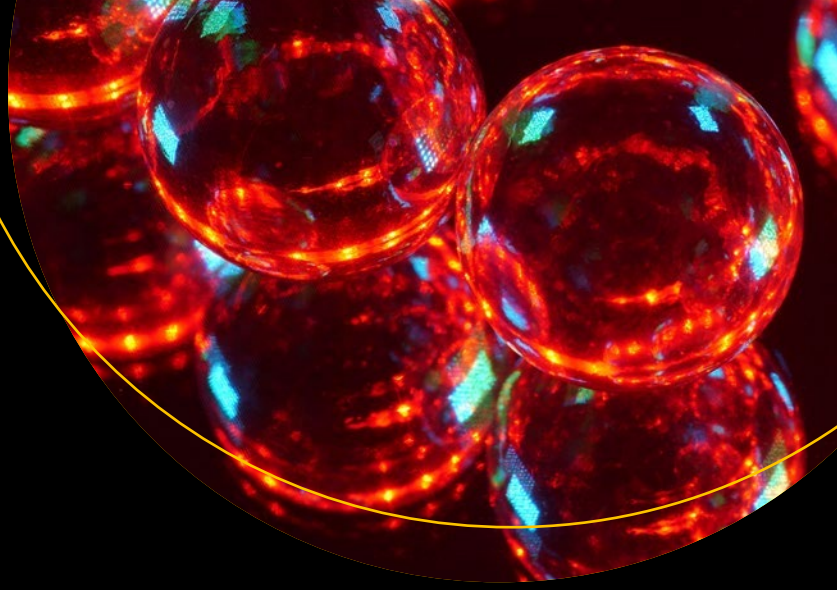

Introduction to Computational Thinking

Problem Solving, Algorithms, Data Structures, and More

Thomas Mailund

Apress®

Introduction to Computational Thinking

Problem Solving, Algorithms, Data Structures, and More

Thomas Mailund

Apress®

Introduction to Computational Thinking: Problem Solving, Algorithms, Data Structures, and More

Thomas Mailund
Aarhus N, Denmark

ISBN-13 (pbk): 978-1-4842-7076-9 ISBN-13 (electronic): 978-1-4842-7077-6
https://doi.org/10.1007/978-1-4842-7077-6

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Steve Anglin
Development Editor: Matthew Moodie
Coordinating Editor: Mark Powers

Cover designed by eStudioCalamar

Cover image by Engin Akyurt on Unsplash (www.unsplash.com)

Distributed to the book trade worldwide by Apress Media, LLC, 1 New York Plaza, New York, NY 10004, U.S.A. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484270769. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

Table of Contents

About the Author

Thomas Mailund is an associate professor in bioinformatics at Aarhus University, Denmark. He has a background in math and computer science. For the past decade, his main focus has been on genetics and evolutionary studies, particularly comparative genomics, speciation, and gene flow between emerging species. He has published *String Algorithms in C*, *R Data Science Quick Reference*, *The Joys of Hashing*, *Domain-Specific Languages in R*, *Beginning Data Science in R*, *Functional Programming in R*, and *Metaprogramming in R*, all from Apress, as well as other books.

About the Technical Reviewer

Troels Bjerre Lund is an associate professor of computer science at the IT University of Copenhagen. He is an expert in computational game theory and has been teaching programming and software development for the past seven years.

CHAPTER 1

Introduction

Using computers as more than glorified typewriters or calculators is an increasingly important aspect of any scientific or technological field, and knowing how to program a computer to solve new problems has become as essential a skill as mathematics. Learning how to program can be a frustrating experience at times since computers require a level of precision and rigor in how we express our ideas, which we rarely encounter elsewhere in life. While occasionally infuriating, programming can also be very rewarding. Programs are created out of pure thought, and it is a special feeling when you make a computer transform your ideas into actions and see it solve your problems for you.

Solving any kind of problem, on a computer or otherwise, requires a certain level of precision. To address the right question, we must first understand what the problem *is*. We also need to have a precise idea about what an adequate *solution* to the problem would be—or at the very least some way of distinguishing between two solutions to judge if one is better than the other. These are concerns we will need to address in any problem-solving task, but where everyday life might forgive some fuzzy thinking in problem solving, computers are far less forgiving. To solve a problem on a computer, you must first specify with mathematical clarity what the problem is and what a solution is and, after that, how you will go about deriving a solution. And only then can you write a program and put the computer to work.

For the novice programmer, the last step—implementing a solution in a computer language—is often the most frustrating. Computer programs do not allow any ambiguities, and that means that if you do not abide by the computer language's rules—if you get the grammar wrong in the slightest—the computer will refuse even to consider your program. Learning how to write programs the computer will even attempt to run is the first hurdle to overcome.

Many good books can teach you different programming languages, and it is worth your while getting a few of these about the programming languages you plan to use in your future work. This book is not only about programming, however, but about how

T. Mailund, *Introduction to Computational Thinking*, https://doi.org/10.1007/978-1-4842-7077-6_1

computation is done and how you can make computation efficient. Still, from time to time, I will show you tricks for making your programming efficient as well, meaning speeding up how fast you can write your programs, which is a separate issue from how efficient your programs are once you have implemented them. These tricks are generally applicable, provided your programming language supports the features we use, and for a working programmer, efficient *programming* is as important as efficient *programs*. I won't cover these tricks as separate topics, but show them when we study topics where they are useful.

For the programming we do in the book, we will use the Python programming language. The Python language is generally considered an excellent first language to learn because of its high-level yet intuitive features, and at the same time, Python is one of the most popular programming languages for scientific programs. It is one of the most frequently used languages for data science. It is number one on the Kaggle machine learning platform (`www.kaggle.com`). It has powerful libraries for machine learning, data analysis, and scientific computing through various software modules. It is also one of the most popular languages for scripting workflows of data analysis and for administrating computer systems.

We will not explore the full language, however, as the book is already long without discussing all the powerful features in Python. But I will show you how the code we write would look if you implemented it like a "real" Python programmer, with the features of the language you would need to get there. You can safely ignore those parts if you only want to learn the aspects of programming that generalize to other languages and if you are only interested in how to write effective and efficient code. In any language you use regularly, however, it is worth knowing the programming style and idioms for that language. Styles do not directly translate from one language to another, since what is easy to do in one might be hard in another and vice versa. Spend the time to learn how experienced programmers use a language when you learn it, if you want to be effective.

We use Python to illustrate ideas and for exercising topics we cover, but the focus of the book is not on Python programming. The focus is how to think about problem solving in a disciplined way, to consider problems as computational tasks, and how to plan solutions in ways that are computationally efficient. Along the way, we will see different programming paradigms that will show you how you can think about programming in different ways. Primarily, these will be *functional programming* and *object-oriented programming*, both of which are supported by Python. Thinking about structuring data, about how to efficiently manipulate data to solve problems, and how to structure your code so it is easy to write, easy to extend, and easy to maintain is what we mean by *computational thinking*.

Models of the World and Formalizing Problems

Our goal is to learn how to *formalize objectives* in such a way that we can specify *mathematically and objectively* what solutions to our goals are. This also means formalizing what data we have and how we should interpret it. Formalizing a problem might reveal that we do not have sufficient data for the issue at hand. It might also show that we do not truly understand our problem. If we cannot clearly define what we want, we won't be able to formalize how to get it. We might, with some luck, be able to fudge it a bit and get *something* and then use subjective opinion to judge if what we get is what we wanted. This is far from optimal, though. If you and I disagree on whether one solution is better than another or not, we have no way of resolving the issue.

Formalizing problems and formalizing what data we have to work with is what you do in all natural sciences. You might not have thought about it this way before—depending on which science you have a background in—but when we derive theories (or laws) about the natural world, we are making formal statements about how the world works. For some theories there are exceptions—the world is breaking a natural law—which tells us we do not have a comprehensive theory. But any theory worth its salt can be falsified, which is another way of saying that we can judge if a data point matches the formalization of the theory or not.

In the hard sciences, like physics and chemistry, these theories are described in the language of mathematics, often in somewhat complex equations. In sciences describing very complex systems, such as biology that tries to explain life in general, we often have much simpler mathematics, and the rules almost always have exceptions. Biology is more complicated than particle physics, so it is harder to formalize, and thus we stick with simpler equations. There is no point in using very complex mathematics to describe something we do not understand—simple mathematics suffices for that. Any quantitative evaluation of the natural world requires some mathematics and some formalization of scientific theories—even if the mathematics is as simple as counting to see if some quantity is more abundant in some situations than others. All quantitative data analysis involves formalizing our thoughts about reality and reducing data to the relevant aspects for those formal descriptions.

Abstracting the complex natural world to something more manageable is called *modeling*. We build models of the real world—usually mathematical models. We aim at making the models simple enough to understand, yet sophisticated enough to describe the aspects of the world we are interested in. In principle, we could model molecular evolution as a physical system at the level of particles. We don't, because this would be

much too complex for us to work with and probably wouldn't help us answer most of the questions about evolution we are interested in. Instead, we model molecular evolution as random mutations in strings of DNA, abstracting the three-dimensional DNA molecules into one-dimensional strings over the four letters A, C, G, and T. We abstract away aspects of the world that are not relevant for the models, and we abstract away features about the data that are not modeled.

Building models of the natural world is the goal of all the sciences and much too broad a topic for this book. The models are relevant for computational thinking, however. When we formalize how to solve problems, we do so within a model of the world. This model will affect how we can formalize problems and at which level of detail we consider our data. Sometimes, changing the model of reality can change what can be efficiently computed—or make an easy problem intractable. Of course, we should not pick scientific theories based on what we can efficiently calculate, but sometimes, abstracting away aspects of the world that are not essential for the problem at hand will not qualitatively change solutions but might make an otherwise impossible problem easy.

This book is not about modeling the world. We will generally assume that we have some formal models to work with within whatever scientific field we find ourselves. You are rarely in the situation where you can pick your theories at random to satisfy your computational needs, but keep in mind that formalizing the *problem* you want to solve might give you some wiggle room within those formal scientific theories. When, for example, we study genome evolution or population genetics, we abstract complex DNA molecules to the level of strings or reduce populations to gene frequencies. These abstractions are there to simplify the subject matter to something that can be attacked computationally.

What Is Computational Thinking?

Computational thinking is what you do when you take a problem and formalize it, when you distil it into something where you can objectively determine if something is a solution to it or not. For example, given a sequence of numbers, are all positive? Easy to check, and either all the numbers are positive or they are not. Or perhaps the problem is not a yes-no question but an optimization issue. Finding the shortest route to get from points A to B is an optimization problem. It might be easy for us to determine if one path is shorter than another, which would be a yes-no problem, but actually coming up with

short routes might be a harder problem. It is still a computational problem, as long as we can formalize what a path is and how we measure distance.

Computational thinking is also what happens after you have formalized the problem, when you figure out how to solve it. A formal description for how to solve a problem is called an *algorithm*, after the ninth-century mathematician Muhammad ibn Musa al'Khwarizmi who is also responsible for the term algebra. To qualify as an algorithm, a description of how to solve a problem must be in sufficient detail that we can follow it without having to involve any guesswork. If you implement it on a computer, I guarantee you that you do not want to leave any room for guessing. The description must always get to a solution in a finite number of steps—we don't want to keep computing forever—and the description must always lead to a valid solution—we don't want to follow all the steps and end up with something we cannot use anyway.[1]

Designing algorithms is part science and part art. There are general guidelines we can use to approach a computational problem to develop algorithms and general approaches to organizing data such that we can manipulate it efficiently, but you will almost always have to adapt the general ideas to your specific problem. Here, sparks of insight cannot be underestimated—sometimes, just looking at a problem in different ways will open entirely new ways of approaching it. The general approaches can be taught and learned and are the main topic of this book. The art of designing algorithms comes with practice, and as with all skills, the more you practice, the better you get.

Most of the algorithms we will see in this book are used in almost all software that runs on your computer (with the exceptions of some toy examples found in the exercises that are never used in the wild). Sorting and searching in data and arranging data for fast retrieval or fast update is part of almost all computations. The models behind such algorithms are often exceedingly abstract, much more so than any model we would use to describe real-world phenomena. A sorting algorithm might work in a world where the only thing you can do with objects is to determine which of two objects is smaller.

[1]There are exceptions to the requirement that an algorithm should always complete in a finite number of steps. When we implement something like a web service or an operating system, we don't want our programs to terminate after a finite number of calculations, but instead want them to run, and be responsive, indefinitely. In those cases, we relax the requirement and require that they can respond to all events in a finite number of steps. We also have exceptions to always getting correct answers. Sometimes, we can accept that we get the right answer with high probability—if we can quickly test if the answer is correct and maybe rerun the algorithm for another solution and continue this until we get lucky. These are unusual cases, however, and we do not consider them in this book.

Or maybe the algorithm works in a model that allows more structure to data, and this structure can be exploited to make the algorithm more efficient. In any case, what we can do with data depends on our models, and for computation, these models are often remarkably abstract. Such abstract models can feel far from the world your problem originates in, but it is because the models are so very theoretical that we can apply the algorithmic solutions to so many varied problems.

Some people spend their entire lives developing new algorithms for general problems. Those people would be professional computer science academics. Most people who solve problems on computers are not doing this, even if they develop algorithms on a daily basis. When we deal with concrete issues, we can usually do so by combining existing algorithms in the right ways. Having a toolbox of algorithms to pick from and knowing their strengths and weaknesses is more important in day-to-day computational work than being able to design algorithms entirely from scratch—although that can be important as well, of course, on the rare occasions when your toolbox does not suffice.

Whether you can get where you want to go by combining existing algorithms or you have to design new ones, the general approach is the same. You have to break apart big tasks that you do not know how to solve (yet) into smaller tasks that, when all done, will have completed the larger tasks. Steps to a job, such as "find the largest number in a sequence," can be broken into smaller steps such as "compare the first two numbers and remember the largest," "compare the largest of the first two to the third and remember the largest," and so on. You start out with one big task—the problem you want to solve—and you keep breaking down the problem into smaller tasks until they all are tasks you know how to handle—either because they are trivial or because you have an algorithm in your toolbox that can solve them. The practice of breaking down tasks until you can resolve them all is at the heart of computational thinking.

Developing and combining algorithms is a vital part of computational thinking, but algorithms alone do not solve any problems. Algorithms need to be executed to solve concrete problems; we need to follow the instructions they provide on actual data to get actual solutions. Since we rarely want to do this by hand or with pen and paper, we wish to instruct computers how to run algorithms, which means that we have to translate a high-level description of an algorithm to a lower-level description that can be put into a computer program that the machine will then slavishly execute. This task is called *implementing* the algorithm.

Designing an algorithm and implementing it as a computer program are two separate tasks, although tightly linked. The first task involves understanding the problem you want to solve in sufficient detail that you can break it down into pieces that you know how to address. The second task consists in breaking those pieces into even smaller ones that the computer can solve; this is where the algorithm design task meets the programming task.

The abstraction level at which you can implement an algorithm depends intimately on the programming language and the libraries of functionality you have access to. At the most basic level—the hardware of your computer—the instructions available do little more than move bits around and do basic arithmetic.[2] A modern CPU is a very sophisticated machine for doing this, with many optimizations implemented in hardware, but the basic operations you have at this level are still pretty primitive. This is the level of abstraction where you can get the highest performance out of your CPU, but we practically never program at this level, because it is also the level of abstraction where you get the lowest performance out of a programmer. Basic arithmetic is just too low a level of abstraction for us to think about algorithms constructively.

Programming languages provide higher levels of abstraction to the programmer. They can do this because someone has written a program that can translate the high-level operations in the programming language into the right sequence of lower-level operations that the computer can actually execute.[3] Which abstractions are available varies tremendously between programming languages, but they all need to describe programs that are eventually run at the low level of the computer's CPU. The programming language abstractions are just an interface between the programmer and

[2]Okay, at a more fundamental level, a computer is a rock you can communicate with through electricity, but from a computational perspective, basic arithmetic operations are as primitive as they come.

[3]If you think about it, there is an interesting question on how programs that translate high-level instructions into low-level instructions are written. It is hard enough to write a program that works correctly in a high-level programming language; it is substantially harder to do in the language the machine understands. You want to write the programs for dealing with high-level abstractions in programming languages that support these abstractions. But to support the abstractions, you need a program that implements them. That is a circular dependency, and that is problematic. You can solve it by first writing primitive programs that support some abstractions. Now you can use these abstractions to write a *better* program that can handle more abstractions. This, in turn, lets you write even better programs with better abstractions. At this point, you can throw away the most primitive programs because you have implemented a programming language that you can use to implement the programming language itself. This process is known as *bootstrapping*, named after the phrase *"to pull oneself up by one's bootstraps."*

the machine, and the language's implementers have handled how these abstractions are executed at the lower layers of the computer.

We sometimes talk about high-level and low-level programming languages, but there isn't a real dichotomy. There are merely differences in the higher-level abstractions provided by all programming languages. Some programming languages provide an environment for programming very close to the hardware, where you can manipulate bits at the lowest level while still having some abstractions to control the steps taken by your program and some abstractions for representing data beyond merely bit patterns. These we would call low-level languages because they aim to be close to the lowest level of abstraction on the computer. Other languages, high-level languages, provide a programming environment that tries to hide the lower levels to protect the programmer from them. How data is actually represented at lower levels is hidden by abstractions in the language, and the programming environment guarantees that the mapping between language concepts and bits is handled correctly.

Computational Thinking in a Broader Context

To summarize, what we call computational thinking in this book refers to a broad range of activities vital for solving problems using a computer. For some of those activities, computational thinking is merely a tiny aspect. Making models of the real world in order to understand it is the entire goal of science; considering scientific theories in the light of how we can make computations using the equations that come out of the theories is a minute aspect of the scientific process, but an essential one if you want to use your computer to do science. Creating new algorithms to solve a particular problem is also almost entirely computational thinking in action; implementing these algorithms, on the other hand, can be an almost mechanical process once you have fleshed out the algorithm in sufficient detail.

One thing that sometimes complicates learning how to think about computations is that there is rarely a single right answer to any problem you consider. It shares this with natural sciences. While we usually believe that there is a unique natural world out there to explore, we generally do not attempt to model it in full detail; an accurate model of reality would be too complicated to be useful. Instead, we build models that simplify reality, and there is no "right" model to be found—only more or less valuable models. When we seek to solve a problem on a computer, we are in the same situation. We need

to abstract a model of reality that is useful, and there may be many different choices we can reasonably make in doing this, all with different pros and cons.

For any of these models, we have a seemingly endless list of appropriate algorithms we can choose from to solve our problem. Some will be horrible choices for various reasons. They might not solve the problem at hand in all cases, or at all, or they might solve the problem but take so long to do this that in practical terms they never finish computing. Many of the choices, however, will solve the problem and in a reasonable time, but use different computational resources in doing so. Some run faster than others; some can exploit many CPUs in parallel, solving the problem faster but using more resources to do so; some might be fast but require much more memory to solve the problem and therefore might not be feasible solutions given the resources you have. It requires computational thinking to derive these algorithms, but it is also computational thinking to reason about the resources they need and to judge which algorithms can be used in practice and which cannot.

Once you have chosen an appropriate algorithm to solve your problem, you need to implement it to execute it. On itself, the algorithm is useless; only when it is executed does it have any value, and executing it on a computer means you have to implement it as a computer program first. At this step, you need to decide on a computer programming language and then how to flesh out the details the algorithm does not specify. For choosing the programming language, you once again have numerous choices, all with different strengths and weaknesses. Typically, the first choice is between the speed and speed—how fast can you implement the algorithm in a given language vs. how fast it will run once you have implemented it. Typically, high-level languages let you implement your ideas more swiftly, but often at the cost of slightly (or less slightly) slower programs. Low-level languages let you control your computer in greater detail, which allows you to implement faster programs, but at the cost of also having to specify details that high-level languages will shield you from. You shouldn't always go for making your programs as fast as possible; instead, you should go for solving your actual problem as speedy as possible. You can make your program very fast to run by spending a vast amount of time implementing it, or you can implement it quickly and let it run a little longer. You want to take the path that gets you to your solution the fastest. Here, of course, you should also take into account how often you expect to use your program. A program that is run often gains more from being faster than one that runs only for a specific project and only a few times there.

In reality, the choice of programming language is not between all possible languages, but between the languages you know how to write programs in. Learning a new programming language to implement an algorithm is rarely, if ever, worth the time. If you only know one language, the choice is made for you, but it is worthwhile to know a few, at least, and to know both a high-level and a low-level language with sufficient fluency that you can implement algorithms in them with comfortable fluency. This gives you some choice in what to choose when you have a program to write.

The choices aren't all made once you have decided on the programming language, though. There will always be details that are not addressed by your algorithm, but that must be addressed by your program. The algorithm might use different abstract data structures, such as "sets" or "queues" or "tables," and it might also specify how fast operations on these have to be, but when you have to make concrete implementations of these structures or choose existing implementations from software libraries, there are more options to consider. In high-level programming languages, there are fewer details you have to flesh out than in low-level languages, which is one of the reasons it is usually much faster to implement an algorithm in a high-level language than in a low-level language—but there will always be some choices to be made at this point in the process as well.

You might hope you are done when you have implemented your algorithm, but this is usually not the case. You need to feed data into the program and get the answer out, and here you have choices to make about data formats. Your program will not live in isolation from other programs, either, but communicate with the world, usually in the form of files and data formatted in different file formats. Again, there are choices to be made for how you wrap your algorithm in a program. If your algorithm is useful for more than a single project, you might also put it in a software library, and then there are choices to be made about how you provide an interface to it. If you build a whole library of different algorithms and data structures, constructing interfaces to the library is full of critical design decisions, and these decisions affect how other programs can use the algorithms and how efficiently, so this is also an aspect of computational thinking—but here only a part of the broader topic of software engineering.

What Is to Come

The purpose of this book is to introduce computational thinking as basic problem-solving approaches for designing algorithms and implementing them in a computer language, the Python language. We will focus on the design of algorithms more than the implementation of them and only use a subset of the Python programming language for exercises. We will use the Python features necessary so our code behaves the way a Python programmer would expect, with the idioms of the language, but we will mainly use a subset of Python for the core code that you will find in many other languages. This will make it easier to transfer what you learn to other programming languages, but keep in mind that it also means that the solutions we consider are not necessarily the solutions an experienced Python programmer would come up with. There are ways of expressing things in Python that can implement our algorithms more effectively, but those are Python specific and might not be found in other languages.

In many of the following chapters, I will explain how computation is done on an actual computer, not just in Python but on computers in general. General computers do not understand Python programs but do understand more primitive instructions that you can give a CPU, and I will try to put our Python programs in a context of these. I will also explain how computers store data, which they can only do using simple memory words consisting of ones and zeros. These explanations are far from comprehensive and are only intended to give you a feeling for how instructions in a high-level programming language such as Python will have to be translated into much lower-level concepts on actual hardware. When I do explain these concepts, I will not always be completely honest about how Python *actually* handles these issues. Since Python is a very high-level programming language, it supports features that are not found in lower-level languages, and this means that to run a Python program, you need a more complex model of both data and code than you will need in many other languages. I will explain general concepts, but I will give a simplified explanation of them. If you want to know the details of how your computer really deals with these concepts and how Python handles these and more complex features of the language, you will need to find this information elsewhere.

We use a real programming language to explain the algorithms in the book to make it easier for you to experiment with them. Many algorithmic textbooks will not, preferring to describe algorithms in pseudo-code where the abstractions can be fitted to the problem. This might make the description of algorithms slightly more accessible, but can also easily hide away the issues that you will have to resolve actually to implement them.

We prefer to use an actual language. It is a very high-level language, so some details that you will have to deal with in lower-level languages are still hidden from you, but what you can implement in Python you can actually run on your computer. And it is vital that you do take the code in this book and experiment with it.

To get the full benefit out of this book, or any book like it, you must practice. And practice a lot. Programming can look deceptively easy—at least for the complexity level we consider in this book—but it is substantially harder to write your own code than it is to read and understand code already written.[4] Without exercising the skills involved in computational thinking and algorithmic programming, at best you will get a superficial understanding. Watching the Olympics doesn't prepare you for athletics. Each chapter has an exercise set associated with it, and you should expect to use at least as much time doing exercises as you spend reading the chapters if you want the full benefit out of the book.

[4]An interesting thing is that to inexperienced programmers, and with simple programs, it is a lot easier to read a program than to write it. The opposite is the case for experienced programmers working in more extensive and more complex programs. Once programs reach a certain level of complexity, they get harder to read than to write, and a lot of software engineering aims at alleviating this.

CHAPTER 2

Introducing Python Programming

Many textbooks on algorithms will present the algorithms in so-called *pseudo-code*, something that looks like it is written in a real programming language while it is in fact written in an approximation to such a language but where the abstractions and programming constructs are chosen to make the algorithms look as simple as necessary. Since the goal of these books is to present the essentials of an algorithm and not distract the reader with unnecessary language artifacts, it is a sensible approach. It does, however, occasionally hide too many details from the reader, and since the pseudo-code cannot be run by a computer, it is not possible to experiment with it to test different approaches to how an algorithm could be implemented in practice. In this book, we will not use pseudo-code but present all algorithms in the Python programming language. Python is a very high-level language, and in many ways, Python implementations of common algorithms look very similar to pseudo-code versions of them, but with Python, you get a working implementation.

Python is a general-purpose programming language with many advanced features, and it scales well to constructing large software systems. At the same time, it has a very gentle learning curve and lets you implement small programs with minimal programming overhead. It is perfect for our purpose in this book. By knowing just a tiny subset of the language, you will be able to implement the algorithms we cover, and you will be able to experiment with them. Should you decide to make more of a career out of programming, then you can easily pick up the more advanced features of Python and use this language for larger projects as well.

Writing complete programs, especially more extensive applications, requires different skills than the computational thinking we cover in this book. It takes a different skill set to be able to engineer software such that it is scalable and maintainable than the skills that are needed to build efficient algorithms. Those software engineering skills are

T. Mailund, *Introduction to Computational Thinking*, https://doi.org/10.1007/978-1-4842-7077-6_2

beyond the topics here. If you are interested in writing larger systems, there are many excellent textbooks on the market.

Obtaining Python

When you write programs in Python, you will usually do this in one or more plain-text files using a text editor. You cannot use word processors such as Microsoft Word since these do not save their documents as plain text. An excellent editor that supports Python and is available on Windows, macOS, and Linux is Visual Studio Code (`https://code.visualstudio.com`). When you save a file, give it the suffix `.py`. This is not necessary to have Python run your program, but it is the convention, and it makes it easier to recognize that a file contains a Python program.

You can download Python from `www.python.org/`. There are installers for Windows and macOS, and if you use Linux, then chances are that Python is already installed. If not, the package manager on your platform will be able to install it. The dialect of Python we will use in this book is Python 3.x (version numbers that start with 3). The differences between Python 2.x and 3.x, for the purpose of the algorithms we will explore here, are very minor, and all the algorithms in this book work equally well in either version. There are differences in the built-in functions, though, so you should download the installer for a Python 3.x to get exactly the behavior as described here.

In a few places, we will use additional functionality from what you get with a basic Python installation, however. We use a module called `Numpy` for tables when we cover dynamic programming in Chapter 9, and I show you plotting code for empirically validating the running time of algorithms in Chapter 4 where I use a module called `matplotlib`. You do not need either to follow the book, however. There are alternatives to `Numpy` for tables, and you can always plot running times in spreadsheets or other plotting programs. You can install these packages using a tool, `pip`, that is installed together with Python

```
pip install <package name>
```

but if you do not mind a larger installation—taking up more space on your disk—you can install Python together with many packages for scientific computing and data science from `www.anaconda.com`. We only use a tiny fraction of the software that is installed via Anaconda, but everything we do use will be available to you once you have installed Anaconda. If you continue programming in Python after you have read this book,

chances are that you will find a good use for many of the other modules installed by Anaconda, especially if you want to continue your career using Python for data science.

Running Python

When you have installed Python, written a program, and saved it to a file, say `file.py`, then you can run your program by writing

```
python3 file.py
```

in your terminal. If you want to use Python interactively, you can write

```
python3
```

Press Ctrl+D to leave the Python terminal again.

This will give you a Python terminal where you can write instructions to Python and get its response back. For the next few sections, you can just do that. I suggest you type all the following examples into your Python terminal to test the results they give you.

Expressions in Python

Try writing some arithmetic expressions into the Python terminal, for example

```
2 + 4
```

or

```
2 * 3
```

You can write expressions as you know and love them from basic arithmetic. The arithmetic binary operators `+`, `-`, `*`, and `/` work as you would expect them to from the arithmetic you learned in school,[1] as does the unary `-`. That is,

```
-4
```

[1]If you have used other programming languages or if you use Python 2, division might work differently from what you expect. In some languages, including Python 2, / is integer division if you divide two integers, so, for example, 3/2 would be 1 since 2 divides 3 once, with a remainder of 1. In Python 3, / works as you learned in school and is called "true division." Integer division has a separate operator, //; see in the following.

is minus four.

```
2 * 2 + 3
```

is 7. Notice that we interpret `2 * 2 + 3` as you are familiar with in mathematical notation. First, we multiply two by two, and then you add three. Python knows the precedence rules that say that multiplication binds stronger than addition. If you want another precedence, you use parentheses as you would with a pen and paper.

```
(3 + 2) * 2
```

would be 10.

When you do not use parentheses and arithmetic precedence is taken into account, the evaluation proceeds from left to right for the preceding binary operators (for exponentiation, the order is right to left; see in the following). For example,

```
1 / 2 / 2
```

is interpreted as $(1/2)/2 = 1/4 = 0.25$ and not $1/(2/2) = 1.0$. To get right-to-left evaluation, you will need to add parentheses:

```
1 / (2 / 2)
```

If you have not noticed that in the usual arithmetic notation, then trust me it is how it works there as well. In expressions like this, however, I would use parentheses as well to make the order explicit.

You can always use parentheses to change the default evaluation order or simply to make explicit what you intend, even if it is already the default. Writing `(2 * 2) + 3`, while it is the default for `2 * 2 + 3`, doesn't make the expression any harder to read, after all. I wouldn't use parentheses here, and you would probably not either, but you can if you want to.

There are more operations than addition, subtraction, multiplication, and divisions, for example, raising a number to a power. If you want to compute two to the power of four, $2^4 = 16$, you can use the ** operator: `2**4`. This operator has higher precedence than multiplication and division, so `2 * 2**4` is $2 \cdot 2^4 = 32$ and not $(2 \cdot 2)^4 = 256$. This operator is not evaluated left to right but right to left, also following typical mathematical notation. This means that `2**3**4` is interpreted as $2^{3^4} \approx 2.4 \times 10^{24}$ and not $(2^3)^4 = 4086$. If you want to compute the latter, you must write `(2**3)**4`. The notation *is* the same as

it would be in the usual mathematical notation, but if you find it hard to remember, you can always use parentheses to avoid any surprises, even when not strictly necessary.

Python 3 has two different division operators. When you use `/`, you get the division you are used to in mathematics; `1 / 3` gives you a third. However, it is often useful to guarantee that if you divide two integers, you get the integer result of the division. Remember that for division, n/m, you get an integer if m divides n. In general, you can write $n = a \cdot m + b$ where a is the integer number of times that m divides n and $b < m$ is the remainder. To get the integer part of the division, a in this example, you will use the `//` operator. So while `5 / 2` will give you 2.5, `5 // 2` will give you 2. To get the remainder, you use the modulus operator, `%`. Since $5 = 2 \cdot 2 + 1$, we would expect `5 % 2` to be one, and indeed it is. If you evaluate

```
(5 // 2) * 2 + (5 % 2)
```

the $a \cdot m + b$ form of this division, you get five, as expected. Another way to put this is that `5 // 2` is five divided by two, rounded down, or $\lfloor 5/2 \rfloor$.

The rules for integer division and remainder are always like these if we consider positive numbers n and m, but if n or m is negative, the result of the two operators is not necessarily as well defined, and different programming languages have made different choices.

An example might be the best way to illustrate this. Let's say we work on a program where we need to know what weekday any given day is. There are seven weekdays, and we can number them from Monday = 0 to Sunday = 6. If today is Tuesday, we would represent it as 1. If we want to know what day it is 10 days from now, we could do

```
today = 1 # Today is Tuesday
k = (today + 10) % 7 # Weekday 10 days from today
```

and we would find that `k=4`, so a Friday. All programming languages will agree on this. They will reason that you have moved forward 11 days, or `11//7=1` week, and then you have 4 days left, `11%7=4`.

But what if we want to know what weekday it was 10 days ago? Then we could do

```
k = (today - 10) % 7 # Weekday 10 days ago
```

Here, we are evaluating `-9 % 7`, and different programming languages make equally valid, but different, choices on what the result should be. Some will say that you move one week back, so –1 times 7 days, so `-9//7` should be –1, and then you have