 datastructure 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 [35].

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. 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 63** Methods that return `self` should be annotated with the type hint `Self` [269]. 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` [325]. 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[False]`.

> **Best Practice 64** If a parameter of a function can take on only some special constant values `X`, say certain integers or strings, the proper type hint is `Literal[X]` [171]. 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 [35] 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 [35].

Instead of writing to the `stdout`, we can also recall our very first example for the `with` statement back in Section 9.3.5: 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`. 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 [35]. 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 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)

```python
"""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. (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` [130]. 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__()`
  - `__float__` . . . convert to `float`: `float(a)` ≅ `a.__float__()`
  - `__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).

- `--`
  - Arithmetics, see Section 13.3 for selected examples
    - `__abs__` .... absolute value: `abs(a)` ≅ `a.__abs__()`
    - `__minus__` .... negate: `-a` ≅ `a.__minus__()`
    - `__pos__` .... positive: `+a` ≅ `a.__pos__()`
    - `__invert__` .... invert: `~a` ≅ `a.__invert__()`
    - `__add__` .... add: `a + b` ≅ `a.__add__(b)`
    - `__radd__` .... right-add: `a + b` ≅ `b.__radd__(a)`, if `a.__add__(b)` yields `NotImplemented`
    - `__sub__` .... subtract: `a - b` ≅ `a.__sub__(b)`
    - `__rsub__` ... right-subtract: `a - b` ≅ `b.__rsub__(a)`, if `a.__sub__(b)` yields `NotImplemented`
    - `__mul__` .... multiply: `a * b` ≅ `a.__mul__(b)`
    - `__rmul__` ... right-multiply: `a * b` ≅ `b.__rmul__(a)`, if `a.__mul__(b)` yields `NotImplemented`
    - `__truediv__` .... divide: `a / b` ≅ `a.__truediv__(b)`
    - `__rtruediv__` .... right-divide: `a / b` ≅ `b.__rtruediv__(a)`, if `a.__truediv__(b)` yields `NotImplemented`
    - `__mod__` .... modulo-divide: `a % b` ≅ `a.__mod__(b)`
    - `__rmod__` .... right-modulo-divide: `a % b` ≅ `b.__rmod__(a)`, if `a.__mod__(b)` yields `NotImplemented`
    - `__floordiv__` .... integer-divide: `a // b` ≅ `a.__floordiv__(b)`
    - `__rfloordiv__` ... right-integer-divide: `a // b` ≅ `b.__rfloordiv__(a)`, if `a.__floordiv__(b)` yields `NotImplemented`
    - `__pow__` .... exponential: `a ** b` ≅ `a.__pow__(b)`
    - `__rpow__` .... right-exponential: `a ** b` ≅ `b.__rpow__(a)`, if `a.__pow__(b)` yields `NotImplemented`
    - `__matmul__` .... matrix-multiply: `a @ b` ≅ `a.__matmul__(b)`
    - `__rmatmul__` .... right-ma, see Section 13.3trix-multiply: `a @ b` ≅ `b.__rmatmul__(a)`, if `a.__matmul__(b)` yields `NotImplemented`
    - `__divmod__` . compute the result of an integer division as well as the remainder and return them as 2-tuple: `divmod(a, b)` ≅ `a.__divmod__(b)`
    - `__rdivmod__` .... right-sided `divmod`: `divmod(a, b)` ≅ `b.__divmod__(a)`, if `a.__divmod__(b)` yields `NotImplemented`
    - `__round__` .... round: `round(a)` ≅ `a.__round__()`
    - `__trunc__` .... truncated to integer: `math.trunc(a)` ≅ `a.__trunc__()`
    - `__floor__` .... round towards $-\infty$: `math.floor(a)` ≅ `a.__floor__()`
    - `__ceil__` .... round towards $+\infty$: `math.ceil(a)` ≅ `a.__ceil__()`
  - In-Place Arithmetics
    - `__iadd__` .... in-place addition: `a += b` ≅ `a.__iadd__(b)`, returns `self`
    - `__isub__` .... in-place subtraction: `a -= b` ≅ `a.__isub__(b)`, returns `self`
    - `__imul__` .... in-place multiplication: `a *= b` ≅ `a.__imul__(b)`, returns `self`
    - `__itruediv__` .... in-place division: `a /= b` ≅ `a.__itruediv__(b)`, returns `self`
    - `__imod__` .... in-place modulo division: `a %= b` ≅ `a.__imod__(b)`, returns `self`
    - `__ifloordiv__` .... in-place integer division: `a //= b` ≅ `a.__ifloordiv__(b)`, returns `self`
    - `__ipow__` .... in-place exponentiation: `a **= b` ≅ `a.__ipow__(b)`, returns `self`
    - `__imatmul__` .... in-place matrix multiplication: `a @= b` ≅ `a.__imatmul__(b)`, returns `self`

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 over

- Logical or Binary Operators, see Section 3.2.2
  - `__and__` .... and: `a & b` ≅ `a.__and__(b)`
  - `__rand__` .... and: `a & b` ≅ `b.__rand__(b)`, if `a.__and__(b)` yields `NotImplemented`
  - `__or__` .... or: `a | b` ≅ `a.__or__(b)`
  - `__ror__` .... or: `a | b` ≅ `b.__ror__(b)`, if `a.__or__(b)` yields `NotImplemented`
  - `__xor__` .... exclusive or: `a ^ b` ≅ `a.__xor__(b)`
  - `__rxor__` .... exclusive or: `a ^ b` ≅ `b.__rxor__(b)`, if `a.__xor__(b)` yields `NotImplemented`
  - `__rshift__` .... shift right: `a >> b` ≅ `a.__rshift__(b)`
  - `__rrshift__` .... shift right: `a >> b` ≅ `b.__rrshift__(b)`, if `a.__rshift__(b)` yields `NotImplemented`
  - `__lshift__` .... shift left: `a << b` ≅ `a.__lshift__(b)`
  - `__rlshift__` .... shift left: `a << b` ≅ `b.__rlshift__(b)`, if `a.__lshift__(b)` yields `NotImplemented`
- In-Place Logical or Binary Operators
  - `__iand__` .... and: `a &= b` ≅ `a.__iand__(b)`, returns `self`
  - `__ior__` .... or: `a |= b` ≅ `a.__ior__(b)`, returns `self`
  - `__ixor__` .... exclusive or: `a ^= b` ≅ `a.__ixor__(b)`, returns `self`
  - `__irshift__` .... shift right: `a >>= b` ≅ `a.__irshift__(b)`, returns `self`
  - `__ilshift__` .... shift left: `a <<= b` ≅ `a.__ilshift__(b)`, returns `self`

Figure 13.5: An overview of the dunder methods in Python (Part 3).

sets to lists. Maybe we want to implement our own collection? It maybe could be a tree datastructure, 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 `ints`. 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 [130].

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 [283], 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 [139, 216].

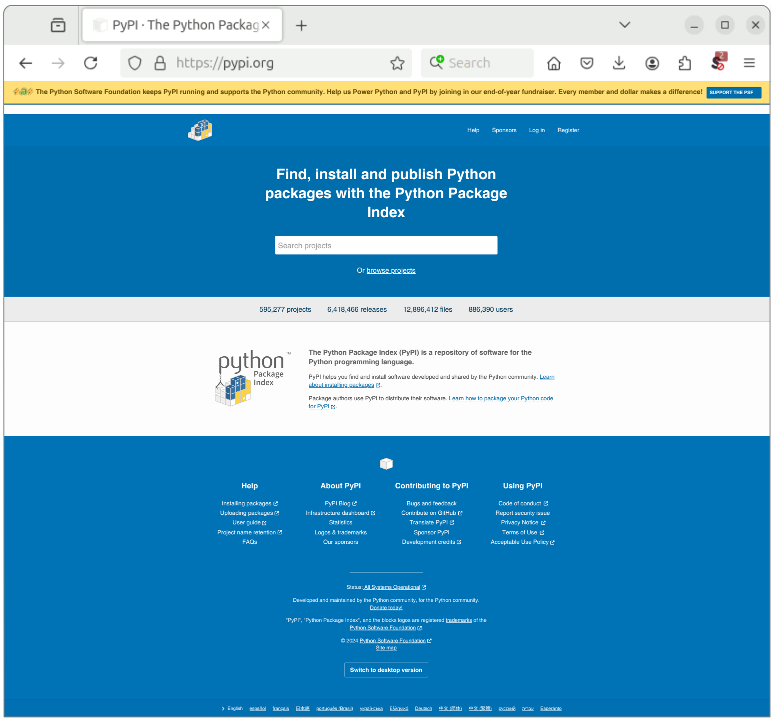


Figure 14.1: A screenshot of the website https://pypi.org, the Python Package Index [283], taken on 2024-12-24.

Listing 14.1: A small Python script using the NumPy library. (src)

```
"""An example for using numpy."""

import numpy as np

print(f"look, a numpy array: {np.array([1, 2, 3.0])}")
```

Useful Tool 9 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 [217].

pip should always be used in a virtual environment, see Best Practice 65 and 66.

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 [187, 308]. 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 65 Packages should *always* be installed in virtual environments and never system-wide (maybe with the exception of pip and venv).

Best Practice 66 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 10 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 [75, 119, 143]. 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.

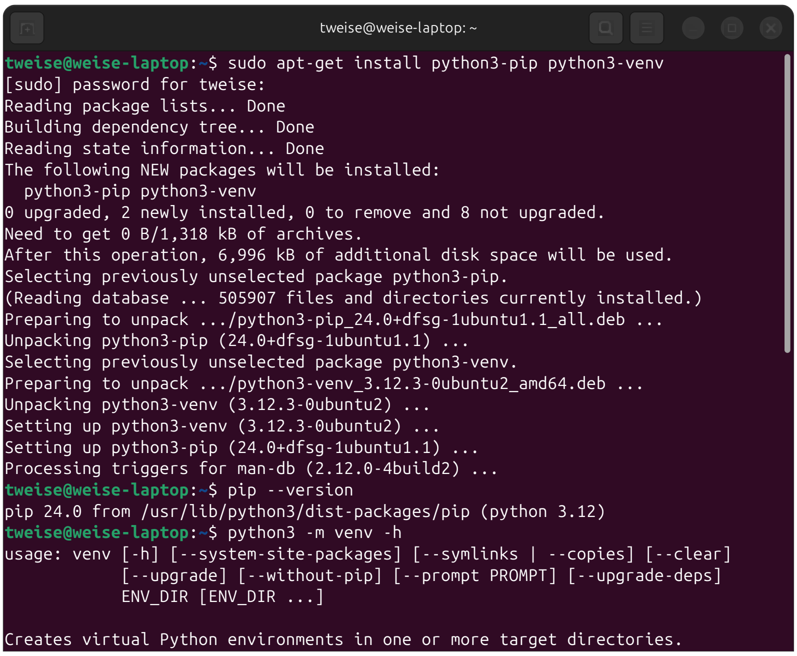


Figure 14.2: Installing pip and venv under Ubuntu Linux: pip is usually already installed, venv maybe or maybe not. We need to use the `apt-get` route to make sure that both `python3-pip` and `python3-venv` are installed.

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 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.

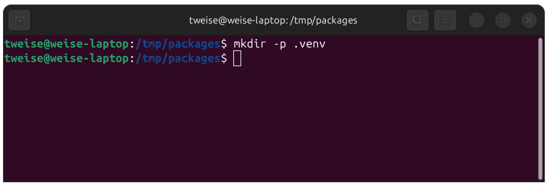
Listing 14.2: An example of using virtual environments and pip under Ubuntu Linux to install NumPy and to run our program Listing 14.1. (stored in file numpy_user_venv.sh; output in Listing 14.3)

```
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
```

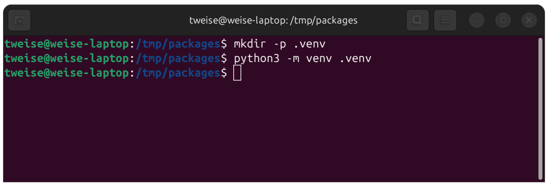
↓ bash numpy_user_venv.sh ↓

Listing 14.3: The stdout of the program numpy_user_venv.sh given in Listing 14.2.

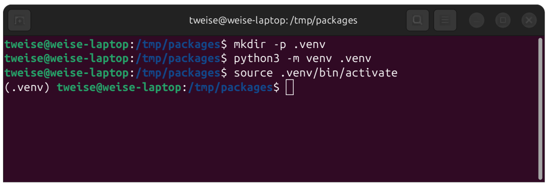
```
# We create the directory '.venv' for the virtual environment.
# We create the (empty) virtual environment inside the directory.
# After creating the virtual environment, we activate it.
# Any Python program now uses the activated virtual environment.
# We install the package 'numpy' into the virtual environment.
Collecting numpy
  Downloading numpy-2.2.4-cp312-cp312-musllinux_1_2_x86_64.whl.metadata (62
    ↪ kB)
Downloading numpy-2.2.4-cp312-cp312-musllinux_1_2_x86_64.whl (17.9 MB)
Installing collected packages: numpy
Successfully installed numpy-2.2.4
# '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.
```



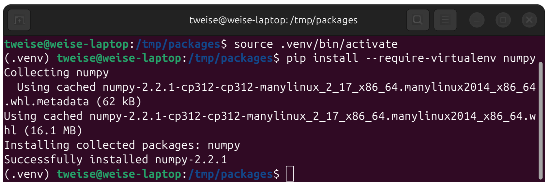
(14.3.1) Create the directory for the virtual environment by typing `mkdir -p .venv` into the terminal and hitting ↲.



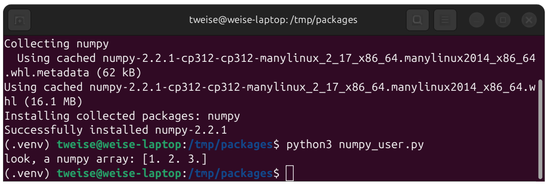
(14.3.2) Set up the virtual environment in directory `.venv` by typing `python3 -m venv .venv` and hitting ↲.



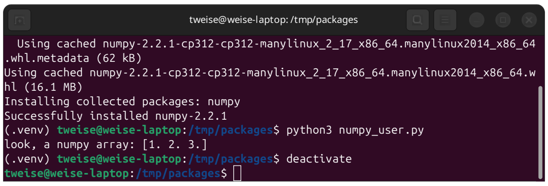
(14.3.3) We activate the virtual environment by `source .venv/bin/activate` + ↲. Notice that the prompt changes: It now has the prefix `(.venv)`.



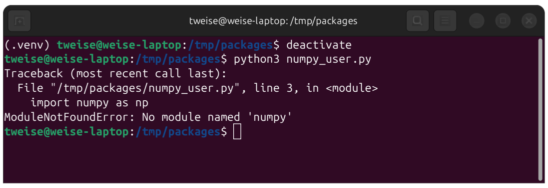
(14.3.4) We install the `NumPy` package into the activated virtual environment vis `pip install --require-virtualenv numpy` + ↲.



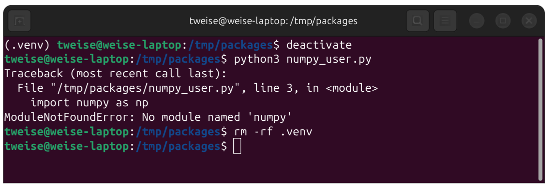
(14.3.5) Now we can execute the `Python` program `numpy_user.py` from Listing 14.1 via `python3 numpy_user.py` + ↲.



(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` command (not illustrated).

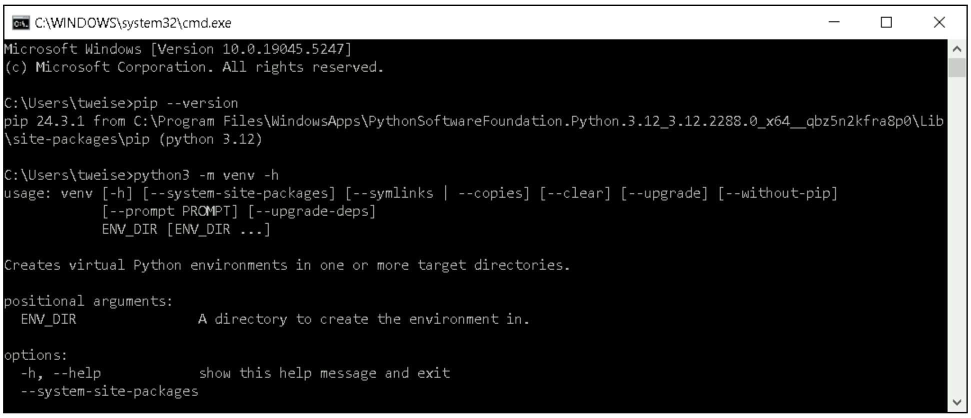


Figure 14.4: Installing pip and venv under Microsoft Windows: They are already installed.

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.

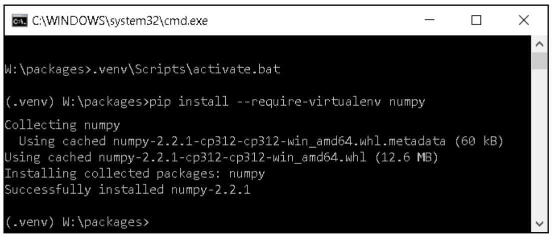
We can now run our program `numpy_user.py` by typing `python numpy_user.py` and hitting ↵. Indeed, as you can see in Figure 14.6.5, the program runs without error and prints `look, a numpy array: [1. 2. 3.]`, exactly as expected. It found the `NumPy` package that was installed in the virtual environment.

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

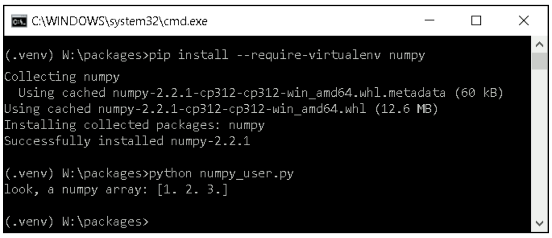
(14.6.1) Create the directory for the virtual environment by typing `md .venv` into the terminal and hitting ↵.

(14.6.2) Set up the virtual environment in directory `.venv` by typing `python -m venv .venv` and hitting ↵.

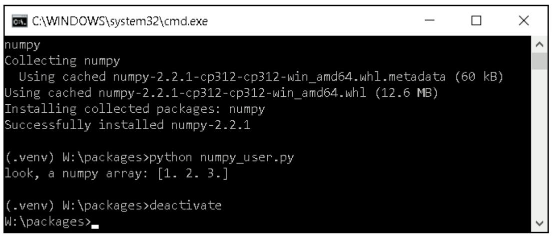
(14.6.3) We activate the virtual environment by `.venv\Scripts\activate.bat` + ↵. Notice that the prompt changes: It now has the prefix `(.venv)`.



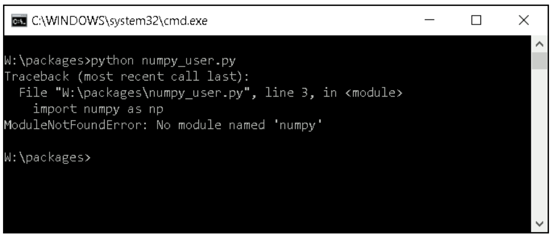
(14.6.4) We install the NumPy package into the activated virtual environment vis `pip install --require-virtualenv numpy` + ↵.



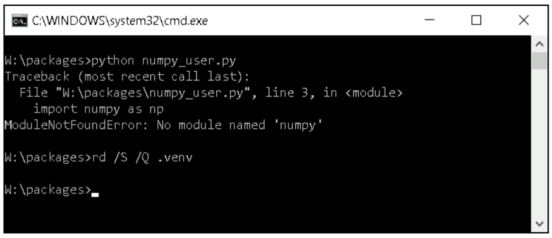
(14.6.5) Now we can execute the Python program `numpy_user.py` from Listing 14.1 via `python numpy_user.py` + ↵.



(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 for 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==1.26.4`. This means that NumPy of exactly the version 1.26.4 is required for our application. Had we written `numpy>=1.26.4`, 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 `1.26.4`, 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 1.26.4." 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 67** 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 `1.26.4` of NumPy be installed. (src)

```
numpy==1.26.4
```

Best Practice 68 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. (stored in file `numpy_user_venv_req.sh`; output in Listing 14.7)

```
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 --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 numpy_user_venv_req.sh` ↓

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.
Collecting numpy==1.26.4 (from -r requirements.txt (line 1))
  Downloading numpy-1.26.4-cp312-cp312-musllinux_1_1_x86_64.whl.metadata
     ↪ (61 kB)
Downloading numpy-1.26.4-cp312-cp312-musllinux_1_1_x86_64.whl (17.8 MB)
Installing collected packages: numpy
Successfully installed numpy-1.26.4
# '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 Virtual Environments in PyCharm

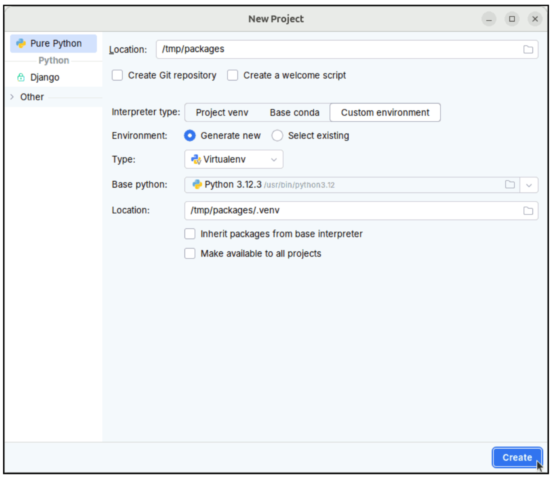
The IDE PyCharm also supports using virtual environments and `requirements.txt` files. Let us here demonstrate this on the example program that requires NumPy which was given as Listing 14.1 and `requirements.txt` file given in Listing 14.5. In Figure 14.7, we work through the steps to create and manage 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.

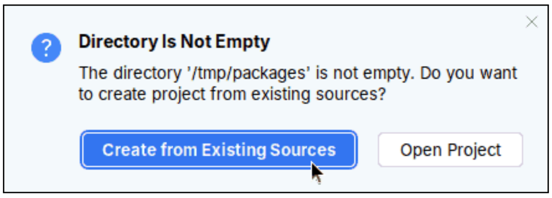
Well, almost: Figure 14.7.2 reminds me that I copied some files into the project folder before creating the project. PyCharm wants to make sure that this is right and asks me so. As answer, I click Create from Existing Sources.

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.

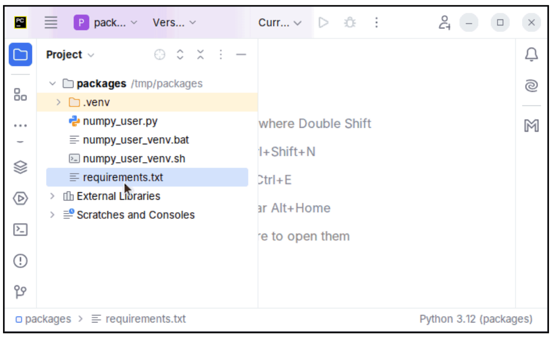
We click on `requirements.txt` and it opens in Figure 14.7.4. Indeed, the file prescribes a single requirement, NumPy in version 1.26.4.



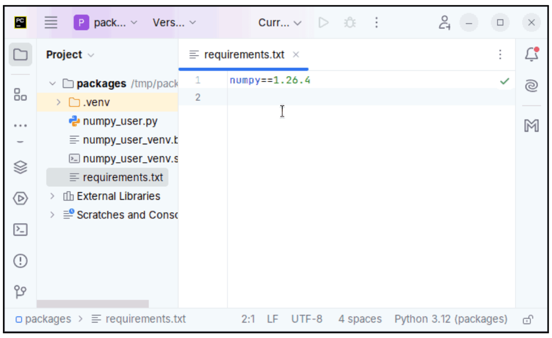
(14.7.1) We create a new project in PyCharm and select a virtual environment as Custom Environment. We click Create.



(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.

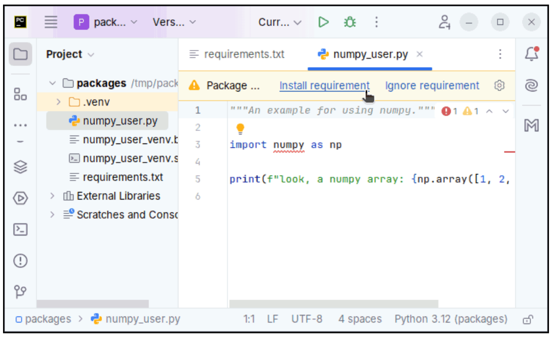


(14.7.3) Let's take a look at `requirements.txt`. We double-click on this file.

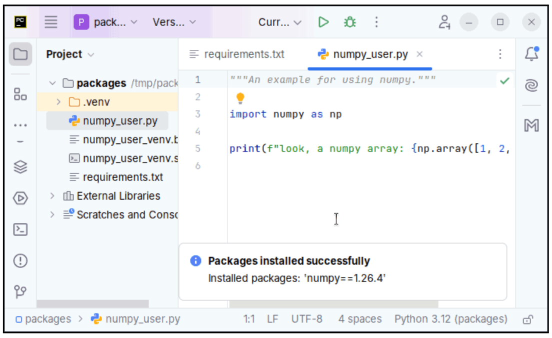


(14.7.4) It contains the one line `numpy==1.26.4`, which means that our project requires NumPy in version 1.26.4.

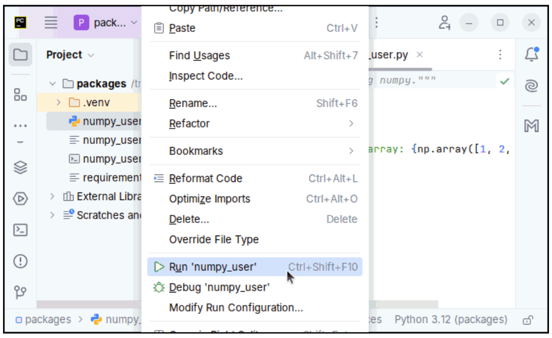
Figure 14.7: Using `requirements.txt` and virtual environments in PyCharm (part 1).



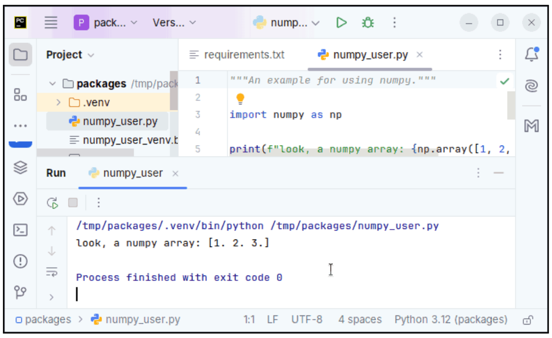
(14.7.5) We now click on `numpy_user.py` and get informed that required packages are missing. We can install them into our project's virtual environment by clicking [Install requirement]. Notice: This question could also have popped up when we opened `requirements.txt`, in which case we would have installed 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 1.26.4, 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 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 69 `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 [259, 291] is maybe the VCS with the most wide-spread use. It is the VCS on which GitHub [213, 235, 270] 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

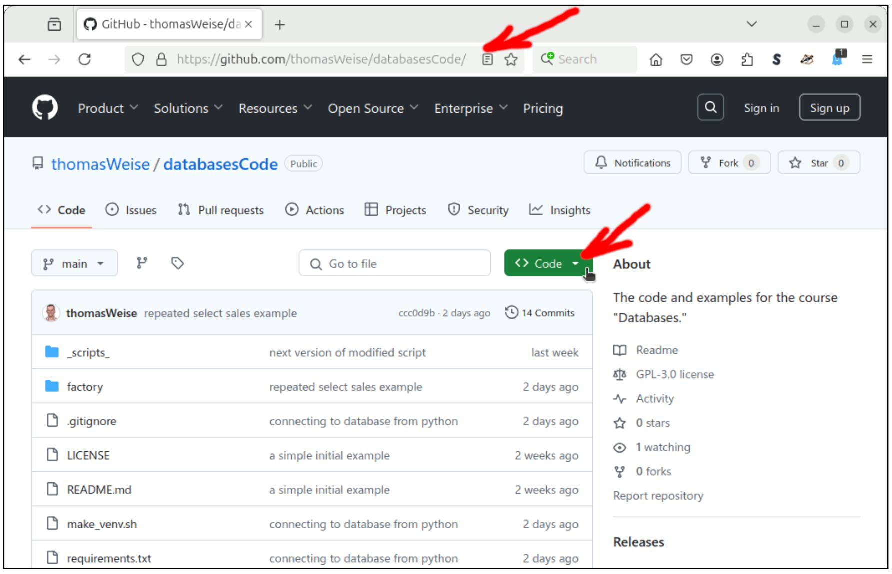
The most fundamental activity you will encounter is *cloning* repositories. Under Git, this basically means to download the complete repository and its history to your machine. You can now make local changes to the downloaded files. You can create commits, that will change the local version of your repository. 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 [58].

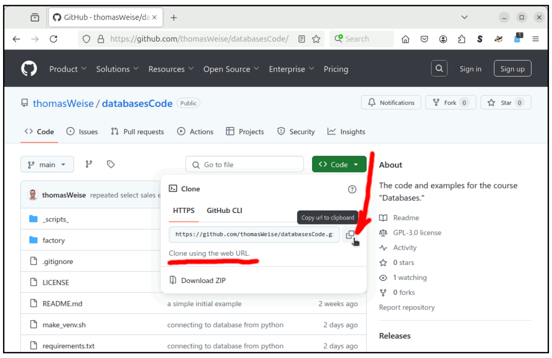
15.1.1 Cloning git Repositories under Pycharm

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 [313] are published like this at `https://github.com/thomasWeise/programmingWithPythonCode`, for example. We also have another book in progress, named *Databases* [312]. It, too, comes with a repository with sources for examples, this time at `https://github.com/thomasWeise/databasesCode`.

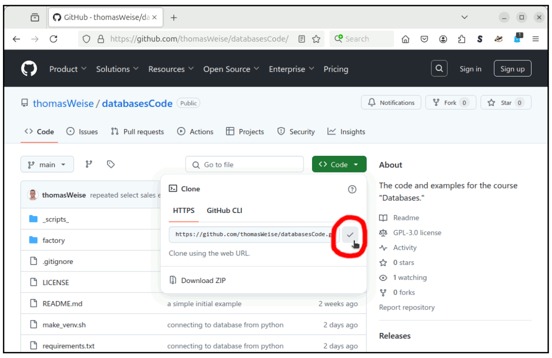
As usual, we work through a topic based on an example. This time, our goal is to *download and get to run code from a GitHub repository in `PyCharm`*. Matter of fact, we already exercised the whole process of cloning the GitHub repository in `PyCharm` with the examples of this book in Section 2.5. To complement the excursion from back then, we this time pick the companion code of our *Databases* book [312] at `https://github.com/thomasWeise/databasesCode` as 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.



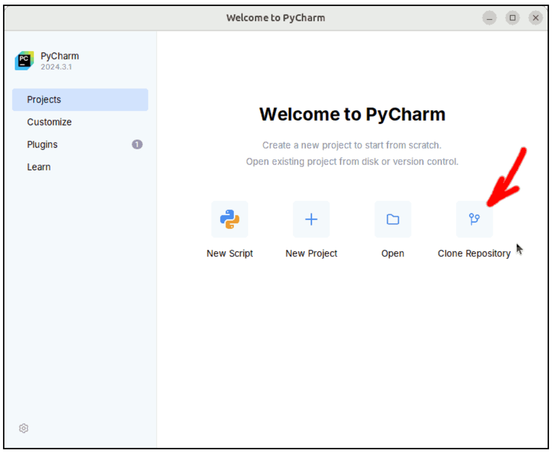
(15.1.1) Maybe we find an interesting repository on GitHub. Let's say it is `https://github.com/thomasWeise/databasesCode`. We click on the Code drop down menu.



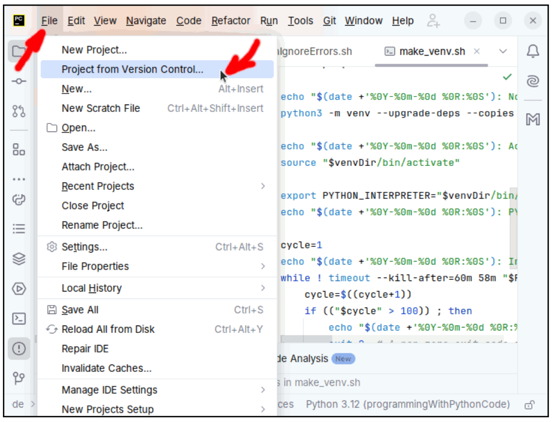
(15.1.2) We click on the button for copying the URL to the clipboard.



(15.1.3) Now the URL is copied to the clipboard and we can paste it wherever we like via Ctrl + V.

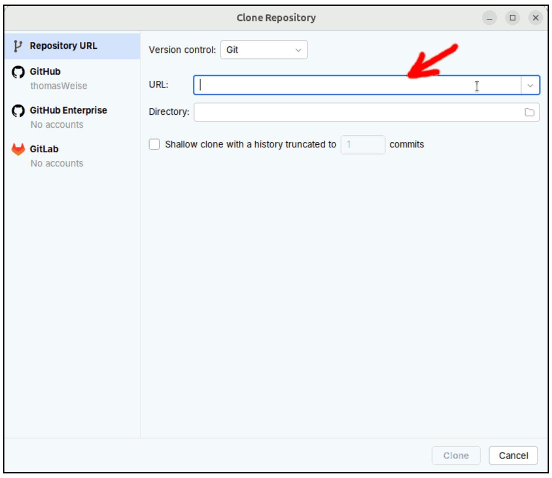


(15.1.4) We open `PyCharm`. If we get to the *Welcome to* `PyCharm` screen, we click Clone Repository.

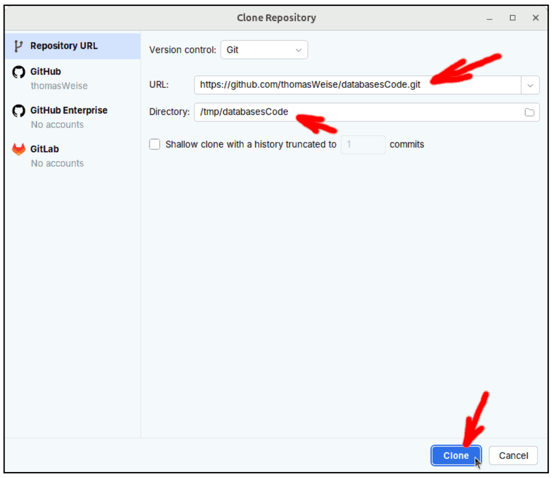


(15.1.5) If we already have a project open in `PyCharm`, we click on ≡ » File » Project from Version Control. . . .

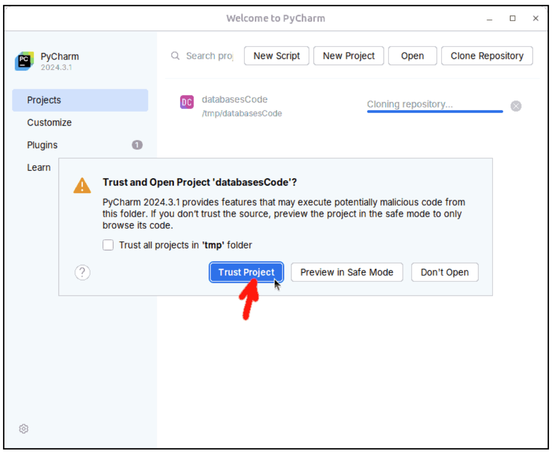
Figure 15.1: Cloning a Git (or GitHub) repository in `PyCharm` and configuring a virtual environment for it.



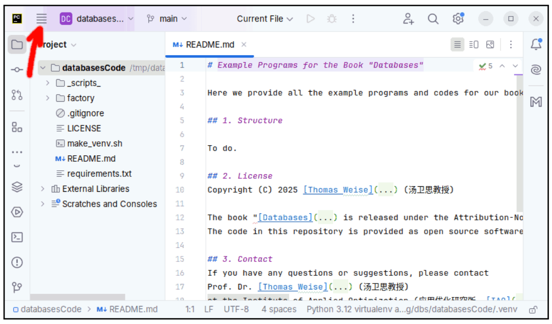
(15.1.6) The "Clone Repository" form appears.



(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.

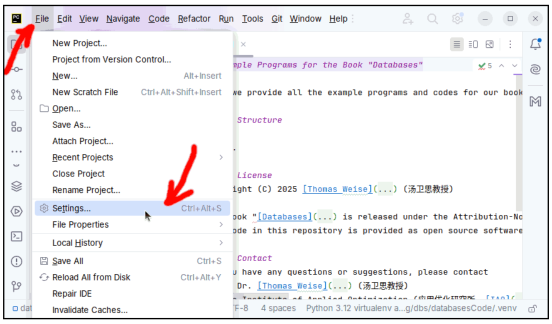


(15.1.9) The repository is downloaded and opens as new project in PyCharm. If this is a Python project, we now can configure its virtual environment settings (see Section 14.3). To do so, we click on ☰.

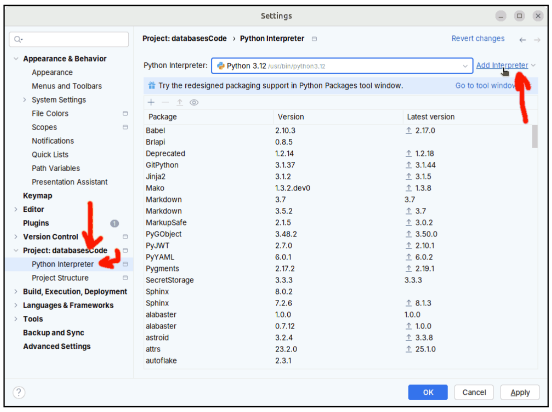
Figure 15.1: Cloning a Git (or GitHub) repository in PyCharm and configuring a virtual environment for it.

Therefore, in Figure 15.1.1, we pretend that you came across this interesting repository on GitHub 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:

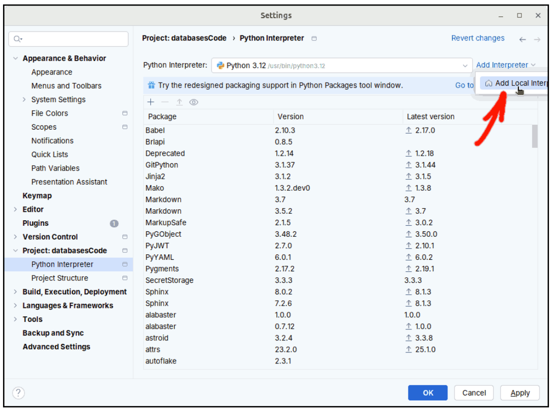
- `https://github.com/user/repository` (or `https://github.com/user/repository.git`) use HTTPS protocol to access the repository `repository` of user `user`. This form is often and commonly used.
- `ssh://git@github.com/user/repository` (or `ssh://git@github.com/user/repository.git`) use the Secure Shell 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 [64].



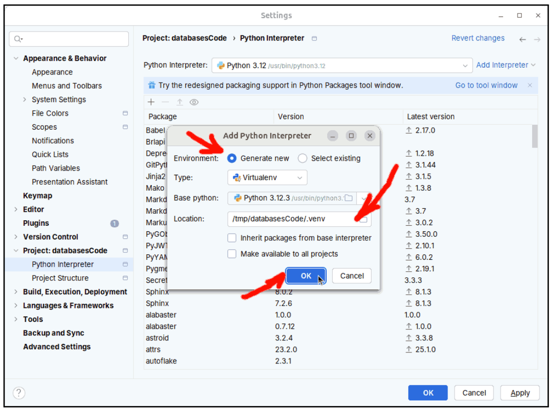
(15.1.10) In the menu that opens, we click on Settings... (we could also press Ctrl + Alt + S).



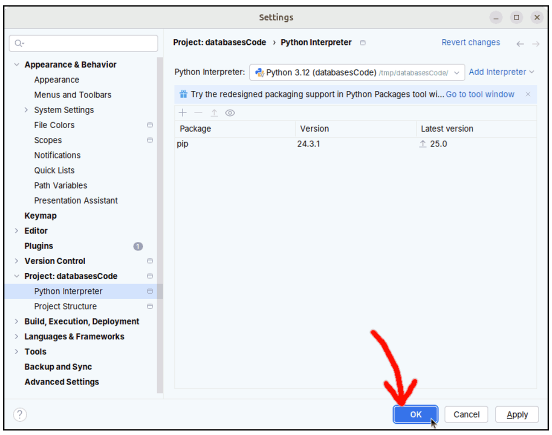
(15.1.11) The Settings menu opens. In the pane on the left-hand side, we click on the item Project: "our project" and then on Python Interpreter. This shows us something like the view on the left hand side: Maybe the system `Python` interpreter is selected or something else. We want to set up a virtual environment for our project, so we click on Add Interpreter.



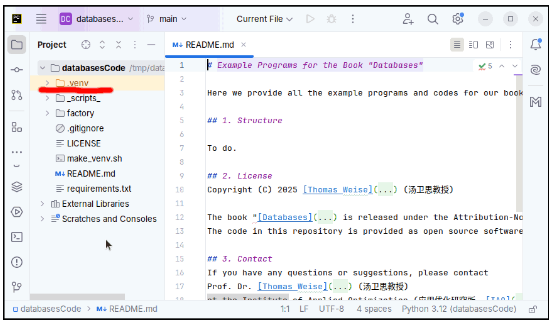
(15.1.12) We then click on Add Local Interpreter.



(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.

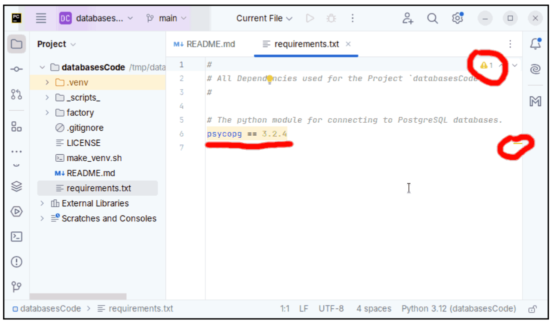


(15.1.14) The new environment is created and we click on OK.

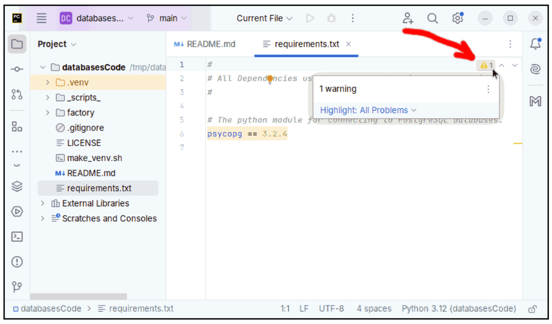


(15.1.15) Indeed, a directory called `.venv` appears in the directory view of our project.

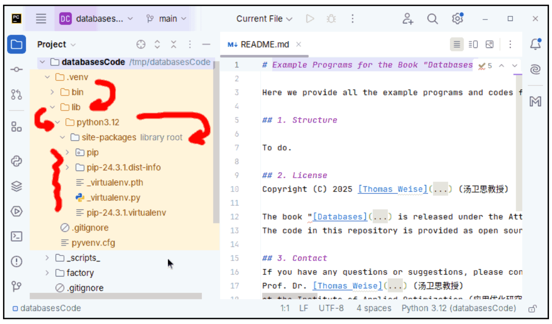
Figure 15.1: Cloning a Git (or GitHub) repository in `PyCharm` and configuring a virtual environment for it.



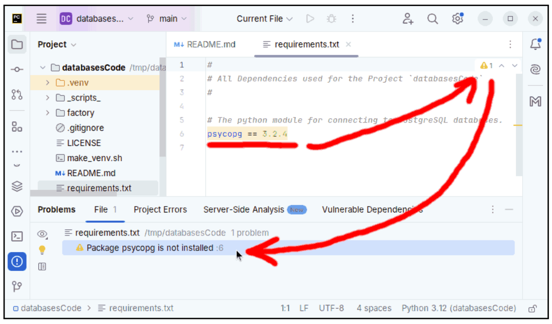
(15.1.16) Many `Python` projects come with a file `requirements.txt` or `requirements-dev.txt`. As discussed in Section 14.2, these list the libraries that the projects depend on. Our example repository also has a file `requirements.txt`, stating that it needs library `psycopg`. This dependency is marked with yellow color, because it is not installed in the virtual environment.



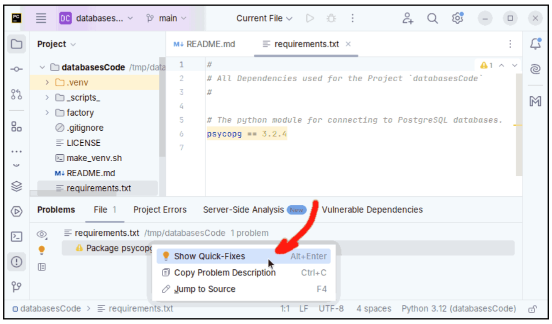
(15.1.17) Clicking on the warnings symbol ⚠ reveals this issue.



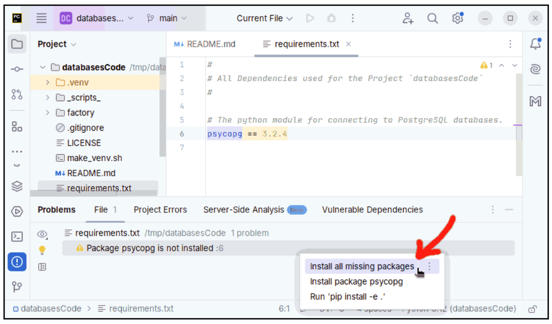
(15.1.18) Indeed: If we look at the `.venv` directory in the directory view, we cannot find the `psycopg` package.



(15.1.19) So we click on the requirements warning…



(15.1.20) …and then on Show Quick-Fixes (or press Alt+Enter).

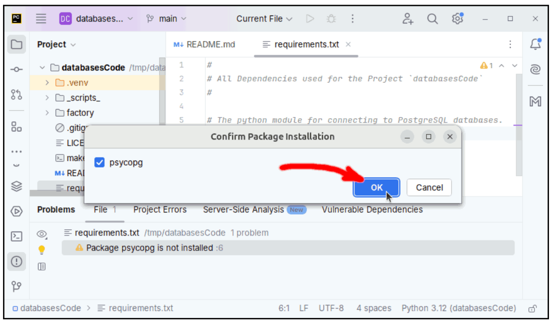


(15.1.21) In the menu that opens up, we select Install all missing packages.

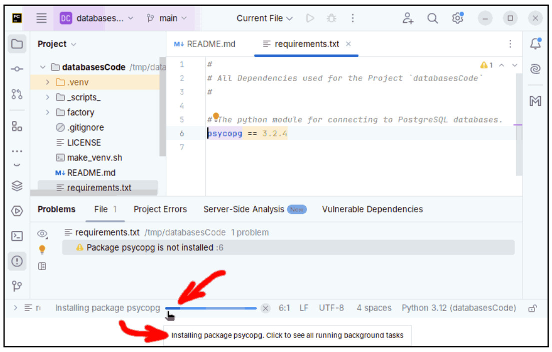
Figure 15.1: Cloning a Git (or GitHub) repository in `PyCharm` and configuring a virtual environment for it.

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`.

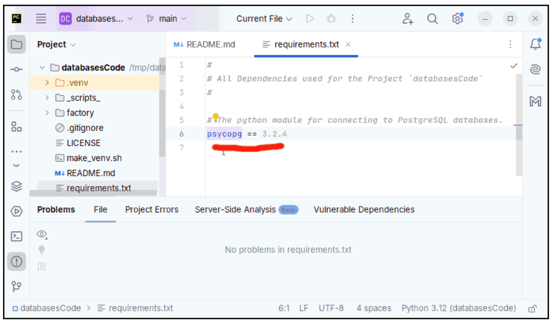
It is important to understand, however, that creating projects by cloning Git repositories is by no means restricted to GitHub. As stated before, Git is a client-server application. You could work in an enterprise that runs its own Git server. You could work with other Git-based repository hosts like 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



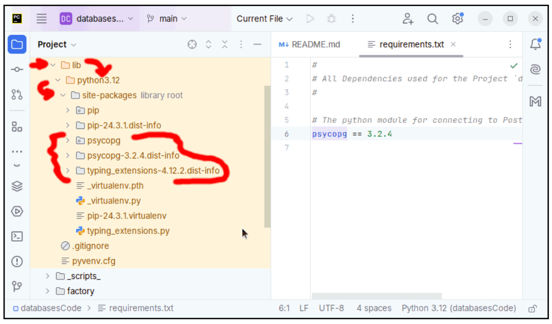
(15.1.22) We get presented a list of packages that will be installed. We click OK.



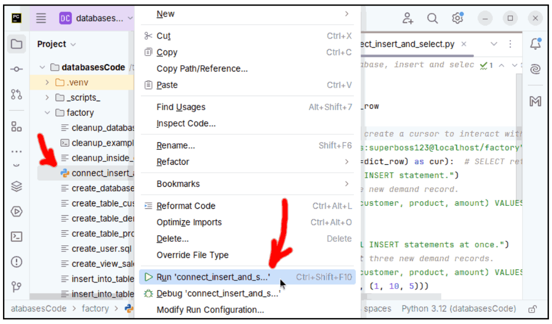
(15.1.23) The required package(s) will now be downloaded and installed.



(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.

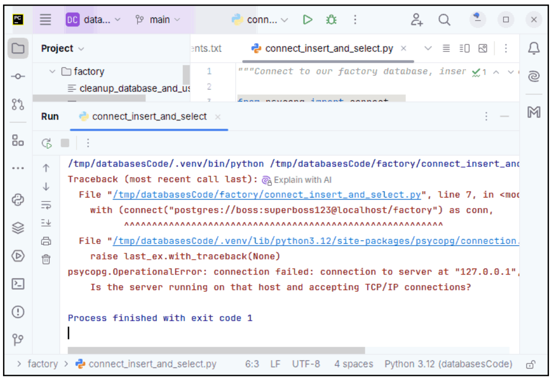
Figure 15.1: Cloning a Git (or GitHub) repository in PyCharm and configuring a virtual environment for it.

different.

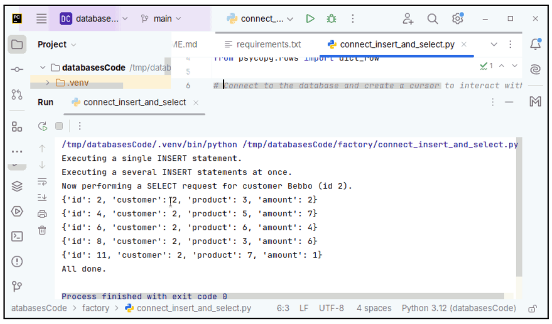
In Figure 15.1.2, we click on the button for copying the URL to the clipboard. Now the URL is copied to the clipboard (Figure 15.1.3) and we can paste it wherever we like via Ctrl+V. But where shall we paste it?

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* [312], and it needs the PostgreSQL database management system (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* [312] 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+Enter) 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 [312]. 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* [312] 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.

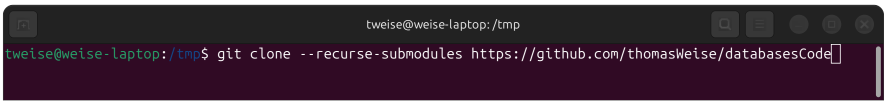
15.1.2 Cloning git Repositories with the git Client

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`.

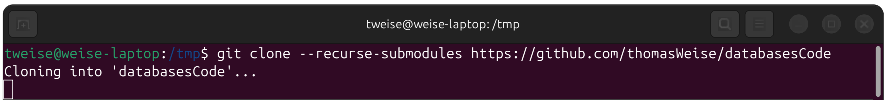
In Figure 15.2, we illustrate how the Git client program is used under Ubuntu Linux. Under Microsoft Windows, it will work analogously. First, we need to open a terminal window. Under Ubuntu Linux, we therefore press Ctrl+Alt+T. Under Microsoft Windows, we instead press ⊞+R, type in `cmd`, and hit ↲. 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 Enter. 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).

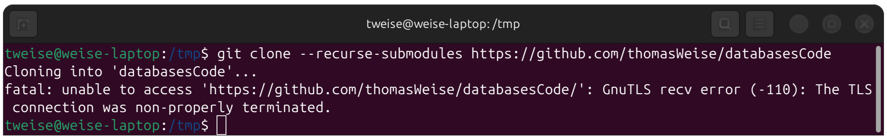
After we hit Enter, the process of cloning the repository begins in Figure 15.2.2. Sadly, when we clone repositories using HTTPS URLs from GitHub, sometimes, there may be connectivity issues. I noticed that using the SSH URLs is much more reliable. An example for such an error is shown in Figure 15.2.3.



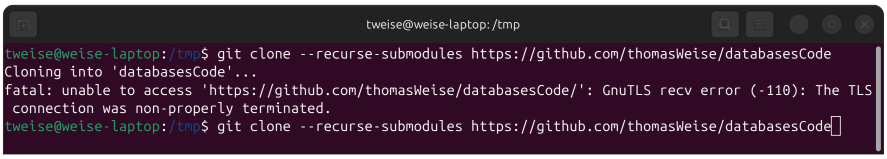
(15.2.1) After opening the `Ubuntu Linux` terminal using Ctrl + Alt + T, we enter the directory into which we want to download the repository. We then type the command `git clone https://github.com/thomasWeise/databasesCode` and hit Enter. The option `--recurse-submodules` can be added just in case.



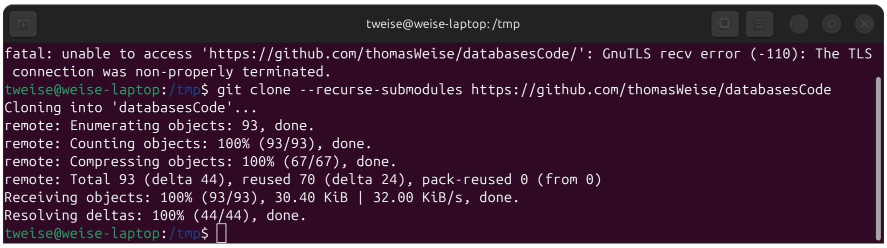
(15.2.2) The process of cloning the repository begins.



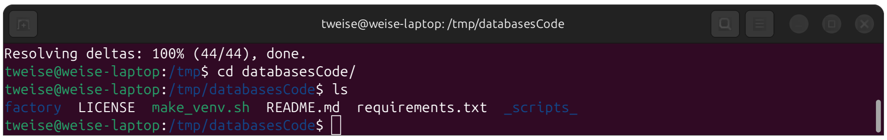
(15.2.3) Sometimes, there may be connectivity issues. But do not fret if that happens. . .



(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.

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 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 `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.

Best Practice 3: Always assume that any `float` value is imprecise. Never expect it to be exact [17, 223].

Best Practice 4: If you need to specify large integers or floats, using underscores (`_`) to separate groups of digits can be very helpful [32]. For example, `37_859_378` is much easier to read than `37859378`.

Best Practice 5: When defining a string literal, the double-quotation mark variant (`"..."`) may be preferred over the single-quotation mark variant (`'...'`); see also Best Practice 6.

Best Practice 6: When defining a multi-line string literal, the double-quotation mark variant (`"""..."""`) is usually preferred over the single-quotation mark variant (`'''...'''`) [108, 302].

Best Practice 7: 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 [302].

Best Practice 8: Variable names should be lowercase, with words separated by underscores [302].

Best Practice 9: 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 10: 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 11: Swapping of variable values can best be done with a multi-assignment statement, e.g., `a, b = b, a`.

Best Practice 12: The names we use in program code should clearly reflect our intentions.

Best Practice 13: Every program should pass static type checking with tools such as `Mypy` (see Useful Tool 2). 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 14: Always use type hints.

Best Practice 15: It is important to integrate type hints from the very start at each project. The idea to first write code and later annotate it with type hints is wrong.

Best Practice 16: Each Python file should start with a string describing its purpose [108]. 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 `"""..."""` [108, 302].

Best Practice 17: 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 where possible.

Best Practice 18: When you need to use an indexable sequence of objects, use a list if you intent to modify this sequence. If you do not intent to change the sequence, use a tuple.

Best Practice 19: Only put immutable objects into tuples. Mutable objects inside tuples makes the tuples modifiable as well, while other programmers may assume that they are immutable. This can lead to strange errors down the line.

Best Practice 20: Only put immutable objects into sets.

Best Practice 21: Sets are unordered. Never expect anything about how the objects you put into a set are actually stored there.

Best Practice 22: 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 23: Dictionary keys must be immutable.

Best Practice 24: Blocks are indented by four spaces.

Best Practice 25: Prefer `elif` over nested `else ... if` constructs.

Best Practice 26: When your `if...then...else` statement only assigns values to variables, use the inline variant discussed in Section 6.4, as it is more compact.

Best Practice 27: If we do not care about the value of a variable (or parameter), we should name it `_` [157]. This information is useful for other programmers as well as static code analysis tools.

Best Practice 28: Do not use (un)equality comparisons of `floats` as loop termination criteria, as they may lead to endless loops. There can always be inputs that cause endless oscillations between values or the appearance of `nan` values (see Section 3.3.5).

Best Practice 29: Function names should be lower case, with underscores separating multiple words if need be [302].

Best Practice 30: All parameters and the return value of a function should be annotated with type hints [324]. From my perspective: *A function without type hints is wrong.*

Best Practice 31: The body of a function is indented with four spaces.

Best Practice 32: 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 33: After the function and its body are defined, leave *two* blank lines before writing the next code [302].

Best Practice 34: Package and module names should be short and lowercase. Underscores can be used to improve readability. [302]

Best Practice 35: 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 continuous integration systems from clogging.

Best Practice 36: A function which is not unit tested is *wrong*.

Best Practice 37: A good unit test 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 38: A good unit test 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 39: Default parameter values must always be immutable.

Best Practice 40: Errors should not be ignored and input should be sanitized. Instead, the input of our functions should be checked for validity (where reasonable). `Exceptions` should be raised as early as possible if any unexpected situation occurs.

Best Practice 41: 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 42: 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 9.

Best Practice 43: Only `Exceptions` should be caught by `except` blocks that we can meaningfully handle. 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 44: 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.

Best Practice 45: 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 46: 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 [287]. 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 47: Where ever possible, the docstrings of functions 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 36).

Best Practice 48: Generator expressions shall be preferred over list comprehension if the sequence of items only needs to be processed once. Generator expression 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 49: Class names should follow the CapWords convention (often also called camel case), i.e., look like `MyClass` or `UniversityDepartment` (not `my_class` or `university_department`) [302].

Best Practice 50: At the beginning of a `class`, a docstring is placed which describes what the class is good for. This docstring and include doctests to demonstrate the class usage, or such doctests can be placed in the docstring of the module.

Best Practice 51: Object attributes must only be created inside the initializer `__init__`. An initial value must immediately be assigned to each attribute.

Best Practice 52: Every attribute of an object must be annotated with a type hint and a documentation comment when created in the initializer `__init__` [173]. type hints work as with normal variables.

Best Practice 53: The type hint `Final` marks an attribute as immutable. All attributes that you do not intend to change should be annotated with `Final`.

Best Practice 54: An attribute is documented in the line *above* the attribute initialization by writing a *comment* starting with `#:`, which explains the meaning of the attribute [266]. (Sometimes, the documentation is given as string directly below the attribute definition [109], but we stick to the former method, because it has proper tool support, e.g., by `Sphinx`.)

Best Practice 55: All methods of `classes` must be annotated with docstrings and type hints.

Best Practice 56: 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 57: Names of attributes and attributes that start with a double leading underscore (`__`) are to be treated as *private* methods and attributes. 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).

Best Practice 58: For implementing `__eq__` and `__hash__`, the following rules hold [201]:

- 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.
- Then, 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 59: Constants are module-level variables which must be assign a value upon definition and which must be annotated with the type hint `Final` [274].

Best Practice 60: The names of constants contain only capital letters with underscores separating words. Examples include `MAX_OVERFLOW` and `TOTAL` [302].

Best Practice 61: Constants are documented by writing a comment starting with `#:` immediately above them [266].

Best Practice 62: Every time you declare a variable that you do not intend to change, mark it with the type hint `Final` [274]. On 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 63: Methods that return `self` should be annotated with the type hint `Self` [269]. Static code analysis tools then see that the method always returns an object of the same class as the object itself.

Best Practice 64: If a parameter of a function can take on only some special constant values `X`, say certain integers or strings, the proper type hint is `Literal[X]` [171]. The same holds if your function always returns the same, simple constant(s) `X`.

Best Practice 65: Packages should *always* be installed in virtual environments and never system-wide (maybe with the exception of pip and venv).

Best Practice 66: 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 67: 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 68: 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 69: 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 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 2: Mypy [172] 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 4.6 on page 69. You can then apply Mypy using the command `mypy fileToScan.py`. We use the Bash script given in Listing 16.1 on page 327 to apply Mypy to the example programs in this book.

Useful Tool 3: 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 [182]. It can be installed via `pip install ruff` as shown in Figure 5.1 on page 83. 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 329.

Useful Tool 4: Pylint is a Python linter that analyzes code for style, potential errors, and possible improvements [226]. It can be installed via `pip install pylint` as shown in Figure 7.1 on page 115. 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 330.

Useful Tool 5: pytest is a Python is a Python framework for writing and executing software tests [160]. It can be installed via `pip install pytest pytest-timeout` as shown in Figure 8.1 on page 128. 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 331. See also Useful Tool 7 later on.

Useful Tool 6: `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 [214, 287].

Useful Tool 7: A doctest is a unit test written directly into the docstring of a function or module. We therefore insert small snippets of Python code and their 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 by modules like `doctest` [80] or tools such as pytest [159] (Useful Tool 5). 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 8: 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 [3, 240, 322]. A comprehensive example on how to use the debugger in PyCharm is given in Section 13.4.

Useful Tool 9: 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 [217].

`pip` should always be used in a virtual environment, see Best Practice 65 and 66.

Useful Tool 10: 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., $a! = 1 * 2 * 3 * 4 * \cdots * (a-1) * a$ [50, 84, 180]. See also Equation 8.1 and Listing 8.1

ϕ The golden ratio (or golden section) 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 [45, 89]. The golden ratio is approximately $\phi \approx 1.618\,033\,988\,749\,894\,848\,204\,586\,834$ [260]. 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 [97, 145, 195], which is approximately $\pi \approx 3.141\,592\,653\,589\,793\,238\,462\,643$. In `Python`, it is provided by the `math` module as constant `pi` with value `3.141592653589793`.

$i..j$ 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. For example, 5..9 is equivalent to $\{5, 6, 7, 8, 9\}$

AI Artificial Intelligence, see, e.g., [242]

Android is a common operating system for mobile phones [258].

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 [110].

apt-get is a command used to install Debian `deb` packages under `Ubuntu Linux` [120, 255, 303]. 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`.

Bash is a the 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 [33, 192, 332]. Learn more at `https://www.gnu.org/software/bash`.

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 [129, 136], the bias is 1023. See Section 3.3.1.

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.

C is a programming language, which is very successful in system programming situations [79, 224].

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 for 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 [25, 204, 231, 237, 279]. These requests and responses are usually sent over network connections. A typical example for 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 , offers access to the DB to the different clients. Here, the client can be some terminal software shipping with the , such as `psql`, or the different applications that access the DBs.

CSV *Comma-Separated Values* is a very common and simple text format for exchanging tabular or matrix data [251]. 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 [69].

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* [312].

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 [327].

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 [3, 240, 322]. See also Useful Tool 8.

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 [108]. They must be delimited by `"""..."""` [108, 302].

doctest *doctests* are 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` [80] or tools such as `pytest` [159]. 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., [114].

e is Euler's number [91], the base of the natural logarithm. $e \in \mathbb{R}$ is an irrational and transcendental number [97, 145], which is approximately $e \approx 2.718\,281\,828\,459\,045\,235\,360$. In `Python`, it is provided by the `math` module as constant `e` with value `2.718281828459045`.

exit code When a process terminates, it can return a single integer value (the exit status code) to indicate success or failure [146]. 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 [129, 136], the exponent is 11 bits with a bias of 1023. See Section 3.3.1.

f-string is a special string in `Python`, which delimited by `f"..."` which can contain expressions in curly braces like `f"a{6-1}b"` that are then turned to text via (string) interpolation, which turns the string to `"a5b"`. f-strings are discussed in Section 3.5.2.

Flask is a lightweight `Python` framework that allows developers to quickly and easily build web applications [2, 54, 289]. It is based on the `Python` WSGI standard [229]. Learn more at `https://flask.palletsprojects.com`.

GIGO Garbage In–Garbage Out, see, e.g., [215]

Git is a distributed Version Control Systems (VCS) which allows multiple users to work on the same code while preserving the history of the code changes [259, 291]. 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 [213, 291]. Learn more at https://github.com.

HTML The Hyper Text Markup Language (HTML) is the text format used by the WWW [22, 126, 290].

HTTP The Hyper Text Transfer Protocol (HTTP) is the protocol linking web browsers to web servers in the WWW [22, 23, 95, 96, 113].

HTTPS The Hypertext Transfer Protocol Secure (HTTPS) is the encrypted variant of Hyper Text Transfer Protocol (HTTP) where data is sent over Transport Layer Security (TLS) [96, 271].

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`.

iOS is the operating system that powers Apple iPhones [49, 262]. Learn more at https://www.apple.com/ios.

iPadOS is the operating system that powers Apple iPads [49]. Learn more at https://www.apple.com/ipados.

IT information technology

Java is another very successful programming language, with roots in the `C` family of languages [27, 178].

JavaScript JavaScript is the predominant programming language used in websites to develop interactive contents for display in browsers [86].

JSON *JavaScript Object Notation* is a data interchange format [34, 282] based on JavaScript [86] syntax.

LAMP Stack A system setup for web applications: `Linux`, Apache (a webserver), `MySQL`, and the server-side scripting language PHP [44, 122].

LibreOffice is on open source office suite [104, 176, 247] which is a good and free alternative to Microsoft Office. It offers software such as LibreOffice `Calc` and LibreOffice `Base`. See [312] for more information and installation instructions.

LibreOffice Base is a DBMSes that can work on stand-alone files but also connect to other popular relational databases [92, 247]. It is part of LibreOffice [104, 176, 247] and has functionality that is comparable to Microsoft Access [20, 55, 293].

LibreOffice Calc is 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 [176, 247], i.e., an alternative to Microsoft `Excel`. It is part of LibreOffice [104, 176, 247].

linter A linter is a tool for analyzing static program code to identify bugs, problems, vulnerabilities, and inconsistent code styles [144, 243]. `Ruff` is an example for a linter used in the `Python` world.

Linux is the leading open source operating system, i.e., a free alternative for Microsoft `Windows` [14, 120, 255, 288, 303]. 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.

macOS or Mac OS is the operating system that powers Apple Mac(intosh) computers [262]. 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 [10, 11, 16, 85, 232]. See https://mariadb.org for more information.

Matplotlib is a Python package for plotting diagrams and charts [133, 135, 143, 205]. Learn more at at https://matplotlib.org [135].

Microsoft Access is a DBMSes that can work on DBs stored in single, stand-alone files but also connect to other popular relational databases [20, 55, 293]. 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 [28, 112, 165]. It is part of Microsoft Office. A free alternative to this commercial software is LibreOffice Calc [176, 247].

Microsoft Office is a commercial suite of office software, including Microsoft Excel, Microsoft Word, and Microsoft Access [165]. LibreOffice is a free and open source alternative.

Microsoft Windows is a commercial proprietary operating system [31]. 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 [81, 188] and part of Microsoft Office. A free alternative to this commercial software is the LibreOffice Writer.

ML Machine Learning, see, e.g., [253]

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 [314]. Learn more at https://thomasweise.github.io/moptipy.

Mypy is a static type checking tool for Python [172] 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 [30, 85, 234, 277, 321]. MySQL is famous for its use in the LAMP Stack. See https://www.mysql.com for more information.

$\mathbb{N}_1$ the set of the natural numbers *excluding* 0, i.e., 1, 2, 3, 4, and so on. It holds that $\mathbb{N}_1 \subset \mathbb{Z}$.

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 datastructures [75, 119, 143]. Learn more at https://numpy.org [197].

$\mathcal{O}(g(x))$ If $f(x) = \mathcal{O}(g(x))$, then there exist positive numbers $x_0 \in \mathbb{R}^+$ and $c \in \mathbb{R}^+$ such that $f(x) \leq c * g(x) \forall x \geq x_0$ [13, 167]. In other words, $\mathcal{O}(g(x))$ describes an upper bound for function growth.

OS Operating System, the system that runs your computer, see, e.g., Linux, Microsoft Windows, macOS, and Android.

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 [18, 175]. Learn more at https://pandas.pydata.org [208].

pip is the standard tool to install Python software packages from the PyPI repository [139, 216]. To install a package `thepackage` hosted on PyPI, type `pip install thepackage` into the terminal. Learn more at https://packaging.python.org/installing.

PostgreSQL An open source object-relational DBMS [94, 200, 220, 277]. 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 [174]. Learn more at https://www.psycopg.org [304].

PyCharm is the convenient Python IDE that we recommend for this course [297, 322, 326]. It comes in a free community 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 [226]. Learn more at https://www.pylint.org and Useful Tool 4 and in [313]

PyPI The Python Package Index (PyPI) is an online repository that provides the software packages that you can install with pip [29, 283, 296]. Learn more at https://pypi.org.

pytest is a framework for writing and executing unit tests in Python [77, 160, 203, 206, 322]. Learn more at https://pytest.org and in Section 8.3 and in [313]

Python The Python programming language [132, 170, 313], i.e., what you will learn about in our book [313]. Learn more at https://python.org.

PyTorch is a Python library for deep learning and AI [209, 230]. 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 $\mathbb{Q} \subset \mathbb{R}$.

$\mathbb{R}$ the set of the real numbers.

$\mathbb{R}^+$ the set of the positive real numbers, i.e., $\mathbb{R}^+ = \{x \in \mathbb{R} : x > 0\}$.

regex A *Regular Expression*, often called "regex" for short, is a sequence of characters that define a search pattern for text strings [137, 161, 189, 193]. In Python, the `re` module offers functionality work with regular expressions [161, 233]. In PostgreSQL, regex-based pattern matching is supported as well [218].

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 [59, 116, 118, 264, 275, 312, 319].

Ruff is a linter and code formatting tool for Python [182]. Learn more at https://docs.astral.sh/ruff or in Useful Tool 3.

Scikit-learn is a Python library offering various machine learning tools [212, 230]. Learn more at https://scikit-learn.org.

SciPy is a Python library for scientific computing [143, 307]. Learn more at https://scipy.org.

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 for servers are web servers [44] 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" [163].

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 [129, 136]. 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 [184]. In `Python`, the function `signature` of the module `inspect` provides some information about the signature of a function [43].

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 [129, 136], the exponent is 52 bits. See Section 3.3.1.

SimPy is a `Python` library for discrete event simulation [334]. Learn more at `https://simpy.readthedocs.io`.

Sphinx `Sphinx` is a tool for generating software documentation [292]. 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 [51]. It is understood many DBMSes. You find the SQL commands supported by `PostgreSQL` in the reference [267].

SQLi attack A SQL injection attack is a web 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 [68, 162, 210, 312]. 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 [268]. Some programming languages, like `Python`, offer built-in datatypes (such as `LiteralString` [268]) to annotate string constants that can be used by static type-checkers. At the time of this writing, `Mypy` does not support this yet [305, 333].

SQLite is an relational DBMS which runs as in-process library that works directly on files as opposed to the client-server model 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 [51, 103, 323]. Learn more at `https://sqlite.org`.

SSH Secure Shell Transport Layer Protocol, a protocol that provides users a secure way to access computers over an unsecure network [15, 48, 331].

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 [154]. 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`) [148]. 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`) [148]. 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`) [148]. 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 for string interpolation is turning `f"Rounded {1.234:.2f}"` to `"Rounded 1.23"`. This is discussed in Section 3.5.2.

sudo In order to perform administrative tasks such as installing new software under `Linux`, root (or "super") user privileges as needed [57]. 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 (SVG) format is an XML-based format for vector graphics [71]. 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, 166]. Learn more at `https://www.tensorflow.org`.

terminal A terminal is a text-based window where you can enter commands and execute them [14, 57]. 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 [83, 236, 271], used by, e.g., HTTPS.

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 [168, 301]. `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` [57, 122]. 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 [138, 284, 294]. 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.

URI A *Uniform Resource Identifier* is an identifier for an abstract or physical resource in the internet [330]. It can be a URL, a name, or both. URIs are supersets of URLs. The connection strings of the `PostgreSQL` DBMS are examples for 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 [24]. URLs are subsets of Uniform Resource Identifiers (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 [138, 330]. 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 3.5.6).

VCS A *Version Control System* is a software which allows you to manage and preserve the historical development of your program code [291]. 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 [306]. Learn more at `https://docs.python.org/3/library/venv.html` [306].

virtual environment A virtual environment is a directory that contains a local `Python` installation [187, 308]. 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.

WWW World Wide Web [22, 74]

x**-axis** The x-axis is the horizontal axis of a two-dimensional coordinate system, often referred to abscissa.

XML The *Extensible Markup Language* is a text-based language for storing and transporting of data [35, 62, 156]. 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 [121], as it protects against several security issues.

YAML *YAML Ain't Markup Language™* is a human-friendly data serialization language for all programming languages [61, 82, 156]. 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

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, 297).

[2] Olatunde Adedeji. *Full-Stack Flask and React*. Birmingham, England, UK: Packt Publishing Ltd, Nov. 2023. ISBN: 978-1-80324-844-8 (cit. on pp. 3, 292).

[3] David J. Agans. *Debugging*. New York, NY, USA: AMACOM, Sept. 2002. ISBN: 978-0-8144-2678-4 (cit. on pp. 234, 289, 292).

[4] 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. 73).

[5] *American Standard Code for Information Interchange (ASCII 1963)*. Tech. rep. ASA X3.4-1963. New York, NY, USA: American Standards Association Incorporated, 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. 50).

[6] AndrewBadr. *TimeComplexity*. 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. 89, 93, 217).

[7] 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. 246).

[8] "Annotating `Callable` Objects". In: *Python 3 Documentation*. Vol. The Python Standard Library. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2024. URL: `https://docs.python.org/3/library/typing.html#annotating-callables` (visited on 2024-10-09) (cit. on p. 137).

[9] 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. 1).

[10] 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 [11] (cit. on pp. 294, 304).

[11] 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 [10] (cit. on pp. 294, 304).

[12] Ivo Babuška. "Numerical Stability in Mathematical Analysis". In: *World Congress on Information Processing (IFIP'1968)*. Aug. 5, 1966–Aug. 10, 1968, Edinburgh, Scotland, UK. Ed. by A.J.H. Morrell. Vol. 1: Mathematics, Software. 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. 196, 197, 199, 204, 314).

[13] Paul Gustav Heinrich Bachmann. *Die Analytische Zahlentheorie / Dargestellt von Paul Bachmann*. Vol. Zweiter Theil of Zahlentheorie: Versuch einer Gesamtdarstellung dieser Wissenschaft in ihren Hauptheilen. 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. 294).

[14] 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. 7, 293, 297).

[15] 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. 296).

[16] 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. 294).

[17] 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. 32, 284).

[18] 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, 294).

[19] 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 2024-07-06) (cit. on p. 32).

[20] Ben Beitler. *Hands-On Microsoft Access 2019*. Birmingham, England, UK: Packt Publishing Ltd, Mar. 2020. ISBN: 978-1-83898-747-3 (cit. on pp. 293, 294).

[21] Jon Bentley. *Programming Pearls*. 2nd ed. Reading, MA, USA: Addison-Wesley Professional, Oct. 7, 1999. ISBN: 978-0-201-65788-3 (cit. on pp. 118, 119).

[22] 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. 293, 298).

[23] 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. 293).

[24] 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. 297).

[25] 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. 291).

[26] 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).

[27] Joshua Bloch. *Effective Java*. Reading, MA, USA: Addison-Wesley Professional, May 2008. ISBN: 978-0-321-35668-0 (cit. on pp. 195, 293).

[28] 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. 294).

[29] 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. 295).

[30] 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. 294).

[31] 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. 7, 294).

[32] 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. 35, 284).

[33] Ron Brash and Ganesh Naik. *Bash Cookbook*. Birmingham, England, UK: Packt Publishing Ltd, July 2018. ISBN: 978-1-78862-936-2 (cit. on pp. 291, 328).

[34] 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. 246, 293).

[35] 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. 246–248, 298).

[36] 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, Nov. 1999, see https://maths-people.anu.edu.au/~brent/pd/rpb183tr.pdf (cit. on pp. 123, 125).

[37] Florian Bruhin. *Python f-Strings*. Winterthur, Switzerland: Bruhin Software, May 31, 2023. URL: https://fstring.help (visited on 2024-07-25) (cit. on p. 45).

[38] 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. 187).

[39] "Built-in Constants". In: *Python 3 Documentation*. Vol. The Python Standard Library. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2024. URL: https://docs.python.org/3/library/constants.html (visited on 2024-12-09) (cit. on p. 214).

[40] "Built-in `Exceptions`". In: *Python 3 Documentation*. Vol. The Python Standard Library. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2024. URL: https://docs.python.org/3/library/exceptions.html (visited on 2024-10-29) (cit. on pp. 161, 162, 217).

[41] "Built-in Functions". In: *Python 3 Documentation*. Vol. The Python Standard Library. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2024. URL: https://docs.python.org/3/library/functions.html (visited on 2024-12-09) (cit. on p. 218).

[42] "Built-in Types". In: *Python 3 Documentation*. Vol. The Python Standard Library. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2024. URL: https://docs.python.org/3/library/stdtypes.html (visited on 2024-08-22) (cit. on p. 78).

[43] 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. 296).

[44] Jason Cannon. *High Availability for the LAMP Stack*. Shelter Island, NY, USA: Manning Publications, June 2022 (cit. on pp. 293, 295).

[45] 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. 243, 291).

[46] 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 pp. 2, 3).

[47] 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. 73).

[48] 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. 296).

[49] 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. 293).

[50] 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. 122, 291).

[51] 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. 133, 296).

[52] Eric Chapman. *GitHub Actions*. Birmingham, England, UK: Packt Publishing Ltd, Mar. 2024. ISBN: 978-1-80512-862-5 (cit. on p. 170).

[53] *Chinese Internal Code Specification, National Standard Extended (GBK)*. Technical Supervision Standard Letter 1995 229. China, Beijing (中国北京市): Standardization Department of the State Administration of Technical Supervision, Department of Science, and Quality Supervision of the Ministry of Electronics Industry, Dec. 15, 1995 (cit. on p. 50).

[54] 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. 292).

[55] 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. 293, 294).

[56] "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 p. 35).

[57] 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. 7, 297).

[58] "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. 273).

[59] 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. 295).

[60] *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): National Standards Administration, 1980–May 1, 1981. URL: https://openstd.samr.gov.cn/bzgk/gb/newGbInfo?hcno=5664A728BD9D523DE3B99BC37AC7A2CC (visited on 2024-07-26) (cit. on p. 50).

[61] 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. 246, 298).

[62] 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. 246, 298).

[63] "`collections.abc` – Abstract Base Classes for Containers". In: *Python 3 Documentation*. Vol. The Python Standard Library. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2024. URL: `https://docs.python.org/3/library/collections.abc.html` (visited on 2024-08-22) (cit. on p. 78).

[64] *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. 275).

[65] "`contextlib` – Utilities for `with`-Statement Contexts". In: *Python 3 Documentation*. Vol. The Python Standard Library. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2024. URL: `https://docs.python.org/3/library/contextlib.html` (visited on 2024-11-01) (cit. on pp. 157, 158).

[66] 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. 89, 93, 217).

[67] 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. 110, 172, 183, 184).

[68] 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. 296).

[69] "`csv` – CSV File Reading and Writing". In: *Python 3 Documentation*. Vol. The Python Standard Library. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2024. URL: `https://docs.python.org/3/library/csv.html` (visited on 2024-11-14) (cit. on p. 292).

[70] 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 p. 58).

[71] 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. 205, 246, 297).

[72] "Data Model". In: *Python 3 Documentation*. Vol. The Python Language Reference. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2024. Chap. 3. URL: `https://docs.python.org/3/reference/datamodel.html` (visited on 2024-08-22) (cit. on p. 78).

[73] 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/15762270n10/15762270n10p21.pdf` (visited on 2024-08-10) (cit. on p. 58).

[74] 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. 298).

[75] 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, 259, 294).

[76] *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).

[77] 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. 128, 295).

[78] "Dictionaries". In: *Python 3 Documentation*. Vol. The Python Language Reference. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2024. Chap. 3.2.7.1. URL: https://docs.python.org/3/reference/datamodel.html#dictionaries (visited on 2024-08-27) (cit. on p. 94).

[79] 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. 291).

[80] "Doctest – Test Interactive Python Examples". In: *Python 3 Documentation*. Vol. The Python Standard Library. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2024. URL: https://docs.python.org/3/library/doctest.html (visited on 2024-11-07) (cit. on pp. 171, 289, 292).

[81] 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. 190, 246, 294).

[82] 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. 246, 298).

[83] 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. 297).

[84] Jacques Dutka. "The Early History of the Factorial Function". *Archive for History of Exact Sciences* 43(3):225–249, Sept. 1991. Heidelberg, Baden-Württemberg, Germany: Springer-Verlag GmbH Germany. ISSN: 0003-9519. doi:10.1007/BF00389433. Communicated by Umberto Bottazzini (cit. on pp. 122, 291).

[85] 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 p. 294).

[86] *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. 293).

[87] 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. 137, 140).

[88] Euclid of Alexandria (Εύκλείδης). "Elementary Number Theory". In: *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. Chap. Book 7. ISBN: 978-0-615-17984-1. URL: https://farside.ph.utexas.edu/Books/Euclid/Elements.pdf (visited on 2024-09-30) (cit. on p. 123).

[89] Euclid of Alexandria (Εύκλείδης). *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. 243, 291).

[90] 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. ISSN: 1432-4350. doi:10.1007/BF01699475. URL: https://www.researchgate.net/publication/301720080 (visited on 2024-09-24). Translation of [91]. (Cit. on p. 310).

[91] 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 [90] for a translation. (Cit. on pp. 292, 310).

[92] 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. 293).

[93] 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. 243).

[94] 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. 295).

[95] 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. 293).

[96] 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. 293).

[97] 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. 30, 291, 292).

[98] "Floating-Point Arithmetic: Issues and Limitations". In: *Python 3 Documentation*. Vol. The Python Tutorial. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2024. Chap. 15. URL: https://docs.python.org/3/tutorial/floatingpoint.html (visited on 2024-12-08) (cit. on pp. 31, 32, 196).

[99] "Formatted String Literals". In: *Python 3 Documentation*. Vol. The Python Tutorial. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2024. Chap. 7.1.1. URL: https://docs.python.org/3/tutorial/inputoutput.html#formatted-string-literals (visited on 2024-07-25) (cit. on p. 45).

[100] 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. 116).

[101] "`fractions` – Rational Numbers". In: *Python 3 Documentation*. Vol. The Python Standard Library. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2024. URL: https://docs.python.org/3/library/fractions.html (visited on 2024-12-12) (cit. on p. 222).

[102] 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, Japan: Department of Information Engineering, Hiroshima University. ISSN: 2185-2839. URL: http://www.ijnc.org (visited on 2024-09-28) (cit. on p. 123).

[103] 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)*, Sept. 5-9, 2022, hybrid/Sydney, NSW, Australia (cit. on pp. 133, 296).

[104] 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. IFIP Advances in Information and Communication Technology (IFIPAICT). Heidelberg, Baden-Württemberg, 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. 190, 246, 293).

[105] 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. 141).

[106] 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).

[107] 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 p. 217).

[108] 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. 50, 84, 284, 285, 292).

[109] 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. 194, 287).

[110] 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. 291).

[111] 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. 3).

[112] 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. 294).

[113] Ilya Grigorik. *HTTP Protocols*. Sebastopol, CA, USA: O'Reilly Media, Inc., Dec. 2017. ISBN: 978-1-4920-3046-1 (cit. on p. 293).

[114] 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, 292).

[115] Sławomir Gryś. *Computer Arithmetic in Practice*. Boca Raton, FL, USA: CRC Press, Sept. 2023. ISBN: 978-1-000-93492-2 (cit. on p. 39).

[116] 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. 295).

[117] 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. 118, 119).

[118] 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. 295).

[119] 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, 259, 294).

[120] Michael Hausenblas. *Learning Modern Linux*. Sebastopol, CA, USA: O'Reilly Media, Inc., Apr. 2022. ISBN: 978-1-0981-0894-6 (cit. on pp. 7, 291, 293).

[121] Christian Heimes. `defusedxml` *0.7.1: XML Bomb Protection for Python stdlib Modules*. Mar. 8, 2021. URL: https://pypi.org/project/defusedxml (visited on 2024-12-15) (cit. on p. 298).

[122] 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. 7, 293, 297).

[123] 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. 198).

[124] 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. 174).

[125] Raymond Hettinger. "What's New In Python 3.8: f-String Support `=` for Self-Documenting Expressions and Debugging". In: *Python 3 Documentation*. Vol. What's New in Python. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2024. URL: https://docs.python.org/3/whatsnew/3.8.html#bpo-36817-whatsnew (visited on 2024-12-01) (cit. on p. 47).

[126] 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. 246, 293).

[127] D. Richard Hipp et al. *How SQLite Is Tested*. 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. 133).

[128] D. Richard Hipp et al. *Well-Known Users of 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 p. 133).

[129] 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. 30, 35, 36, 291, 292, 296).

[130] Trey Hunner. *Every Dunder Method in Python*. Reykjavík, Iceland: Python Morsels, Mar. 19, 2024. URL: https://www.pythonmorsels.com/every-dunder-method (visited on 2024-12-18) (cit. on pp. 252, 255).

[131] Trey Hunner. *Python Big $\mathcal{O}$: The Time Complexities of Different Data Structures in Python; Python 3.8-3.12*. Reykjavík, Iceland: Python Morsels, Apr. 16, 2024. URL: https://www.pythonmorsels.com/time-complexities (visited on 2024-08-27) (cit. on pp. 89, 217).

[132] John Hunt. *A Beginners Guide to Python 3 Programming*. 2nd ed. Undergraduate Topics in Computer Science (UTICS). Cham, Switzerland: Springer, 2023. ISBN: 978-3-031-35121-1. doi:10.1007/978-3-031-35122-8 (cit. on pp. 1, 295).

[133] 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, 294).

[134] 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. 73).

[135] 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, 294).

[136] *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. 30, 35, 36, 214, 291, 292, 296).

[137] *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 Sept. 1991 at the url (cit. on p. 295).

[138] *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. 51, 155, 297, 325).

[139] "Installing Python Modules". In: *Python 3 Documentation*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2024. URL: https://docs.python.org/3/installing (visited on 2024-08-17) (cit. on pp. 126, 258, 295).

[140] "`itertools` – Functions Creating `Iterators` for Efficient Looping". In: *Python 3 Documentation*. Vol. The Python Standard Library. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2024. URL: https://docs.python.org/3/library/itertools.html (visited on 2024-11-09) (cit. on pp. 185, 188).

[141] Mike James. *Programmer's Python: Everything is an Object – Something Completely Different*. 2nd ed. I/O Press, June 25, 2022 (cit. on p. 211).

[142] 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, 72).

[143] 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, 259, 294, 295).

[144] 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 p. 293).

[145] 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. 30, 291, 292).

[146] "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. 292).

[147] 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).

[148] "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 pp. 296, 297).

[149] 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. 196, 197, 199, 200, 204, 314).

[150] 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 (cit. on p. 58).

[151] 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. 69).

[152] Andreas Klein. "A Generalized Kahan-Babuška-Summation-Algorithm". *Computing* 76(3-4):279–293, Jan. 2006. Heidelberg, Baden-Württemberg, Germany: Springer-Verlag GmbH Germany. ISSN: 0010-485X. doi:10.1007/s00607-005-0139-x. Based on [12, 149, 191] (cit. on p. 197).

[153] 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. 1).

[154] Donald Ervin Knuth. *Fundamental Algorithms*. 3rd ed. Vol. 1. The Art of Computer Programming. Reading, MA, USA: Addison-Wesley Professional, 1997. ISBN: 978-0-201-89683-1 (cit. on p. 296).

[155] Donald Ervin Knuth. *Sorting and Searching*. Vol. 3. The Art of Computer Programming. Reading, MA, USA: Addison-Wesley Professional, 1998. ISBN: 978-0-201-89685-5 (cit. on pp. 89, 93, 118, 119, 217).

[156] Katie Kodes. *Intro to XML, JSON, & YAML*. London, England, UK: Payhip, 2019–Sept. 4, 2020 (cit. on pp. 246, 298).

[157] 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. 108, 285).

[158] 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. 116, 117).

[159] 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. 171, 289, 292).

[160] 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. 128, 289, 295).

[161] Andrew M. Kuchling. *Regular Expression HOWTO*. Vol. Python HOWTOs. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2024. URL: https://docs.python.org/3/howto/regex.html (visited on 2024-11-01) (cit. on p. 295).

[162] 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. 296).

[163] Jay LaCroix. *Mastering Ubuntu Server*. 4th ed. Birmingham, England, UK: Packt Publishing Ltd, Sept. 2022. ISBN: 978-1-80323-424-3 (cit. on p. 295).

[164] "Lambda". In: *Python 3 Documentation*. Vol. The Python Language Reference. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2024. Chap. 6.14. URL: `https://docs.python.org/3/reference/expressions.html#lambda` (visited on 2024-10-09) (cit. on p. 139).

[165] 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. 294).

[166] 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, 297).

[167] 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. 294).

[168] 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. 297).

[169] 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. 78, 86, 89, 93).

[170] Kent D. Lee and Steve Hubbard. *Data Structures and Algorithms with Python*. Undergraduate Topics in Computer Science (UTICS). Cham, Switzerland: Springer, 2015. ISBN: 978-3-319-13071-2. doi:10.1007/978-3-319-13072-9 (cit. on pp. 1, 295).

[171] 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. 248, 288).

[172] 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. 69, 70, 289, 294).

[173] 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. 194, 287).

[174] 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. 295).

[175] Reuven M. Lerner. *Pandas Workout*. Shelter Island, NY, USA: Manning Publications, June 2024. ISBN: 978-1-61729-972-8 (cit. on pp. 3, 294).

[176] *LibreOffice – The Document Foundation*. Berlin, Germany: The Document Foundation, 2024. URL: `https://www.libreoffice.org` (visited on 2024-12-12) (cit. on pp. 190, 246, 293, 294).

[177] 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. 158).

[178] 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. 293).

[179] 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 p. 217).

[180] 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. 123, 291).

[181] *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).

[182] 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. 83, 289, 295).

[183] 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 p. 45).

[184] 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. 296).

[185] 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, 141).

[186] 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. 137).

[187] 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. 259, 298).

[188] *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. 190, 246, 294).

[189] 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 p. 295).

[190] 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. 243).

[191] 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. 197–199, 204, 314).

[192] 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. 291, 328).

[193] Thomas Nield. *An Introduction to Regular Expressions*. Sebastopol, CA, USA: O'Reilly Media, Inc., June 2019. ISBN: 978-1-4920-8255-2 (cit. on p. 295).

[194] 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. 89, 217).

[195] 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 pp. 30, 291).

[196] "Numeric Types – `int`, `float`, `complex`". In: *Python 3 Documentation*. Vol. The Python Standard Library. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2024. URL: https://docs.python.org/3/library/stdtypes.html#typesnumeric (visited on 2024-07-05) (cit. on p. 30).

[197] 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, 294).

[198] 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. 73).

[199] 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) (cit. on p. 58).

[200] 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. 295).

[201] "`object.__hash__(self)`". In: *Python 3 Documentation*. Vol. The Python Language Reference. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2024. URL: https://docs.python.org/3/reference/datamodel.html#object.__hash__ (visited on 2024-12-09) (cit. on pp. 217, 218, 287).

[202] "Objects, Values and Types". In: *Python 3 Documentation*. Vol. The Python Language Reference. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2024. Chap. 3.1. URL: https://docs.python.org/3/reference/datamodel.html#objects-values-and-types (visited on 2024-12-07) (cit. on p. 211).

[203] 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. 128, 295).

[204] 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. 291).

[205] 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, 294).

[206] 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. 128, 295).

[207] 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. 73).

[208] 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, 294).

[209] 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, 295).

[210] 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. 296).

[211] *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. 190, 205).

[212] Fabian Pedregos, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake VanderPlas, 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, 295).

[213] 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. 273, 293).

[214] 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. 168, 169, 289).

[215] 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. 141, 292).

[216] 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. 258, 295).

[217] 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. 259, 290).

[218] "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. 295).

[219] *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), Feb. 20, 2025. URL: `https://www.postgresql.org/docs/17/index.html` (visited on 2025-02-25).

[220] *PostgreSQL Essentials: Leveling Up Your Data Work*. Sebastopol, CA, USA: O'Reilly Media, Inc., Mar. 2024 (cit. on p. 295).

[221] 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. 72, 102, 137).

[222] 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. 9).

[223] 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 & Assessment, 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. 32, 284).

[224] *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. 26, 291).

[225] "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).

[226] Pylint Contributors. *Pylint*. Toulouse, Occitanie, France: Logilab, 2003–2024. URL: `https://pylint.readthedocs.io/en/stable` (visited on 2024-09-24) (cit. on pp. 115, 289, 295).

[227] *Python 3 Documentation*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2024. URL: `https://docs.python.org/3` (visited on 2024-07-05).

[228] "Python Setup and Usage". In: *Python 3 Documentation*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2024. URL: `https://docs.python.org/3/using` (visited on 2024-07-05) (cit. on p. 7).

[229] *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. 292).

[230] 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, 295).

[231] 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 p. 291).

[232] Federico Razzoli. *Mastering MariaDB*. Birmingham, England, UK: Packt Publishing Ltd, Sept. 2014. ISBN: 978-1-78398-154-0 (cit. on p. 294).

[233] "`re` – Regular Expression Operations". In: *Python 3 Documentation*. Vol. The Python Standard Library. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2024. URL: `https://docs.python.org/3/library/re.html#module-re` (visited on 2024-11-01) (cit. on p. 295).

[234] 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), Oct. 2021, pp. 1–8. ISSN: 2577-1647. ISBN: 978-1-6654-3554-3. doi:`10.1109/IECON48115.2021.9589382` (cit. on p. 294).

[235] *Repositories Documentation*. San Francisco, CA, USA: GitHub Inc, 2025. URL: `https://docs.github.com/en/repositories` (visited on 2025-02-05) (cit. on p. 273).

[236] 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. 297).

[237] 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. 291).

[238] Hans Riesel. *Prime Numbers and Computer Methods for Factorization*. 2nd ed. Progress in Mathematics (PM). New York, NY, USA: Springer Science+Business Media, 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. 110, 172, 183, 184).

[239] 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. 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. 71).

[240] 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. 234, 289, 292).

[241] 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 p. 128).

[242] 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, 105, 291).

[243] 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 p. 293).

[244] 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. 32).

[245] 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. 182, 185).

[246] 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. 116, 117).

[247] 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. 293, 294).

[248] "Sequences". In: *Python 3 Documentation*. Vol. The Python Language Reference. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2024. Chap. 3.2.5. URL: https://docs.python.org/3/reference/datamodel.html#sequences (visited on 2024-08-24) (cit. on pp. 78, 82).

[249] "Set Types – `set`, `frozenset`". In: *Python 3 Documentation*. Vol. The Python Standard Library. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2024. URL: https://docs.python.org/3/library/stdtypes.html#set (visited on 2024-08-27) (cit. on pp. 89, 91).

[250] "Set Types". In: *Python 3 Documentation*. Vol. The Python Language Reference. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2024. Chap. 3.2.6. URL: https://docs.python.org/3/reference/datamodel.html#set-types (visited on 2024-08-27) (cit. on p. 91).

[251] 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. 292).

[252] 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. 141).

[253] Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge, England, UK: Cambridge University Press & Assessment, 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, 105, 294).

[254] 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) (cit. on p. 198).

[255] 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. 291, 293).

[256] 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 p. 183).

[257] Avi Silberschatz, Henry F. 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. 89, 93, 217).

[258] 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. 291).

[259] Anna Skoulikari. *Learning Git*. Sebastopol, CA, USA: O'Reilly Media, Inc., May 2023. ISBN: 978-1-0981-3391-7 (cit. on pp. 273, 293).

[260] Neil James Alexander Sloane. *Decimal Expansion of Golden Ratio phi (or tau) = (1 + sqrt(5))/2*. Ed. by John Horton Conway. Vol. A001622. 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. 243, 291).

[261] Neil James Alexander Sloane. *Decimal Expansion of Square Root of 2*. Ed. by John Horton Conway. Vol. A002193. 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. 243).

[262] 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. 293, 294).

[263] Eric V. 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 p. 45).

[264] 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. 295).

[265] *Snap Documentation*. London, England, UK: Canonical Ltd., 2025. URL: https://snapcraft.io/docs (visited on 2025-01-01) (cit. on p. 9).

[266] 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. 194, 222, 287).

[267] "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. 296).

[268] 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. 74, 296).

[269] 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. 247, 288).

[270] Bogdan Stashchuk. *Git and GitHub Crash Course*. Birmingham, England, UK: Packt Publishing Ltd, June 2021. ISBN: 978-1-80181-370-9 (cit. on p. 273).

[271] 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. 293, 297).

[272] *`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. 211).

[273] 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 p. 58).

[274] 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. 194, 195, 222, 230, 287).

[275] Allen Taylor. *Introducing SQL and Relational Databases*. New York, NY, USA: Apress Media, LLC, Sept. 2018. ISBN: 978-1-4842-3841-7 (cit. on p. 295).

[276] "Text Sequence Type – `str`". In: *Python 3 Documentation*. Vol. The Python Standard Library. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2024. URL: https://docs.python.org/3/library/stdtypes.html#textseq (visited on 2024-07-25) (cit. on p. 45).

[277] 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. 294, 295).

[278] The Editors of Encyclopaedia Britannica, ed. *Encyclopaedia Britannica*. Chicago, IL, USA: Encyclopædia Britannica, Inc.

[279] The Editors of Encyclopaedia Britannica, 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. 291).

[280] The Editors of Encyclopaedia Britannica, 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. 98).

[281] "The Import System". In: *Python 3 Documentation*. Vol. The Python Language Reference. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2024. Chap. 5. URL: https://docs.python.org/3/reference/import.html (visited on 2024-10-01) (cit. on p. 126).

[282] *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. 246, 293).

[283] *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. 258, 295).

[284] *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. 51, 297).

[285] "The `with` Statement". In: *Python 3 Documentation*. Vol. The Python Language Reference. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2024. Chap. 8.5. URL: https://docs.python.org/3/reference/compound_stmts.html#with (visited on 2024-12-15) (cit. on p. 157).

[286] 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 p. 128).

[287] "`timeit` – Measure Execution Time of Small Code Snippets". In: *Python 3 Documentation*. Vol. The Python Standard Library. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2024. URL: `https://docs.python.org/3/library/timeit.html` (visited on 2024-11-07) (cit. on pp. 168, 169, 286, 289).

[288] 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. 7, 293).

[289] 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. 3, 292).

[290] 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. 293).

[291] 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. 273, 293, 298).

[292] 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. 296).

[293] Laurie A. Ulrich and Ken Cook. *Access For Dummies*. Hoboken, NJ, USA: For Dummies, Dec. 2021. ISBN: 978-1-119-82908-9 (cit. on pp. 293, 294).

[294] *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. 51, 297).

[295] *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. 50).

[296] 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. 295).

[297] 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. 7, 295).

[298] 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. 3).

[299] 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. 214, 227).

[300] 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. 157).

[301] 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. 71–73, 297).

[302] 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, 50, 54, 55, 84, 121, 122, 126, 191, 222, 284–287, 292).

[303] Sander van Vugt. *Linux Fundamentals*. 2nd ed. Hoboken, NJ, USA: Pearson IT Certification, June 2022. ISBN: 978-0-13-792931-3 (cit. on pp. 291, 293).

[304] 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, 295).

[305] 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. 73, 74, 296).

[306] "`venv` – Creation of Virtual Environments". In: *Python 3 Documentation*. Vol. The Python Standard Library. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2024. URL: `https://docs.python.org/3/library/venv.html` (visited on 2024-12-25) (cit. on p. 298).

[307] 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 Pedregos, 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, 295).

[308] "Virtual Environments and Packages". In: *Python 3 Documentation*. Vol. The Python Tutorial. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2024. Chap. 12. URL: `https://docs.python.org/3/tutorial/venv.html` (visited on 2024-12-24) (cit. on pp. 259, 298).

[309] 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. 214).

[310] 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. 173).

[311] 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. 166).

[312] 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), Institute of Applied Optimization (应用优化研究所, IAO), 2025. URL: `https://thomasweise.github.io/databases` (visited on 2025-01-05) (cit. on pp. 74, 273, 279, 280, 292, 293, 295, 296).

[313] 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), Institute of Applied Optimization (应用优化研究所, IAO), 2024–2025. URL: `https://thomasweise.github.io/programmingWithPython` (visited on 2025-01-05) (cit. on pp. iv, 273, 295).

[314] Thomas Weise (汤卫思) and Zhize Wu (吴志泽). "Replicable Self-Documenting Experiments with Arbitrary Search Spaces and Algorithms". In: *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` (cit. on pp. 3, 294).

[315] 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 p. 183).

[316] 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).

[317] 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. 110, 172, 184).

[318] *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. 190, 205).

[319] *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. 295).

[320] 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. 128).

[321] 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. 294).

[322] Kevin Wilson. *Python Made Easy*. Birmingham, England, UK: Packt Publishing Ltd, Aug. 2024. ISBN: 978-1-83664-615-0 (cit. on pp. 234, 289, 292, 295).

[323] 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. 133, 296).

[324] 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. 122, 285).

[325] "With Statement Context Managers". In: *Python 3 Documentation*. Vol. The Python Language Reference. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2024. 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. 157, 247).

[326] Martin Yanev. *PyCharm Productivity and Debugging Techniques*. Birmingham, England, UK: Packt Publishing Ltd, Oct. 2022. ISBN: 978-1-83763-244-2 (cit. on pp. 7, 295).

[327] 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. 292).

[328] 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 p. 164).

[329] Erin Yepis. "Developers want more, more, more: The 2024 results from Stack Overflow's Annual Developer Survey". In: *Stack Overflow*. New York, NY, USA: Stack Exchange Inc., Jan. 1, 2025. URL: `https://stackoverflow.blog/2025/01/01/developers-want-more-more-more-the-2024-results-from-stack-overflow-s-annual-developer-survey` (visited on 2025-01-07) (cit. on p. 3).

[330] 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 [138] (cit. on pp. 155, 297).

[331] 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. 296).

[332] Giorgio Zarrelli. *Mastering Bash*. Birmingham, England, UK: Packt Publishing Ltd, June 2017. ISBN: 978-1-78439-687-9 (cit. on pp. 291, 328).

[333] 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. 74, 296).

[334] 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, 296).

[335] 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): 工业和信息化部 (Ministry of Industry and Information Technology): 国家市场监督管理总局 (State Administration for Market Regulation) and 国家标准化管理委员会 (Standardization Administration of the People's Republic of China), 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. 50).

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,

Listing 16.1: A Bash script for executing Mypy, which prints the command line and the exit code; see Useful Tool 2. (src)

```bash
#!/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.
$command 2>&1
exitCode="$?"  # 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."
```

so Bash shell scripts are not in the center of our attention. We here cannot explain how Bash scripts work or what their syntax is. There exist plenty of books and resources on this interesting topic, such as [33, 192, 332] 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. 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 4.19 and 4.20 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 for 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 7.15 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 for the output of pytest. Listing 16.5 uses pytest as well, but this time executes doctests. An example can be found in Listing 10.9.

Listing 16.2: A Bash script for executing Ruff, which prints the command line and the exit code; see Useful Tool 3. (src)

```bash
#!/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.
$command 2>&1  # Run ruff.
exitCode="$?"  # 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 4. (src)

```bash
#!/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.
$command 2>&1  # Run pylint.
exitCode="$?"  # 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 5. (src)

```bash
#!/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.
$command 2>&1
exitCode="$?"  # 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 7. (src)

```bash
#!/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.
$command 2>&1
exitCode="$?"  # 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."
```