Computational Thinking with Python

Pieter Spronck

Computational Thinking with Python

Pieter Spronck

Version 1.0.0

June 10, 2023

The original form of this book is LaTeX source code. Compiling this LaTeX source has the effect of generating a device-independent representation of a textbook, which can be converted to other formats and printed.

The LaTeX source for this book will (at some point in time) be available from `http://www.spronck.net/pythonbook`

The latest version of this book will always be available from `http://www.spronck.net/pythonbook`**.**

Preface

This course book is a companion for my book "The Coder's Apprentice: Learning Programming with Python 3."

The course is designed to teach computational thinking skills to students who manage to learn the elements of programming languages, but have troubles designing algorithms. It aims to provide students with the ability to think about problems in such a way that they can design a program to solve them.

The course assumes that the student knows the basics of a programming language. The course uses Python 3, but other programming languages should work as well.

You are welcome to use this course and adapt it to your own needs, but please give credit where credit is due.

Feedback on the course is appreciated, to fix errors and inconsistencies, to add extra information, and to improve the learning experience of the students. In particular, I am interested in other aspects of Computational Thinking for which students can be taught practical approaches.

You can send feedback to `pythonbook@spronck.net`. Please indicate that your feedback concerns the Computational Thinking book, and not the Python book.

Pieter Spronck
June 9, 2023
Tilburg, The Netherlands

Pieter Spronck is a Professor of Computer Science at Tilburg University, The Netherlands.

Contents

Preface **v**

1 Introduction **1**

1.1 Prerequisites and assumptions 2

1.2 Outline 3

2 Everyday Computational Thinking **5**

2.1 Handling a shopping list 5

2.2 Algorithms 6

2.3 Flow diagrams 6

2.4 Shopping list flow diagram 7

2.5 A second shopping list flow diagram 10

2.6 Shopping in real life 10

3 Thinking about Sorting **13**

3.1 Pseudo code 13

3.2 Real life sorting 14

3.3 Sorting a deck of cards 15

3.4 Efficiency of sorting a deck of cards in a naive way 15

3.5 Efficiency of sorting a deck of cards in a less naive way 17

3.6 More efficient sorting 17

3.7 Designing algorithms for a computer 18

4 Functions **21**

4.1 A simple function . . . 21
4.2 Contract programming . . . 22
4.3 Using functions . . . 23
4.4 Advantage of using functions . . . 24
4.5 When is a function doing too much? . . . 25
4.6 When is a function doing too little? . . . 26
4.7 What to consider . . . 27

5 Problem Decomposition **29**

5.1 Problem: Which word occurs the most? . . . 29
5.2 Problem decomposition: Flow diagram . . . 30
5.3 Selecting data structures . . . 31
5.4 Using functions for steps . . . 31
5.5 Further decomposition . . . 32
5.6 Implementing steps . . . 35
5.7 Isn't this code too convoluted? . . . 37
5.8 How to decompose a problem . . . 38

6 Top-down Development **41**

6.1 Top-down programming . . . 41
6.2 Postpone tasks via functions . . . 42
6.3 Developing a number input function . . . 44
6.4 How to implement a program top-down . . . 45

7 Bottom-up Development **47**

7.1 Bottom-up programming . . . 47
7.2 Rolling dice . . . 48
7.3 Improvements to the dice rolling program . . . 50
7.4 How to implement a program bottom-up . . . 51

8 Generalization **53**

8.1 Getting input with constraints . . . 53
8.2 Generalizing the input function . . . 55
8.3 Abstraction . . . 56

9 Implementation Tips **59**

9.1 Choosing data structures 59

9.2 Building loops over sequences 62

9.3 Building loops with an indeterminate end 63

9.4 Initializing and deinitializing 65

9.5 Capturing errors 66

10 Recursion **69**

10.1 Recursion in real life 69

10.2 Guessing a number 70

Chapter 1

Introduction

During my years of teaching programming to university students, I have found that while most students have no problems learning the language elements of programming languages, a considerable number of them have troubles designing algorithms.

A programming language is like a toolkit. It offers the programmer ways to specify commands, conditions, loops, and other constructs. It is the programmer's job to combine these constructs in different ways so that the resulting program solves a particular problem.

Programming courses that teach students a programming language usually are limited to teaching the different language elements. They introduce topics such as: what is a loop, what different ways does the language offer to write a loop, and how exactly does a loop work? The students then get to practice with these language elements to solve simple, straightforward problems. Such courses do not teach students how to solve more complex problem for which it is not immediately clear which language elements need to be used, and how and when they need to be used.

Designing an approach to solve a problem with a computer program requires, maybe surprisingly, no programming skills. It requires computational thinking skills.

What is computational thinking? I have found many definitions of the term, all different and all focusing on different aspects. In general, computational thinking encompasses the skills one needs to analyze a problem in such a way that one can design a computational solution for the problem, i.e., a solution which consists of a series of specific steps which, when executed exactly as specified, lead to a desired outcome. Once such a solution is designed, it can be turned into a program, which is when programming skills come in.

What kind of skills are we talking about? Obviously, the ability to think logically and analytically helps. The ability to see patterns and to use abstractions is highly beneficial. The ability to distinguish relevant details from irrelevant ones is crucial. Occasionally, creativity plays a role in designing computational solutions. These skills are not only necessary for creating programs which solve complex problems, but are also highly desirable when dealing with real-life problems and when working in science.

One would hope that such skills would be taught in secondary schools, but usually the opposite is true. Most secondary schools teach pupils to solve a limited set of problems in very particular ways, leading to students who are used to following predefined steps

to solve problems, rather than coming up with solution approaches on their own. A few topics in secondary schools, such as math, train pupils in computational thinking skills – however, even for these topics pupils can get a passing grade if they just blindly follow the teacher's instructions. The consequence is that many students enter university with only limited computational thinking skills.

However, my experience is that most students can pick up these skills when practicing with solving programming problems. They struggle at first, but over time, with enough practice, they get better at creating solutions. Many students have told me that there came a moment that they "saw the light," after which creating programs became much easier and more fun. I have met only very few students who never reached the stage where they had acquired the necessary computational thinking skills.

For me the question is how I teach computational thinking skills to students who still need to develop them further. Until recently, my only approach was telling students to practice: building solutions for problems with increasing complexity, asking for help when getting stuck, and studying other people's solutions after having designed a solution on one's own. I show students how I approach problems step by step, hoping that they pick up the skills that I am using by observing me.

This approach works reasonably well. Most students who have trouble creating programming solutions do possess plenty of computational thinking skills, but sometimes they lie dormant because there was little reason to use them in the past. With practice, when using these skills, over time using them becomes natural.

Regardless, I believe that the training of computational thinking skills can be more effective if it does not just rely on practicing with creating solutions. There are common concepts to computational thinking which can be made explicit and can be taught to students, to give them assistance in computational thought processes.

This course teaches those common concepts.

1.1 Prerequisites and assumptions

This course assumes that the students have basic programming skills. The programming language used in the course to create solutions is Python 3, but other programming languages will work as well. I picked Python because it is taught at many universities, and is easy to use, while still being extremely powerful. It is also really easy to read, so that those who know a different programming language but not Python, usually are still able to read and understand Python programs.

The book is mostly useful to students who have learned about the following topics: expressions, variables, functions, conditions, iterations, strings, lists, dictionaries, and text files. However, even those who have not gotten much further than iterations may find some of the discussed topics useful.

The book is meant for self study. If a student is already able to use computational thinking skills, it is of little value. The course is really aimed at students who know programming language elements but have problems coming up with solutions to programming exercises.

1.2 Outline

Each chapter briefly discusses a topic on the practical use of computational thinking, usually guided by an example. Solutions will be developed step by step, to demonstrate how the particular use of computational thinking which is discussed, is applied. Usually the solution is also developed as a Python program (Python 3.5 or higher required).

The chapters are not long. I deliberately wanted a short course. If I would hand someone a 200-page book they would probably not get into it at all. I wanted each chapter to quickly explain a simple concept which can be applied in many ways. Students should learn these concepts, and when they encounter a new problem, think which concepts apply to that problem. This may help creating solutions.

Each chapter ends with several exercises. Note that these exercises are not sufficient to practice with the topics under discussion. However, computational thinking is required for almost any exercise that you do during a programming course. So you can practice with the approaches which are explained by doing any exercise at all. For starters, the exercises in "The Coder's Apprentice," for which this course is a companion, are suitable for practicing with computational thinking, in particular the harder ones.

I wish to note that this course may not be complete. There may be many more topics that I could discuss. So I am interested in suggestions for further topics, for which I can then create new chapters.

Chapter 2

Everyday Computational Thinking

We do a lot of planning in our daily lives. To plan well, one has to use some form of computational thinking. Most people do this naturally. In this chapter I will show some examples of this, to demonstrate that, regardless how easy or how hard you think programming is, computational thinking is already a natural part of how you analyze problems.

2.1 Handling a shopping list

Suppose that you have to make dinner in the evening, and you have to go shopping for ingredients. You check out the recipe that you want to make, and write down a list of ingredients that you need to purchase. Your goal is to have all those ingredients in your home when you are done shopping. Consider the following approaches:

- *Approach 1*: You stay home and play videogames all day. When evening comes, the ingredients have mystically appeared in your fridge.
- *Approach 2*: You leave your house and wander about in a random way. If by chance you encounter one of the ingredients on the list, you take it. When dusk sets in your return home.
- *Approach 3*: You take your list of ingredients into town. There you ask everyone you see whether they can help you acquire one of the items on the list. If they can, you do what they tell you to do. When the shops close, you return home.

Using your computational thinking skills, analyzing these approaches, you will probably conclude that none of them is likely to result in your desired goal. The first one most certainly will not, and the second and third will only succeed if you are really, really lucky. They definitely do not guarantee success. In practice, you will not follow any of these approaches – you will have rejected them out of hand.

Here are some approaches that are likely to achieve your goal.

- *Approach 4*: You look at the top item on the list. You then check whether you already have this item in your home. If you do not, you go into town, go to the store that sells the item, purchase the item, and go home again. You then cross off the item. You repeat this procedure until you have no more items on the list.

- *Approach 5*: First, you go through all the items on the list, and cross off those that you have already in your home, and then go into town. Second, you look at the top item on the list, go to the store that sells the item, purchase the item, and cross it off. You repeat this second step until you have no more items on the list. Third, you go home.
- *Approach 6*: First, you go through all the items on the list, and cross off those that you have already in your home, and then go into town. Second, you look at the top item on the list, go to the store that sells the item. At that store you go through all the items on the list, and if the store has them, purchase them and cross them off. You repeat this second step until you have no more items on the list. Third, you go home.
- *Approach 7*: This approach is the same as the third approach, except that in the first step you will check whether you have crossed off all the items on the list. If you find that you did, then you are finished at that point.

Each of these approaches should in the end reach the desired goal. If you are only interested in reaching the goal, computationally speaking you have no preference for any of them. From a perspective of efficiency, however, you may have a preference.

2.2 Algorithms

An algorithm is a sequence of well-defined instructions which can be followed to reach a certain goal. A computer program is an implementation of an algorithm.

What exactly is meant with "well-defined" depends on who or what is supposed to follow the instructions of the algorithm. If, for instance, there is an algorithm for handling a shopping list with one of the approaches mentioned above, and you are the person who executes the algorithm, an instruction such as "purchase a liter of low-fat milk" might be sufficiently well-defined. An instruction such as "go into town" might not be, however, as you might wonder which town is meant, if there are certain time limits, and what vehicle you should use.

For the purpose of the discussions in this chapter, however, I will not bother too much about making instructions well-defined. Later on, when discussing algorithms that have to be turned into computer programs, we have to worry about that, but not right now.

There are different ways of expressing algorithms. A very specific one is writing computer code using a programming language. A slightly more high-level one is using so-called "pseudo-code," which is a language which looks a bit like a programming language, but is more general so that it can be translated to different programming languages. A very high-level one is using so-called "flow diagrams."

2.3 Flow diagrams

A flow diagram or "flowchart" consists of boxes connected by lines. Boxes with different shapes have a different role. The boxes contain instructions. The lines which connect the boxes together indicate the order in which the instructions must be executed. A beginning and an end to the flow diagram are specified.

Traditionally, flow diagrams were used by computer programmers to define their programs before they implemented them. As creating flow diagrams tends to be a very time-intensive process which is unwieldy for anything but really small programs, almost no one uses them nowadays, except for educational purposes. For the purpose of this course it makes sense to explain them here.

We will only use the most basic flow diagram elements, as shown in Figure 2.1

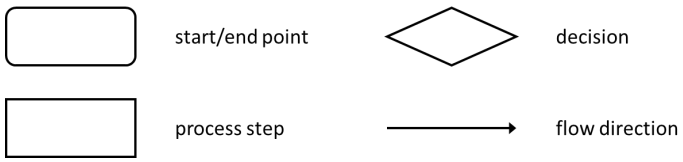

Figure 2.1: Elements of a flow diagram.

The start/end point element will bear the label "start" or "end." In general, in a flow diagram there is only one "start" point, which is where the algorithm starts. There may be multiple "end" points; the algorithm will end as soon as one of those is reached.

By following the flow direction arrows, you follow the steps of the algorithm. You move in the arrow's direction. Sometimes, when the direction is unambiguous, the arrow tip is left off (as you cannot "flow backwards" in the diagram.

A "process step" node indicates an instruction that must be followed.

A "decision" node has multiple direction lines coming out of it. In the node is a question, and each line is marked by a possible answer to that question. You follow the direction line which bears the correct answer to the question. Very often the questions have only two answers: "yes" or "no," or "true" or "false." There is nothing against having decision nodes with more than two possible answers, though, except that the flow diagram may become unreadable if there are too many direction lines coming out of one decision node.

I will now use flow diagrams to schematically represent some of the approaches to handling a shopping list discussed above.

2.4 Shopping list flow diagram

Figure 2.2 shows the flow diagram which represents Approach 4 as given above.

To make abundantly clear how a flow chart is processed, suppose that you have a shopping list that contains the following items:

- spaghetti (sold at supermarket)
- pasta sauce (sold at supermarket)
- minced meat (sold at butcher)
- cheese (sold at supermarket)

If you follow the flow diagram with this shopping list, where you already have pasta sauce in the home, you take the following steps (if you have troubles following this, then please go over the steps one by one until you understand):

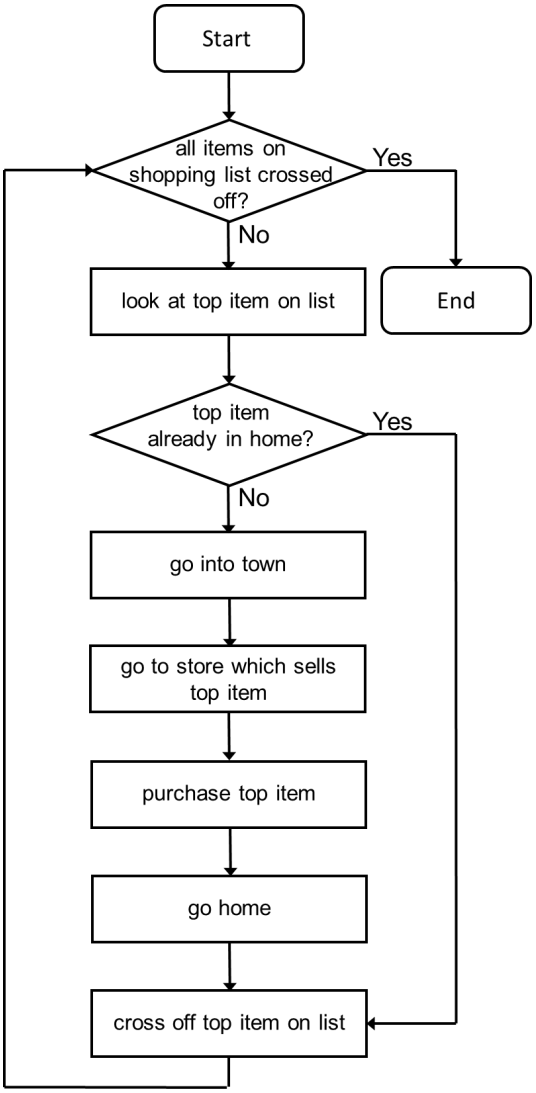

Figure 2.2: Flow diagram of Approach 4.

- Start
- all items on shopping list crossed off?: **No**
- look at top item on list: **spaghetti**
- **spaghetti** already in home?: **No**
- go into town
- go to store which sells **spaghetti**: **supermarket**
- purchase **spaghetti**
- go home
- cross off **spaghetti** on list
- all items on shopping list crossed off?: **No**

- look at top item on list: **pasta sauce**
- **pasta sauce** already in home?: **Yes**
- cross off **pasta sauce** on list
- all items on shopping list crossed off?: **No**
- look at top item on list: **minced meat**
- **minced meat** already in home?: **No**
- go into town
- go to store which sells **minced meat**: **butcher**
- purchase **minced meat**
- go home
- cross off **minced meat** on list
- all items on shopping list crossed off?: **No**
- look at top item on list: **cheese**
- **cheese** already in home?: **No**
- go into town
- go to store which sells **cheese**: **supermarket**
- purchase **cheese**
- go home
- cross off **cheese** on list
- all items on shopping list crossed off?: **Yes**
- End

A few remarks on things you may have noticed when examining this run-through of the flow diagram:

- While the first step described in Approach 4 is "look at the top item of the list," the first step in the flow diagram is the question whether all items on the shopping list are crossed off, while only the second step is "look at top item on list." The reason I have chosen to draw the flow diagram as I did, is that if by chance all items of the list are already crossed off when you use the flow diagram, there is no top item to look at, and "look at top item on list" would result in an error. This change to Approach 4 is an example of what you may need to do when turning a procedure into an algorithm. An algorithm is supposed to "work" under any circumstance, in this case, even if the shopping list is empty.
- I filled in "top item" and "store" with specific values. For "top item" I substituted "spaghetti," "pasta sauce," "minced meat," and "cheese," and for store I substituted "supermarket" and "butcher." This is an example of using "variables" in a flow diagram. The instructions in the diagram may refer to certain elements which may take on different values at different moments.

- While the flow diagram contains "loops" (arrows pointing back at earlier points of the diagram) and "conditions" (decision points where the diagram splits based on a question), when the flow diagram gets executed it turns into a sequence of instructions. When you examine the list of instructions that the flow diagram turns into, it is very clear that this list is inefficient, as you travel to town and back home no less than three times.

2.5 A second shopping list flow diagram

Figure 2.3 shows the flow diagram which represents Approach 5 as given above.

You may notice that this flow diagram consists of two parts. In the first part, the list is checked and items which you already have in the home are crossed off. In the second part, you go into town, purchase everything that you need to purchase, and go home again. This approach seems to be more efficient than Approach 4, and indeed, if you write out the sequence of steps for the shopping list used to check the previous flow diagram, you will notice that you go into town only once, rather than three times.

You may also notice that Approach 5 is still not as efficient as it could be. You visit the supermarket twice in this approach, and in between the visits you go to the butcher. Even if you would reorder the shopping list to put the minced meat at the top or bottom, the instruction "go to store which sells top item" would be encountered twice for the supermarket. Approach 6 attempts to solve this issue.

Moreover, Approach 7 introduces yet another optimization which was not demonstrated with the example shopping list used. Namely, if you have all items on the list already in your home, then there is no need to go into town altogether, so Approach 7 makes sure that exceptional situation is handled as well.

2.6 Shopping in real life

There is no doubt in my mind that if you go shopping for ingredients of a recipe in real life, you will use a pretty optimized series of steps that bring the ingredients into your home. You will not play video games the whole day expecting that the ingredients just appear, nor will you use a random process in the hopes that it will lead to the desired result. You will not go shopping if you already have all the ingredients in your home. You will not travel into town multiple times if you can avoid it, and you will try to do all your shopping in one store before you go to the next.

This means that in real life, you are quite capable of designing an algorithm for the activity of "going shopping," even if you have not written out the algorithm in a flow diagram.

You can argue that you do this automatically – you never spend any time on "designing a shopping algorithm," you just go shopping and it turns out that you do it in a relatively efficient way. This may be true, but you are still using an algorithm. Moreover, you are able to recognize inefficiencies in your algorithms and correct them. You adapt to new situations. If you drive your car into town and you find that the road you always use is inaccessible, you find a way around it.

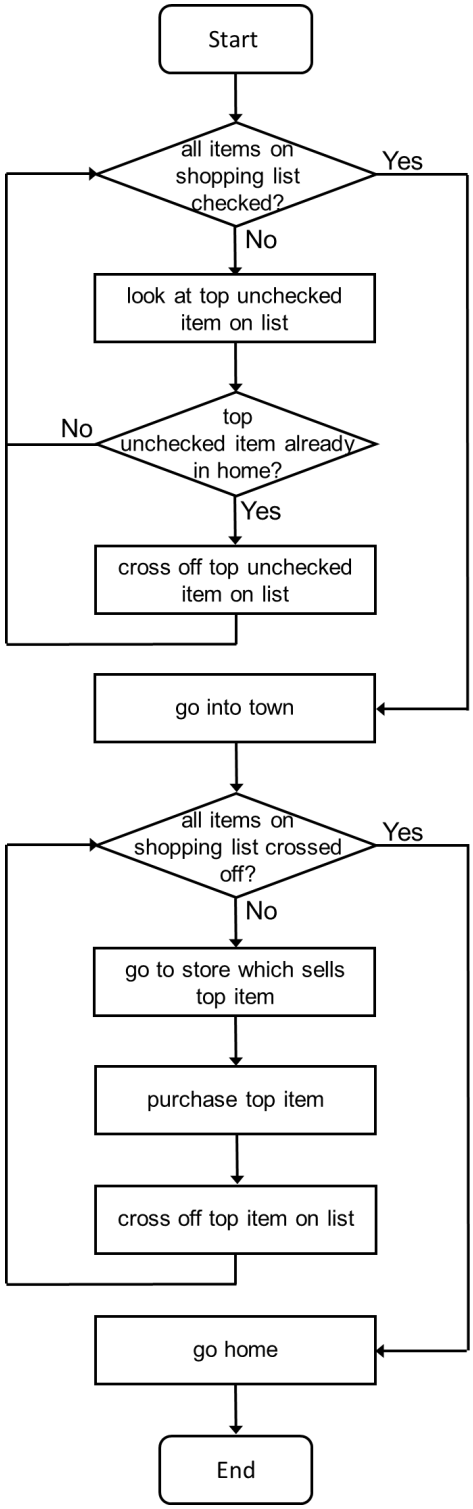

Figure 2.3: Flow diagram of Approach 5.

Exercises

Exercise 2.1 Create a flow diagram for Approach 6 and/or 7 which are listed at the top of this chapter.

Exercise 2.2 Run through the flow diagram you created in Exercise 2.1 with the example shopping list given in the chapter, with a shopping list with only items you have in the home, and with a shopping list that is empty.

Exercise 2.3 Design an algorithm for doing the dishes. You may keep it fairly high level. Include in the algorithm both the possibility that you have a dish washer and that you have to do the dishes by hand. Create a flow diagram.

Chapter 3

Thinking about Sorting

The point I wanted to make in the previous chapter is that you are able to think computationally, to design algorithms, because you do it all the time. However, as a human you have the ability to adapt your algorithms on the fly if they are faulty. That is called "improvisation," which, by the way, is also a form of computational thinking. Because of your ability to improvise, there is no need for you to really think carefully about the algorithms you use in your daily life. This is different for computers.

An algorithm for a computer must be precise. It must be complete. It must be able to deal with any unforeseen circumstance, otherwise the algorithm will not reach its legitimate end and thus will not produce the desired result.

Programming languages have been created to allow you to specify algorithms in a way that a computer can understand them. It is up to you, the programmer, to ensure that the algorithms that you present the computer with are indeed able to achieve the result that you want them to reach.

In this chapter I will discuss the everyday task of sorting things, which is something that you occasionally need to do and for which you usually think a bit on how to do it most efficiently. This kind of thinking is also needed when writing computer programs. I will start, however, with a discussion of pseudo code, which can be an intermediate step between thinking about an algorithm and implementing it into code.

3.1 Pseudo code

This course assumes that you can code in Python (even if you are not strong at designing algorithms with Python). So you can write program code. Writing pseudo code is a lot like writing program code, except that in pseudo code you can take "shortcuts." Your code does not need to be able to run. It just must give a good, preferably unambiguous insight in what a program should look like. A programmer should be able to turn the pseudo code into a program in any computer language that would be suitable for implementing the algorithm.

Often, in pseudo code you will be explicit about certain code elements which you would not need to be explicit about in your chosen programming language. For instance, in

Python, if I want to create a variable, I just introduce the variable name and assign an initial value to it, e.g., `num = 0` creates a variable `num` which is a numerical variable, as a numerical value was assigned to it. In other programming languages variables have to be declared with a specific type before you can assign them a value. Therefore, someone who knows C++ but not Python might protest when they see a statement in pseudo code that just says `num = 0`, because in the eyes of this person the algorithm does not know that `num` is supposed to be an integer. Therefore, in pseudo code you would probably write `int num = 0`, to make explicit that `num` is a variable of the integer type, which can be used from the moment it has been declared.

Another thing that you will often see in pseudo code is that code "blocks" are started with a curly starting bracket and closed with a curly closing bracket. The indentation of a code block would still be like it is in Python (as that is the most readable), but the curly brackets are there to make explicit that the statements within belong to a code block, as most programming languages are not as picky as Python about indentation.

Finally, in pseudo code statements are often ended with a semi-colon, as that is common in most programming languages.

The advantage of writing pseudo code is that you are allowed to be fairly "loose" in what you write. For instance, if in a statement you want to sort a list of persons with the name `personlist` by last name, followed by first name, you can just write `sort personlist by lastname, firstname` without being more specific on how you actually implement that.

While the shopping list examples of the previous chapter are probably not suitable for pseudo code, considering that you will not implement them in actually program code, I could write a program in pseudo code for the flow diagrams that I gave. For instance, for the first flow diagram (Figure 2.2), the pseudo code is:

```
while not all items on the list are crossed off
{
    item topitem = top item on the list;
    if not topitem is in the home
    {
        go into town;
        go to store which sells topitem;
        purchase topitem;
        go home;
    }
    cross off topitem on the list;
}
```

If it is not clear to you that this pseudo code is equivalent to the flow diagram in Figure 2.2, please study it until it is clear.

3.2 Real life sorting

I will now give an example of a real-life task for which you will actually design an algorithm when you encounter it. Suppose that you have several boxes of books, and an empty bookcase. You are going to place the books on the shelves of the bookcase, and want to have them on the shelves in a particular order, for instance alphabetically. A situation like

this may occur when you are moving houses. Ordering books on shelves is not something that you do every day, so you may not have an algorithm for it ready that you apply without thinking. So, how will you do this?

Suppose that you want to order the books on the shelves by the last name of the author. A straightforward approach is that you begin by placing the first book on the top-left position of the top shelf, and every next book that you take you insert in its correct spot between the books that you have already placed. While this works, you will be constantly shifting books on shelves, as shelves fill up and you have to make room for new books.

A slightly more complex approach is that you assign each shelf a range of letters, which are the first letters of the author names. For instance, the top shelf A–D ("Aafjes"–"Dyson"), the second shelf E–H, the third shelf I–L, etc. You then do the same as in the first approach, except that you place the books on the shelf where the author belongs. This will result in a lot less shifting of books. The disadvantage of the approach is that you probably are going to end up with some shelves that do not fill completely, while other shelves get overly full and you will have to reassign letters during the process.

A third approach, if you have the space, is that you first sort the books in piles by the first letter of the author's last name. Thus you end up with about 26 piles of books on the floor. You then handle each pile separately, starting with the pile of authors whose name starts with an A. If a pile seems to be too big to handle in one go, you may repeat the process for that pile, for instance by splitting it into two or more smaller piles based on the second letter of the author's last name.

3.3 Sorting a deck of cards

Take a deck of cards, and shuffle it (yes, this is an exercise, so please do it). Now sort the deck as follows: on top of the deck will be all the Diamonds cards, ordered from 2 (lowest) to Ace (highest). Below that will be the Clubs, again ordered from 2 to Ace. Below that the Hearts, and at the bottom the Spades, again, both ordered from 2 to Ace. If there are any Jokers, put them on the bottom. Once you have finished doing it, think about which algorithm you used.

It is more than likely that you first made four stacks of cards, each for one of the suits, and then sorted each of the stacks in your hand, as you can probably hold 13 cards in one hand. Alternatively, to sort a stack you may just have spread it out on the table, picked out the Ace and put it aside, then picked out the King and put it on top of the Ace, then picked up the Queen, etc.

The point is that you probably would not have tried to sort all the cards at once, holding 52 (or more, with Jokers) in your hands while trying to move cards around until they are in the right order. You would not have done it like that because you intuitively know that it makes for an unwieldy and inefficient process.

3.4 Efficiency of sorting a deck of cards in a naive way

To estimate how efficient a particular method of sorting a deck of cards is, we can count how often you have to look at a particular card. Note: determining the efficiency of an

algorithm means that you determine its "time complexity" (if it is about the speed of the algorithm). I will not discuss the formal approach to determine time complexity here, but will simply use the loose definition of "counting looks at cards," which suffices for the purposes of this chapter.

One way of sorting the deck would be to simply search the whole deck from top to bottom to find the first card, which would be the 2 of Diamonds. Once you have found the card, you put it aside, and search the deck for the next card, which would be the 3 of Diamonds. Once you have found it, you put it on top (or bottom, depending on how you are working) of the 2 of Diamonds. Then you search for the 4 of Diamonds, then the 5 of Diamonds, etc. You do this until you have sorted the deck.

Assuming that the deck contains 52 cards (no Jokers), how many cards do you need to look at before you have sorted the deck using this algorithm? That depends on how the deck is shuffled. If, after shuffling, the deck was already sorted, you only need to look at 52 cards, because every time you search for a card, it will be the first one you look at. This is the "best-case" scenario.

What is the "worst-case" scenario? That is that the deck is sorted in reverse order. Using the given procedure, you will have to look at 52 cards to find the 2 of Diamonds, at 51 cards to find the 3 of Diamonds, at 50 cards to find the 4 of Diamonds, etc. In total you will look at 1378 cards.

In practice, you probably will look at less cards, because you will have found the card that you are looking for earlier than that. On average, the card you search for will be halfway through the remaining deck, so you can probably make do with about half the number of looks than in the worst-case scenario, so around 689 looks.

In computer science, the approach to sorting a deck of cards as described here is called "selection sort." It is known as a sorting algorithm that is less efficient than sorting can be, though it works fine if the number of items that need to be sorted is relatively small.

Note that in selection sort you will always be looking for the "lowest" item in the remainder of the set of items you are sorting, without knowing what that lowest item actually is. For sorting the deck of cards, we knew which was the next card that we were looking for, and thus could stop searching when we found that card. However, if some cards would be missing from the deck, a sorting algorithm which would search for a specific card would not give the desired result. So it may be preferable to search for the "lowest" of the remaining cards rather than a specific card, but that would mean that for each pass through the deck, you would have to look at every card, and this would need the worst-case number of looks.

A pseudo code description of selection sort is:

```
# selection sort algorithm which sorts a list of items called "items"
list sorteditems = empty set;
while not list items is empty
{
    item lowestitem = the lowest item in list items;
    items.remove(lowestitem);
    sorteditems.append(lowestitem);
}
```

If you want to turn this pseudo code into actual Python code, you should be able to go through it step by step and replace each line with legitimate Python. If you think a line

cannot be implemented by a single line of Python, you can replace it with a function call, and implement that function later (more on this in the next chapter). However, the pseudo code above can be turned line by line into actual Python code. Try it, before you examine the solution below.

```
items = [4,7,17,12,0,-3,2,17,5,6,4,1,0,-6,8,23,11,10]

# Here follows a translation to Python of the pseudo code given
    above.
sorteditems = []
while len( items ) > 0:
    lowestitem = min( items )
    items.remove( lowestitem )
    sorteditems.append( lowestitem )

print( sorteditems )
```

3.5 Efficiency of sorting a deck of cards in a less naive way

Let's now look at the sorting of a deck of cards in the way I described above, where you first split the deck into four piles, each of one suit, and then sort those piles before putting them together.

To create the piles with the suits, you have to look at each card once, which is 52 looks.

Now you have to find an approach to sort each of the piles. Let's use the same naive approach described above, which feels a lot less unwieldy for 13 cards than if you use it for the whole deck. In the worst-case scenario, you find the first card with 13 looks, the second card with 12 looks, the third card with 11 looks, etc. In total, in the worst-case scenario, you need 91 looks to sort a suit. On average it will be about 45.5 looks. For 4 suits, that will be a total of 182 looks.

This means that, using the less naive approach, you need 52 + 182 = 234 looks. This is a considerable improvement over the 689 looks that we need for the naive approach. This means that, if you indeed used an approach in which you first sorted out the suits and then sorted the cards for each suit, you designed an algorithm that is considerably more efficient than if you naively went through the deck separately for each card.

In computer science, this approach is called "bucket sort" for the creation of the four piles, followed by "selection sort" for each of the piles.

3.6 More efficient sorting

There are sorting methods which are more efficient than the two described above. Yet in practice you will not use them when you have to sort a deck of cards. The reason is that the more efficient algorithms split the deck of cards in more complex ways; to give an example: an algorithm could, for instance, instruct you to split the deck into three piles, with pile 1 containing all Diamonds cards and the Clubs up to the 8 of Clubs, pile 2 contains

the remaining Clubs cards and the Hearts cards up to the Jack of Hearts, and pile 3 all the remaining cards. For a human, making such a split needs more concentration than just splitting the deck into piles which contain different suits.

For a computer such distinctions need not matter. So when you design an algorithm that a computer should execute, you sometimes can make the algorithm "better" than an algorithm that you would have to execute manually, by taking into account that many things which are hard or non-intuitive for a human are actually easy for a computer.

3.7 Designing algorithms for a computer

While I argued above that a computer may run algorithms which would be hard to use for a human, usually when you need to design an algorithm for a computer it makes sense to consider the question: "how would I solve this problem if I had to do it manually?" While pondering this question, you should not reject approaches because they would be too boring or too time-intensive. Since the algorithm you come up with would be used by a computer, it does not matter if it is boring or time-intensive, because we have computers to do the boring and time-intensive things for us. As long as the steps of the algorithm you design lead to the desired outcome, in principle you have found something that is worth your while to try out in code.

My experience is that for many students the hint "think about how you would do this task if you had to do it manually" is insufficient to get them to design an algorithm. Therefore, in the next chapters I will describe approaches that you can use to come up with simple algorithms. My goal is to give you some handholds which you can use when you need to write code for a problem that you see for the first time.

Exercises

Exercise 3.1 In the previous chapter you created a flow diagram for doing the dishes (Exercise 2.3). Turn this flow diagram into pseudo code.

Exercise 3.2 Take two decks of cards with different-colored card backs (or, if you do not have those ready, imagine that you do). Shuffle the decks together. Now sort them again, as described in the chapter (Diamonds, Clubs, Hearts, Spades, and within each suit ordered from 2 to Ace), with each of the two card backs in their own pile. What algorithm do you use?

Exercise 3.3 In the previous exercise you probably first distributed the cards over two piles, by card back. Can you think of an algorithm of about similar efficiency that uses a different approach?

Exercise 3.4 Can you think of a different approach to sorting a deck of cards than the naive and slightly less naive ways discussed in the chapter? Can you estimate the efficiency of this approach?

Exercise 3.5 Design an algorithm to play tic-tac-toe. You do not have to play it attempting to win, you just have to play legal moves. This is easiest if you use a tic-tac-toe diagram with 9 numbered fields to play on. At least describe the algorithm in English, but if you can, turn it into a flow diagram or pseudo code. Or, for an extra challenge, real Python code.

Exercise 3.6 Consider the following approach to sort a deck of cards. You start by taking a random card from the deck, for instance, the first one. Then you go through the deck, putting all cards that need to be higher in the deck than the card you selected on top of the selected card, and putting all the other cards on the bottom of it. You can put the selected card sideways, so that you can see where it is. Once you have done this, the selected card is in the right spot in the deck, though all the other cards may not yet be. Now you repeat the procedure for the pile on top of the selected card, and the pile on the bottom of the selected card. You continue repeating the procedure for smaller and smaller piles until the deck is sorted. In computer science, this is called "quicksort." The card that you select to split each of the piles is called the "pivot" for this algorithm.

1. If you do not understand how the described algorithm works, then try it out for a single suit of 13 cards. Things will soon fall into place.
2. Assuming that the pivot always splits a pile of cards in two piles of about equal size, can you make an estimate of the efficiency of this approach to sorting a deck of cards?
3. If you compare the efficiency of this approach with the slightly less naive approach discussed in the chapter, you will probably find that it seems to be a bit less efficient than that approach. However, suppose that a deck of cards consists of 4 suits of 25 cards each. Would the “quicksort” approach still be less efficient?
4. Why would or wouldn’t you use the quicksort approach in real life?

Chapter 4

Functions

A function is a programming construct that may or may not receive some arguments, performs a task (in which it uses arguments that it received), and may or may not (but usually will) deliver one or more outcome values. In Python, a function is defined with the keyword **`def`**. Since this course assumes that you know the basics of Python, it assumes that you know about functions. However, you may not yet realize how and why you should use functions.

4.1 A simple function

Below I give a function which converts a temperature given in degrees Kelvin to degrees Celsius. The function, which is called `kelvin2celsius`, gets one parameter (or argument), namely `kelvin`, which is a temperature given in degrees Kelvin. It returns a floating-point value which is the corresponding temperature in degrees Celsius.

```
# kelvin2celsius converts a temperature in degrees Kelvin (given
# to the function as a numerical argument "kelvin" which must be
# at least zero) to a temperature in degrees Celsius, which it
# returns as a floating-point value.
def kelvin2celsius( kelvin ):
    return kelvin - 273.15

temp = 100
print( f"{temp} degrees Kelvin is {kelvin2celsius( temp ):.2f}
   degrees Celsius" )
```

When you examine the function, you see that it is pretty simple: it merely subtracts 273.15 from the argument given to the function, and returns the resulting value. Maybe you knew how simple this conversion is. Maybe you did not. One of the big advantages of having functions is that you do not need to know how to do something, as long as you know how to use a function that does it for you.

For the function `kelvin2celsius()` you need to know:

- that you can give it a temperature in degrees Kelvin as an integer or floating-point value
- that it will calculate the corresponding temperature in degrees Celsius
- that it will return the calculated temperature as a floating-point value

There are two more things that you should also know, or at least realize:

First, the function will not have any side effects, i.e., by calling the function you will not influence anything else in the program. Caveat: The function may influence the argument or arguments given to it, if they are references (pointers). However, this then should be made clear in the function description. If you do not understand references, you can forget about them for this course, but they are an important advanced topic of a programming course.

Second, the return value of the function is undefined if the argument given to it is not a legal value. For instance, if you call the function with -100 as argument, it will return -373.15; however, as zero degrees Kelvin is absolute zero, there are no lower temperatures, and thus the return value is not a legal temperature. Moreover, if you call the function with, for instance, a string, it will simply crash your program. The function could, of course, check whether the argument given is a legal value, and handle it in a graceful way (for instance, in the function above you could say that if an illegal temperature is given as argument, you will return -10000). But in principle it has no responsibility to do so, unless the function description says that it does.

4.2 Contract programming

"Contract programming" is a programming paradigm in which software designers write a precise definition of software components (such as functions) not only with respect to what they do, but specifically preconditions, postconditions, and invariants. Such a definition is called a "contract." I will not go into the exact technical meaning of what contract programming is and requires, but I will use the contract metaphor to discuss how one should think about functions.

A precondition is a description of the program's situation before a call to a function is made. In principle, a function can only know about this situation what is communicated to it by the arguments given to it. While it is true that in Python a function can also access global variables that exist outside the function, it is highly preferable that it does not do that. Because if it does not, when you call a function you know exactly what the function gets to work with: the arguments that you give to it, and nothing else. This makes the function independent from the program it resides in.

From the perspective of the function, the precondition entails that it gets arguments that it can use. In the function description it must be specified what arguments the function needs, such as with respect to the data types and the value ranges. The contract only defines what the function will do if the arguments are "legal." If you give the function anything else, what the function does is not specified.

The postcondition of a function can be interpreted as its return value or values, and, if applicable, the changes it has made to the data structures which you have handed to it. For example, the `kelvin2celsius()` function above has as its postcondition that it presents

you with the temperature in degrees Celsius which is equivalent to the temperature in degrees Kelvin given to the function. It does not make changes to a data structure. For an example of a function that changes a data structure: think about a function that you can give a list of numbers, which will then sort that list. Once the function has finished, the original list has been changed: it is now sorted.

Finally, for functions the main invariant should be that no data within reach of the program is changed, with the exception of data that the function is given access to.

In actuality, invariants are also meant to be about certain relations within the data. For instance, suppose that you are working with a program that is used to collect orders, such as you see on websites where you can buy stuff online. You have a list of items that you are going to order, and you also see the total cost of these items. Suppose now that there is a function that you can use to add an item to the list. An invariant would be that the sum of the prices of the items on the list is equal to the total cost shown, which entails that the function which adds the item also ensures that the total cost is updated when the function ends. When you write functions for a fairly complex programs which may influence relations between data items in the program, it makes sense to consider invariants about those relations as well.

4.3 Using functions

There are two main ways in which you can use functions to make thinking about algorithms easier.

The first is that when you think about an algorithm, you may realize that you need to do several things, each of which requires some non-trivial code. Taking all that code together might feel overwhelming. However, if you can split the whole algorithm in multiple function calls, it may become easier and more natural. Since this use of functions ties in with Problem Decomposition, I will leave it for the next chapter, which is all about that.

The second is that you can use a function to encapsulate a functionality that you know you need to use, but can extract from the rest of your program in a way that you can solve it separately. This may sound a bit abstract, so let me make this clear with an example.

When you do text processing, you often have to examine individual words in the text. For instance, you may wish to count how often a particular word occurs in the text. Now, to get access to the individual words, you need to split the text into those words. Usually a programming language has built-in functionalities for that; for instance, Python has the string method `split()`, which splits a string into individual words. However, suppose that I use that `split()` method on a particular string using the following code:

```
drseuss = "You know you're in love when you can't fall asleep
   because reality is finally better than your dreams."
drseusslist = drseuss.split()
print( drseusslist )
print( f"Number of times \"you\" occurs in this list: {drseusslist
   .count('you')}" )
```

How often does the word “you” occur in the given string? The answer is three times, but each of these times, it appears differently in the list of words produced by `split()`. You

get one “You”, one “you’re”, and one “you”. You want the word “you” to be recognized in each of these cases. A simple solution is to turn the string into lowercase characters, and replace all characters in the string which are not letters with spaces, before you apply `split()`. You may say that you will “clean” the string before processing it.

This “cleaning” of the string can be considered a separate activity that you need to undertake. You can imagine that you implement this activity in a function. That function would have as precondition that it receives a string, and as postcondition that it returns a “cleaned” version of the string. Naturally, the function should not affect the rest of the program.

Maybe you need to do it only once, maybe you need it multiple times. If you need it multiple times, it definitely makes sense to place it in a function, so that you only need to have it once in your program. But even if you only need it once, turning a separate functionality into a function makes a program more readable and easier to understand and maintain.

Here is an implementation of the `stringclean()` function:

```
def stringclean( inputstring ):
    outputstring = ""
    for letter in inputstring.lower():
        if letter >= 'a' and letter <= 'z':
            outputstring += letter
        else:
            outputstring += " "
    return outputstring

drseuss = "You know you're in love when you can't fall asleep
   because reality is finally better than your dreams."
drseusslist = stringclean( drseuss ).split()
print( drseusslist )
print( f"Number of times \"you\" occurs in this list: {drseusslist
    .count('you')}" )
```

4.4 Advantage of using functions

This is the big advantage of using functions that you should learn from the example above:

Creating a function for a certain functionality allows you to concentrate on a small part of the problem that you have to solve with a program. As long as you know what input the functionality needs (in the “cleaning” example: a string – and nothing else), and you know the output that the functionality needs to produce (in the “cleaning” example: a cleaned version of the input string), you can concentrate on developing the functionality disregarding all the other things that your program needs to do.

Assuming that you have sufficient knowledge of the programming language that you need to use, usually writing a function is a relatively short and easy task. The main issue is that you need to recognize for what functionality it makes sense to put it in a function. In general, a function should implement a functionality that is relatively small, that can be

extracted from the rest of the problem that you need to solve, and that is not doing "too much."

4.5 When is a function doing too much?

The remark that a function should not do "too much" needs further exploration.

Let's go back to the "cleaning" example. Would it make sense to develop a function which counts how often the word "you" occurs in a string? Here is an implementation of such a function:

```
def stringcountyou( inputstring ):
    outputstring = ""
    for letter in inputstring.lower():
        if letter >= 'a' and letter <= 'z':
            outputstring += letter
        else:
            outputstring += " "
    return outputstring.split().count("you")

drseuss = "You know you're in love when you can't fall asleep
   because reality is finally better than your dreams."
print( f"Number of times \"you\" occurs in this string: {
   stringcountyou(drseuss)}" )
```

Think about why this function `stringcountyou()` is doing "too much."

A main reason why it is doing too much, is that if you also want to count how often the word "love" occurs in the string, you would have to develop a separate function for that, which would be the same as the first function except that in the last statement of the function would use the word "love" instead of "you". Obviously that is not a good idea, as you would need a piece of code very similar to the function given for every different word you want to count. The function is too specific.

Now, what if I would make the function a bit more general? Instead of letting it count "you", I will let it count any word, by giving the word that it needs to count as parameter. Like this:

```
def stringcountword( inputstring, word ):
    outputstring = ""
    for letter in inputstring.lower():
        if letter >= 'a' and letter <= 'z':
            outputstring += letter
        else:
            outputstring += " "
    return outputstring.split().count(word)

drseuss = "You know you're in love when you can't fall asleep
   because reality is finally better than your dreams."
for word in ['you','love']:
```

```
    print( f"Number of times \"{word}\" occurs in this string:  {
   stringcountword(drseuss,word)}" )
```

This definitely improves the function. However, is it still doing "too much"?

There is at least one reason why you may think that this function is still doing too much, and that is that it will clean the original string every time that you want to count how often a particular word occurs in the string. In practice, you would only need to clean the string once, and use the cleaned string every time you want to count a word. In fact, the function is not only cleaning the string but also splitting it, and that splitting you are now doing for every word as well, while it would suffice to do it only once.

A final reason why I would assess this function as doing too much, is that cleaning a string is a functionality that may have multiple uses. You can use it to count words, but you can, for instance, also use it to check how many words in the sentence occur in a dictionary. Or you can use it when you want to find out how long the longest word in a text is. If you combine the cleaning of the string with the use of the cleaned string (which is what the function `stringcountword()` does), then you make your functionality less generally useable.

This reason might not hold for a program that only needs to count how often a word occurs. However, an experienced programmer would probably still consider that later the program may need changing and in that case it might be discovered that the cleaning code should have been isolated from the word counting code.

4.6 When is a function doing too little?

In principle, a function can never do "too little."

You may wonder whether it makes sense to develop a function which consists of only a single line of code. My answer would be: if one line of code is all it needs, why not?

Take, for instance, the `kelvin2celsius()` function at the top of this chapter. It consists of one line of code, which subtracts 273.15 from the input value. If you need to do this conversion in a program, why not just do this subtraction in the main code rather than call a function for it? There are multiple reasons why you should have a function for this:

- It makes the code self-documenting. If I see a call to a function `kelvin2celsius()`, I immediately understand what the code is doing. If instead I see 273.15 being subtracted from another number, I might not understand what it does. You can, of course, write a comment next to the subtraction, but I personally rather have code for which the comments are not necessary for understanding.
- It removes the necessity for me to understand how you convert Kelvin to Celsius. The function call turns this calculation into a "black box." There are other temperature conversions which are more complex, such as converting Fahrenheit to Celsius. If I simply call functions for these conversions, the details of the conversion can be considered irrelevant for the understanding of the main program.
- It allows me to easily expand the functionality without affecting the main program. For instance, suppose that I have a Python program that needs to convert Kelvin to

Celsius, and in the code I do that by simply subtracting 273.15 from the Kelvin value, as I assume that the value will always be legal. At some point it is discovered that illegal values are actually possible, and should be captured using exceptions (if you do not know what exceptions are, don't worry, you don't need to know that for this example). If I do the conversion in a function, I only need to adapt that function. If I do not have a function for the conversion, I have to search the whole program for places where 273.15 is subtracted from a value and capture exceptions there.

4.7 What to consider

In summary, when you examine a problem and strike upon a functionality that you may want to lift out of the problem and develop a separate function for, consider the following:

- Is the functionality clearly defined? What exactly does it need to do?
- Precondition: what does the functionality need as input? Make sure that you can give it everything it needs.
- What should happen if the requirements of the precondition are not met? It is okay if the functionality then produces nonsense or crashes the program, but you should at least consider whether you can handle erroneous inputs gracefully.
- Postcondition: what does the functionality produce? This will define the return values.
- Will the functionality have side effects with respect to the program it is part of? In general, it should not, but if it does this should be documented.
- Is the functionality doing "too much"? If you can recognize multiple steps in the functionality, some of which would have a general purpose beyond the functionality, you may have situation wherein it makes sense to split the functionality up further.

Exercises

Exercise 4.1 Create a function `celsius2kelvin()` which converts a temperature given in degrees Celsius (a floating-point number) into degrees Kelvin.

Exercise 4.2 If F is a temperature in degrees Fahrenheit, and C is a temperature in degrees Celsius, then $C = (5 * (F - 32)/9)$. Develop a function `fahrenheit2celsius()` which converts a temperature in degrees Fahrenheit to degrees Celsius, and a function `celsius2fahrenheit()` which does it the other way around.

Exercise 4.3 Develop functions `kelvin2fahrenheit()` and `fahrenheit2kelvin()` which convert between degrees Fahrenheit and degrees Kelvin. Rather than developing these functions from the ground up, use calls to (some of) the functions from the previous exercises.

Exercise 4.4 You have to develop a program that gets a temperature as input, and shows what this temperature is in Kelvin, Celsius, and Fahrenheit. The input temperature is given

as a number and a letter. The number (which you may assume is an integer) is the temperature value, and the letter is a `C`, `F`, or `K`, which indicates that the number represents a temperature in Celsius, Fahrenheit, or Kelvin respectively. For instance, if the input is "100C" that represents 100 degrees Celsius. The number can be negative.

You may use the functions developed in previous exercises. Which other function or functions would be useful to have? Develop the program.

Chapter 5

Problem Decomposition

The core skill that you must acquire to successfully design and implement algorithms is "problem decomposition." This entails that you are able to divide a problem into a collection of smaller, manageable problems, each of which you can solve separately, which together solve the problem as a whole.

5.1 Problem: Which word occurs the most?

Consider the following task: you have a text file, and you have to determine which word in the text file occurs the most. How will you accomplish this task?

This is a very straightforward problem which you may see on an exam. On the exam, you usually get some code which you have to complete, which could look as follows:

```
# Function get_word_with_highest_count() gets the name of a text
# file as input, and should return the word that occurs most
# often in this file.
def get_word_with_highest_count( filename ):
    # Write your code here.
    return "<word with highest count>"

print( get_word_with_highest_count( "pc_woodchuck.txt" ) )
```

An experienced programmer will hardly think about this problem, but will code a solution in a few minutes. The skills which such a programmer uses come with years and years of writing programs, and are not natural to most people who start out designing and implementing more complex programs. An inexperienced programmer will have to approach the problem systematically.

If you get such a task as an inexperienced programmer, what you should definitely not do is start typing code. What you should do is think about the problem and devise a general solution. Because the problem as a whole might seem too daunting a task, what you should do is decompose the problem into smaller problems.

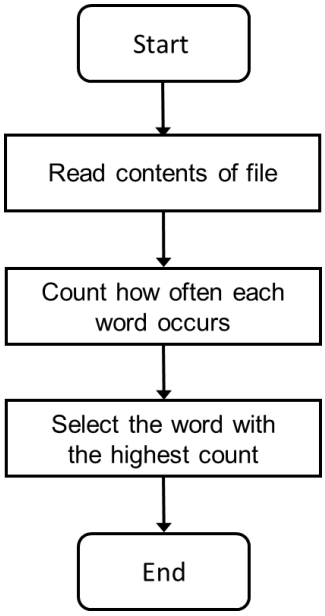

Figure 5.1: Flow diagram of finding the word that occurs the most.

5.2 Problem decomposition: Flow diagram

Which smaller problems do you have to solve to accomplish the task? Obviously, you will have to read the contents of the file. You will have to count how often each word in the file occurs. And then you have to select the word that occurs the most. This can be represented by the flow diagram shown in Figure 5.1.

We may consider each of the steps in this flow diagram a function. A function has a precondition/input (what goes into the function) and a postcondition/output (what does the function produce). Usually, the product of one step is what goes into the next step. So what are the inputs and outputs of each of these steps?

For the first step, "Read contents of file," the input is the name of the file. In the code supplied for the problem, the file name is indeed given, so it is available. What the step should produce is "the contents of the file." The step is not explicit about how these contents are represented, but you probably know that if you read the contents of a file, you either get a string or a list of strings.

Regardless how you represent the contents of the file, they are the input for the next step, "Count how often each word occurs." What this step should produce is some data structure which contains all the words that are in the contents of the file, with for each word how often it occurs.

Again, it is not clear yet how you represent the words with their counts, but they are the input for the last step, "Select the word with the highest count." The output of this step is a single word, namely the word which occurs the most, which you indeed can select if you know all the words in the file contents with their counts.

5.3 Selecting data structures

As indicated, you need two data structures for the steps which are given in the flow diagram in Figure 5.1. One data structure needs to represent the contents of the file, and a second data structure needs to represent all the words with their counts.

You know that when you read a file, you can either get a string or a list of strings. Which of these two would be most appropriate for the present problem? A list of strings is mainly useful if you need to do something with the separate lines in the file, but the lines are not important for the current problem. So a string suffices. So, let's decide to represent the contents of the file as a string.

How to represent words with their counts? Since we have to store multiple words – we do not know yet how many –, we probably need a data structure which has some repetition in it. The three basic data structures in Python which have this are the list, the dictionary, and the set. Since we have to store a count with each word, we have the following main possibilities:

- a list of tuples (word,count)
- a dictionary with the word as key and the count as value
- a set of tuples (word,count)

With some experience you may realize that the dictionary is probably the best choice, as you will have to go over all the words in the file contents, and update the count for a word every time you encounter it. Searching for a word in a list is slow (unless the list has a structure which allows fast searching), while for a dictionary it is fast.

A set of tuples will not work, but you have to know a bit about sets to realize why not. Suppose that I have in my set stored that the word "you" occurs twice; that would entail that according to my description, `("you",2)` is an element of the set. I find the word "you" once more in the file contents, so I have to remove the item `("you",2)` from the set, and add the item `("you",3)`. But I do not actually know that `("you",2)` is in the set. So I have to search the whole set to see if somewhere there is an item in the set which has "you" as the first element of the item – if I find it, I can read the second element, remove the item, and add a new item with 1 added to the second element; otherwise I add `("you",1)` to the set. Really, if I am going to jump through such hoops, I better just use a list.

For now, let's decide that we will use a dictionary to store the words with their counts.

5.4 Using functions for steps

We can now write with comments in the given code which steps we need to take:

```
def get_word_with_highest_count( filename ):
    # Step 1: Read the contents of filename, and put them in
    # string "contents".
    # Step 2: Build a dictionary "wordcounts" which contains
    # all the different words in "contents" with their counts.
    # Step 3: Select the word with the highest count from
    # "wordcounts".
    return "<word with highest count>"
```

While usually you will not write extra functions for small functionalities, you can actually do that. Since it makes sense to think of these steps as functions, I will make such functions here explicitly.

```
# Reads the contents of filename and returns those as a string.
def read_contents( filename ):
    return ""

# Builds a dictionary of all the words in the string contents,
   with their counts.
def build_dictionary( contents ):
    wordcounts = {}
    return wordcounts

# Selects the word with the highest count from dictionary
   wordcounts.
def select_highest_count( wordcounts ):
    return ""

def get_word_with_highest_count( filename ):
    contents = read_contents( filename )      # Step 1
    wordcounts = build_dictionary( contents ) # Step 2
    return select_highest_count( wordcounts ) # Step 3

print( get_word_with_highest_count( "pc_woodchuck.txt" ) )
```

We can now develop each of these functions separately, so that we can concentrate on just a small functionality. For instance, you probably have seen code that reads the contents of a file many times. If you have such code available, you can just copy and paste it (maybe with small adaptations, which you can make if you understand what you are copying/-pasting). Here is the `read_contents()` function completed (you can test it fairly easily by just printing what it returns).

```
def read_contents( filename ):
    with open( filename ) as fp:
        return fp.read()
```

We could probably also implement `select_highest_count()` fairly quickly if we had a test dictionary available. However, the second step, building a dictionary might still seem rather daunting. We need to further decompose the problem.

5.5 Further decomposition

Even if we feel that the problem decomposition we have made seems a good approach to creating a solution, we still may find that certain steps in the solution are hard to get grips on. That usually entails that such steps try to do too many things and need further decomposition.

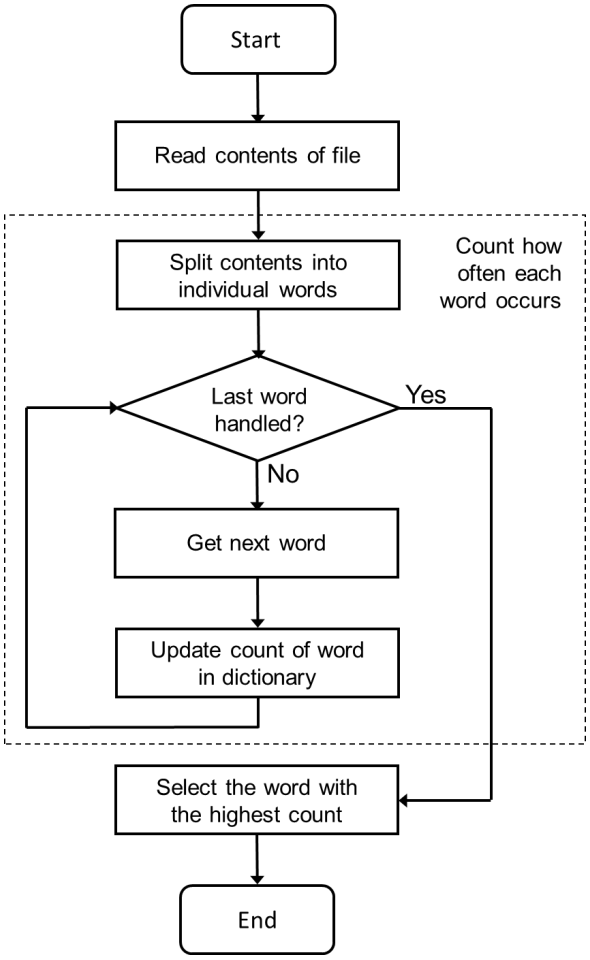

Figure 5.2: Expanded flow diagram of finding the word that occurs the most.

In the example given, going from a string to a dictionary which contains all the words that are in the string with their counts, is such a daunting step. What will we need to do to solve this?

First, we will need to split the string into individual words. We have to do that, because we have to handle individual words. Then we have to process each of those words, keeping track in a dictionary of how often we have encountered the word.

The previous flow diagram is expanded in Figure 5.2 to include further decomposition.

Since we now have extra steps in the flow diagram, we have to decide what the input and output of each of these steps is.

The step "Split contents into individual words" gets as input the contents of the file, represented as a string. As output it produces the individual words. A natural way to represent these words is as a list.

The list produced in the previous step is then processed in a loop, which is visualized in the flow diagram. Every word is examined, and the count of that word is updated in the

dictionary. The input for the step "Update count of word in dictionary" is the word that is being processed and also the dictionary. Without the dictionary, there is nothing to update! Once the loop is finished, we have a dictionary with word counts, which indeed is what needs to be produced.

We can add these extra steps to the code that we are writing:

```
# Reads the contents of filename and returns those as a string.
def read_contents( filename ):
    with open( filename ) as fp:
        return fp.read()

# Splits the string contents into a list of individual words.
def split_contents( contents ):
    wordlist = []
    return wordlist

# Updates the count of word in the dictionary wordcounts.
def update_wordcounts( wordcounts, word ):
    pass

# Builds a dictionary of all the words in the string contents,
# with their counts.
def build_dictionary( contents ):
    wordcounts = {}
    wordlist = split_contents( contents )
    for word in wordlist:
        update_wordcounts( wordcounts, word )
    return wordcounts

# Selects the word with the highest count from dictionary
# wordcounts.
def select_highest_count( wordcounts ):
    return ""

def get_word_with_highest_count( filename ):
    contents = read_contents( filename )      # Step 1
    wordcounts = build_dictionary( contents ) # Step 2
    return select_highest_count( wordcounts ) # Step 3

print( get_word_with_highest_count( "pc_woodchuck.txt" ) )
```

If it is not immediately clear to you how this code represents the flow diagram in Figure 5.2, study it until you understand.

Note that the function `update_word_counts()` does not return anything. The reason is that it updates a dictionary wordcounts; the effect that the function has is changing a dictionary, and not producing a return value. This is possible since dictionaries are "passed by reference."

5.6 Implementing steps

Let's now consider what is needed to implement the steps that we have distinguished.

`read_contents()` is already implemented.

`split_contents()` needs to split a string into individual words, and should produce a list of those words. If you do not know how to do that immediately, you may wish to further decompose this function. However, if you think back to the previous chapter, you may remember that in that chapter we already implemented this functionality. So with some copying and pasting, and a little bit of adaptation, you can easily fill this in. Note that this copying and pasting also introduces a new function in the code, namely the function `stringclean()`, which we developed earlier.

When we add this functionality to the code that we are developing, we can test it before we continue with implementing the rest of the code. Here is the code for the function `split_contents()`, which can be tested separately if you want:

```
# Replaces all non-letters in a string with spaces and makes the
   string lowercase.
def stringclean( inputstring ):
    outputstring = ""
    for letter in inputstring.lower():
        if letter >= 'a' and letter <= 'z':
            outputstring += letter
        else:
            outputstring += " "
    return outputstring

# Splits the string contents into a list of individual words.
def split_contents( contents ):
    return stringclean( contents ).split()
```

The next function we need to consider is `update_wordcounts()`. In this function we need to look up a word in a dictionary, if it isn't in the dictionary we have to add it with value 1, and if it is already in the dictionary we need to add 1 to the value. This is pretty standard code which you probably have at your fingertips, or quickly look up in the study material. It is only one line of code.

Finally, we need to implement the last step, where we have the select the word with the highest count. There are many different implementations that we can make for this functionality. One simple implementation is that we process each key of the dictionary and keep track of which has the highest value. Another approach is to turn the dictionary into a list of tuples of (key,value)-pairs, then sort that list according to the values. You can probably come up with other approaches as well. I don't think this step needs further decomposition, but you can make such a further decomposition if you need it.

Here is the completed code:

```
# Reads the contents of filename and returns those as a string.
def read_contents( filename ):
    with open( filename ) as fp:
```

```
        return fp.read()

# Replaces all non-letters in a string with spaces and makes the
   string lowercase.
def stringclean( inputstring ):
    outputstring = ""
    for letter in inputstring.lower():
        if letter >= 'a' and letter <= 'z':
            outputstring += letter
        else:
            outputstring += " "
    return outputstring

# Splits the string contents into a list of individual words.
def split_contents( contents ):
    return stringclean( contents ).split()

# Updates the count of word in the dictionary wordcounts.
def update_wordcounts( wordcounts, word ):
    wordcounts[word] = wordcounts.get( word, 0 )+1

# Builds a dictionary of all the words in the string contents,
   with their counts.
def build_dictionary( contents ):
    wordcounts = {}
    wordlist = split_contents( contents )
    for word in wordlist:
        update_wordcounts( wordcounts, word )
    return wordcounts

# Selects the word with the highest count from dictionary
   wordcounts.
def select_highest_count( wordcounts ):
    highestcount = 0
    highestword = ""
    for word in wordcounts:
        if wordcounts[word] > highestcount:
            highestword = word
            highestcount = wordcounts[word]
    return highestword

def get_word_with_highest_count( filename ):
    contents = read_contents( filename )      # Step 1
    wordcounts = build_dictionary( contents ) # Step 2
    return select_highest_count( wordcounts ) # Step 3

print( get_word_with_highest_count( "pc_woodchuck.txt" ) )
```

5.7 Isn't this code too convoluted?

You may wonder whether the code we produced for the problem is too convoluted. Besides the basic function `get_word_with_highest_count()` that we needed to implement, we implemented six other functions. Two of these only have one line of code, and one has two lines of code. All the functions are very simple and straightforward.

My answer is that, yes, I think the code is a bit too convoluted the way it is written now. Having so many small functions does not make the code more readable. However, as an illustration of problem decomposition I think this works well: you can actually see that solving a bigger, fairly hard problem can be accomplished by solving six small, easy problems.

In general, however, for this problem I would probably keep the function `stringclean()` as a separate function (as it was copied from somewhere else), but leave the rest of the code in the function `get_word_with_highest_count()`, perhaps with some comments in between the different steps of the flow diagram. The code would then look like this:

```
def stringclean( inputstring ):
    outputstring = ""
    for letter in inputstring.lower():
        if letter >= 'a' and letter <= 'z':
            outputstring += letter
        else:
            outputstring += " "
    return outputstring

def get_word_with_highest_count( filename ):

    # Read the contents of filename
    with open( filename ) as fp:
        contents = fp.read()

    # Build a dictionary of all the words in the string contents,
    # with their counts.
    wordcounts = {}
    wordlist = stringclean( contents ).split()
    for word in wordlist:
        wordcounts[word] = wordcounts.get( word, 0 )+1

    # Select the word with the highest count from dictionary
    highestcount = 0
    highestword = ""
    for word in wordcounts:
        if wordcounts[word] > highestcount:
            highestword = word
            highestcount = wordcounts[word]
    return highestword

print( get_word_with_highest_count( "pc_woodchuck.txt" ) )
```

5.8 How to decompose a problem

Problem decomposition goes through the following steps:

- If a problem seems to be too big to solve in one go, try to determine the different steps you need to take to solve the problem. You may be inspired by how you would solve the problem manually to determine the steps. It may help initially to draw a flow diagram or write pseudo code for the problem.
- For each step, determine what you need to execute the step, and what the step produces. Determine whether you indeed have what you need to take the step.
- If a step still seems to big to solve it in one go, further decompose it into smaller steps.
- Once all the steps are so simple that you can write code for them without much further thinking, implement and test the steps one by one.

Exercises

Exercise 5.1 Several issues were not dealt with in the program developed in this chapter. One issue is what you will do when the file cannot be found. In that case, the program should return the word "Error". Another is that there may be more than one word which occurs the most. In that case, the program should return of those possible answers the word that is earliest in the alphabet. Identify in which steps you will have to take care of these issues, and adapt the program accordingly. Note that one of the advantages of a good decomposition is that it isn't hard to find where an adaptation must be made.

Exercise 5.2 When you count how often words occur in a file, you will find that some words occur only once, some words occur twice, some words occur three times, etc. Design and implement a function which returns how many words in a file occur only once. You probably realize that you can reuse most of the code which is given in the chapter; you only need to adapt the last step a bit.

Exercise 5.3 In continuation of the previous exercise, design and implement a function which returns which wordcount occurs least often for a file. For instance, supposed that 5 words occur once, 3 words occur twice, 6 words occur three times, and 2 words occur four times, then the function returns 4, as the wordcount that occurs the least is 4 (namely it occurs for only 2 words). If you base this function on the code that was developed in this chapter, you will have to redesign the final step of the code. Do this with a flow diagram before implementing it. You may want to build a new data structure from the dictionary `wordcounts` in your solution.

Exercise 5.4 Take a number. If the number is even, divide it by 2, otherwise multiply it by 3 and add 1. Repeat the procedure with the new number. Continue repeating the procedure until you reach 1. E.g., if you start with 10, you divide by 2 to get 5. You multiply by 3 and add 1, which gives 16. 16 gets divided by 2 which gives 8. 8 gets divided by 2 which gives 4. 4 gets divided by 2 which gives 2. 2 gets divided by 2 which gives 1. The procedure was executes 6 times, namely 5 times dividing by 2, and once multiplying by 3 and adding 1.

The Gollatz Conjecture says that this procedure will inevitably lead to 1. While the Conjecture is not proven yet, it has been checked up to numbers of 19 digits.

Design and implement a function which gets a number as input and returns how often the procedure was executed before 1 was reached. Start by making a flow diagram which represents a task decomposition. If you can implement such tasks immediately, do so. Otherwise, make a further decomposition before you start implementing.

Exercise 5.5 In the Hangman game, you have to guess a word. You do this by guessing letters. The word is displayed as dashes for each letter, where a dash is replaced by the correct letter when you have guessed it. For instance, the word PROGRAMMING would be displayed as `----------` as long as you have not guessed any letters, but when you have guessed R, it is displayed as `-R--R------`, and if you then guess M, it is displayed as `-R--R-MM---`. When guessing letters, a "fail" means that you guess a letter which is not in the word. You have a limited number of fails available before you lose the game. You win if you guess the word.

Consider a Hangman program in which the computer controls the game. The computer selects a word, and lets the human player make guesses. The game ends with a win for the human when the word is completed, or with a loss for the human when there are 10 fails.

Design and implement a Hangman program. Start with a high-level flow diagram. Each of the steps in the flow diagram is a task, and the flow diagram represents the problem decomposition. If for a task you can easily implement an algorithm, program it. If not, make a further decomposition of the task.

Once you have an initial version of the program working, you will probably see further improvements that you can make. For instance, if your implementation lets the computer select a word from a small list, you can let the computer select the word from a dictionary which is stored in a text file. If you allowed the human to repeat letters as guesses, you can build in a check for that. If you did not tell the human what the actual word was when the game ends in a loss, you should probably add that. You can let the human specify the length of the word before a word is selected. Etcetera. This probably means that some of the tasks you defined become more complex, and you may have to make a further decomposition of the task. However, you should note that if you made a good problem decomposition in the first place, you usually only need to make changes to a single task to add the new functionality.

Chapter 6

Top-down Development

In the previous chapter a problem was divided into tasks, which could be further split up into smaller tasks until all tasks were small enough to handle. This is an example of top-down programming. So at this point you probably already understand how to do top-down programming, but the topic can use some further discussion.

6.1 Top-down programming

In top-down programming we start by creating a program which consists of certain high-level tasks, which, when taken together, would clearly solve the programming problem if they are implemented correctly. In general, a first flow diagram which you would make for such a programming task would represent this top-down view.

For instance, suppose that we want to implement a program which plays the game Hi-Lo. In this game, one player memorizes a number between 1 and, let's say, 1000, and the other player has to guess the number. Every time the second player states a number, the first player will say whether the memorized number is higher than, lower than, or equal to the guessed number. The game ends when the number is guessed correctly, or the second player took too many guesses without arriving at the correct number, e.g., 10 guesses.

Let's assume that the computer memorizes a number, and the human plays the game. Figure 6.1 shows a flow diagram for this game.

If you do not understand why this flow diagram correctly represents the game, study it until you do. In (pseudo) code form, where the function `hi_lo()` plays the game, it looks like this:

```
def hi_lo():
    number = ...; # memorize a number
    guesses = 0;
    while guesses < 10:
        guess = ...; # let human make a guess
        if guess == number:
            print( "You win!" );
            return;
```

```
            if guess < number:
                print( "Higher" );
            else:
                print( "Lower" );
            guesses += 1;
        print( "You lose!" );

hi_lo();
```

When you examine the tasks distinguished in the flow diagram, you probably will say of some of them "I can implement those easily," while for others you may say "I have to think a bit about how to implement these tasks." In fact, in the pseudo code, I already implemented the reports of wins and losses, the reporting of "higher" and "lower," and the increase of the number of guesses, as they are such small and easy tasks.

For the sake of argument, let's say that the two tasks that seem to be more of a challenge to implement are "memorize a number" and "let the human make a guess."

6.2 Postpone tasks via functions

When you must implement a relatively complex program, you do not want to postpone testing the program until you have implemented everything. Using top-down programming, you can postpone implementing certain functionalities by placing them in functions

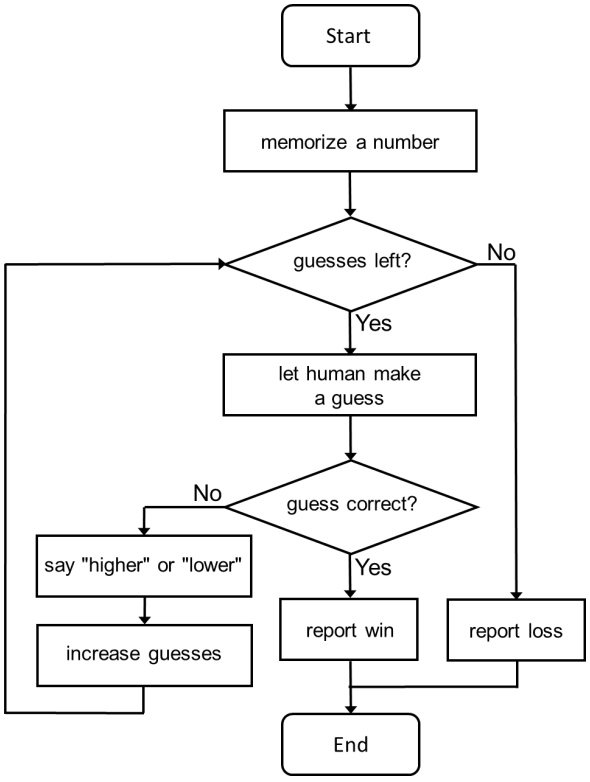

Figure 6.1: Flow diagram for the Hi-Lo game.

and letting these functions return something that is a legal return value but generated in a very simple and straightforward way – usually such a function would just return a value and do nothing more.

For each of these functions you must define what it needs (the input) and what it produces (the output).

I have completed the pseudo code above, and turned it into runnable code, by adding functions for the two functionalities which were considered challenging. Neither of these functions seems to need an input, and produces a number as output. So both functions just return a number.

```
def memorize_a_number():
    return 750

def let_human_make_a_guess():
    return 500

def hi_lo():
    number = memorize_a_number()
    guesses = 0
    while guesses < 10:
        guess = let_human_make_a_guess()
        if guess == number:
            print( "You win!" )
            return
        if guess < number:
            print( "Higher" )
        else:
            print( "Lower" )
        guesses += 1
    print( "You lose!" )

hi_lo()
```

You may notice that you can run this code, and it seems to work well – except that it always uses 750 as the number to guess, and never allows the human to enter a guess.

With the program in the current state, you can develop the challenging functions step by step.

The first function is actually pretty simple. To memorize a number, a random integer between 1 and 1000 must be selected. This can be done with a call to the `randint()` function.

```
from random import randint

def memorize_a_number():
    return randint( 1, 1000 )
```

The second function is actually harder to develop, and may need further task decomposition.

6.3 Developing a number input function

You may think at first that it is fairly simple to create a function that gets a number from the player. You simply make a call to the **input**() function, turn the resulting string into an integer, and return it. And indeed, you can build that implementation in the function given, and it will allow you to test the program as a whole. However, you may also find that there are several parts of the program not yet working well. In particular, if you think about it, your function should be expanded with one or more of the following:

- the player must enter an integer between 1 and 1000
- if the player enters an integer outside the given range, an error message must be displayed and they must enter a new integer
- if the player enters something which is not a number, an error message must be displayed and they must enter a new integer
- as a hint to the player, they should be informed about how many guesses they already made

For the last addition, the function must know how many guesses the player already made. This information can be provided to the function with a parameter.

As an exercise, you can make a flow diagram for just the `let_human_make_a_guess()` function. After that you can make your implementation. For completeness sake, I give my implementation of the finished program here, but with a different flow diagram the implementation may turn out differently.

```
from random import randint

def memorize_a_number():
    return randint( 1, 1000 )

def let_human_make_a_guess( guesses ):
    print( "Number of guesses until now:", guesses )
    while True:
        strnum = input( "Guess a number between 1 and 1000: " )
        if not strnum.isdigit():
            print( "This is not a positive integer" )
            continue
        num = int( strnum )
        if num < 1 or num > 1000:
            print( "This is not a number between 1 and 1000" )
            continue
        break
    return num

def hi_lo():
    number = memorize_a_number()
    guesses = 0
    while guesses < 10:
        guess = let_human_make_a_guess( guesses )
        if guess == number:
```

```
            print( "You win!" )
            return
        if guess < number:
            print( "Higher" )
        else:
            print( "Lower" )
        guesses += 1
    print( "You lose! The correct number was", number )

hi_lo()
```

6.4 How to implement a program top-down

With top-down programming you go through the following steps:

- You make a general, high-level outline of the program, consisting of tasks that you need to implement to make the program work.
- Very simple tasks you implement immediately. For more complex tasks you call functions which simply return a legal value.
- You now should have a program that can run and can be tested, albeit with very limited functionality.
- You now create an implementation of each of the functions that still need an implementation. You do this function by function, and if necessary apply further problem decomposition for each of the functions.

The big advantage of top-down programming is in the third step: very early in the development process you have a working program available that can be tested. Moreover, you generally do not have to develop the functions in the order that the program needs them: even if a function depends on the output of another function which would be called earlier, that other function already exists and gives a legal output, so a more detailed implementation of that function can be postponed if you want.

Exercises

Exercise 6.1 Suppose that in the Hi-Lo game you want to make the maximum number that can be guessed not 1000, but some other value. This value will be specified at the start of the program (e.g., by filling a variable `maxnum`). Identify in which places of the program this new maximum needs to be known. Adapt those places. If it is in a function, it must be given as a parameter to the function.

Exercise 6.2 Suppose that in the Hi-Lo game the number of guesses that the player is allowed to make, should be determined by the player at the start of the program. This will be an extra task in the flow diagram. Extend the flow diagram, and add this functionality to the program.

Exercise 6.3 Turn the Hi-Lo program around: you as a human are going to memorize a number, and the computer must guess it. The flow diagram is going to be almost the same as given above, but it needs a very different implementation for the making of the guess, and the generation of the response "higher," "lower," or "correct." Also, the computer must have the ability to challenge the human player if the responses that the human player gave are clearly in contradiction with each other.

Design and implement this program using a top-down approach.

Exercise 6.4 Consider a program which keeps track of an inventory. The inventory consists of different goods, which have a name, and a quantity. When you start the program, there is no inventory. You can give the program three different commands. A command consists of a letter, potentially followed by a comma, a word, another comma, and an integer.

The first command is A for "add," which adds goods to the inventory. It is followed by a word which indicates the name of the good, and an integer which indicates the quantity of the good. These goods are added to the inventory. For instance, "A,apples,100" adds 100 apples to the inventory. If the good is already in the inventory, you have to add the quantity to what there already is for that good in the inventory.

The second command is R for "remove," which removes goods from the inventory. The word is the good, and the integer is how much of that good should be removed. For instance, "R,apples,50" removes 50 apples from the inventory. If there isn't enough of the good in the inventory, no change is made but an error message is given.

The third command is L for "list," which simply lists all the goods in the inventory with their quantities.

Design this program using a top-down approach. You will probably need to distinguish four tasks which are more complex, for which you should use separate functions (namely getting and unraveling the command, and the three different commands). For the inventory you best use a dictionary.

Chapter 7

Bottom-up Development

In the previous chapter I explained how you can use a top-down approach to quickly create a working program, which is not completed yet, and then fill in different functionalities as you further develop the program. Sometimes, however, a task is of such complexity that you cannot really distinguish different steps which you can implement separately. In such cases, a bottom-up approach might be more suited to solve the problem.

7.1 Bottom-up programming

In bottom-up programming we recognize that a program consists of a simple task, embedded in a more complex task, which in its turn is embedded in an even more complex task, etc. While sometimes a top-down approach might still be used to tackle such a programming problem, it may make more sense to start implementing the simplest task first, and then "work upward or outward."

As this is a rather confusing explanation, an example may work to make clear what I mean by this. Suppose that you get the following programming problem: You roll six six-sided dice; you set all dice that show a six aside, and reroll the remaining dice; you continue doing this until all dice show a six. Estimate, using a simulation, how many rolls you need to make all dice show a six.

It is actually not that hard to imagine how you do this as a human. You would take six dice, and begin rolling, setting aside all sixes until all dice show six. You write down how many rolls you needed. You then repeat the process a large number of times, each time writing down the number of rolls needed. When you think you have gathered enough rolls, you add them all up and divide by the number of trials. This is a boring task, but that is why we have a computer.

When you think about this procedure, you may recognize that it consists of a number of nested loops. The outermost loop is the loop which controls the trials. In there you have a loop which controls the rerolls of the remaining dice. In there you have a loop which rolls each separate die, and keeps track of the dice that show a six.

An experienced programmer may see no issues here and immediately code the whole procedure, but for an inexperienced programmer this is harder. A top-down approach would

start at the outer loop, but it only makes sense to create that outer loop when you have code for the tasks that are in it. In this case, if you want to handle the problem step by step, you may want to start with the innermost task.

Note: I am not saying that a top-down approach could not work here, but for inexperienced programmers it is helpful if you can immediately see that a task that you implemented is doing what it should do, and for a nested loop it is not easy to see that the outer loop is doing what it is supposed to do without having access to the inner loops.

7.2 Rolling dice

When creating a solution for the die-rolling problem, we have to identify what the simplest task is that we need to solve. In the example, the simplest task is the rolling of a single die. There is no smaller task. We start by creating code for this task, which is a call to the `randint()` function.

```
from random import randint

# Roll a single die
die = randint( 1, 6 )

print( die )
```

Now rolling a single die works, you have to continue by rolling multiple dice. The problems says that you roll six dice. Rolling six dice you do in a loop.

```
from random import randint

# Roll 6 dice
for i in range( 6 ):
    # Roll a single die
    die = randint( 1, 6 )
    print( die )
```

The next step is that we run an actual trial: we need to set aside the sixes, and continue repeating the rolling of the remaining dice until we have six sixes. This is probably the hardest step, so we may want to split it up into different steps again. First, let's determine how many sixes we have rolled.

```
from random import randint

number_of_sixes = 0
# Roll 6 dice
for i in range( 6 ):
    # Roll a single die
    die = randint( 1, 6 )
    print( die )
    # Count number of sixes
    if die == 6:
```

```
        number_of_sixes += 1

print( "Number of sixes:", number_of_sixes )
```

The code above represent the rolling of six dice once. But we need to reroll the dice until they all show a 6. This means that we have to build a loop around this code. To build that loop, we need to realize two things: (1) the variable `number_of_sixes` is set to zero before we do the first roll, and we continue rolling dice until this variable is 6; and (2) we only reroll those dice which do not show 6, so we do not roll six dice every time, but `6 - number_of_sixes dice`. This leads to the following code:

```
from random import randint

number_of_sixes = 0
while number_of_sixes < 6:
    # Roll all dice which do not show 6 yet
    for i in range( 6-number_of_sixes ):
        # Roll a single die
        die = randint( 1, 6 )
        print( die )
        # Count number of sixes
        if die == 6:
            number_of_sixes += 1
    print( "Number of sixes:", number_of_sixes )
```

What we have to count is how many rolls we needed to get to six sixes, so we have to count how often the **while**-loop is repeated. We do this by initializing a variable rolls to zero at the beginning, and increasing it every time we go through the outer loop.

```
from random import randint

rolls = 0
number_of_sixes = 0
while number_of_sixes < 6:
    # Roll all dice which do not show 6 yet
    for i in range( 6-number_of_sixes ):
        # Roll a single die
        die = randint( 1, 6 )
        print( die )
        # Count number of sixes
        if die == 6:
            number_of_sixes += 1
    # Count number of rolls
    rolls += 1
    print( "Number of sixes:", number_of_sixes )
print( "Number of rolls:", rolls )
```

You can test this code and see that it works. We have completed the code for running a single trial. We now need to repeat this a large number of times, and calculate the average

of the number of rolls needed. I usually recommend starting with a small number of trials as you do not know yet how slow the code is. Also, before you add running more trials, you should remove the **print**() statements that were added for debugging purposes, otherwise the output will be unreadable.

In the following code a variable total_rolls is added, which keeps track of the total number of rolls for all the trials. At the end of the program, this total is divided by the number of trials to determine the estimated number of rolls needed.

```
from random import randint

total_rolls = 0
for j in range( 1000 ):
    rolls = 0
    number_of_sixes = 0
    while number_of_sixes < 6:
        # Roll all dice which do not show 6 yet
        for i in range( 6-number_of_sixes ):
            # Roll a single die
            die = randint( 1, 6 )
            # Count number of sixes
            if die == 6:
                number_of_sixes += 1
        # Count number of rolls
        rolls += 1
    # Count total number of rolls needed
    total_rolls += rolls

print( total_rolls/1000 )
```

7.3 Improvements to the dice rolling program

While the dice rolling program that we developed works, I wish to point out several potential improvements.

The first improvement has to do with so-called "magic numbers." Sometimes a program contains numbers which have an arbitrary value that needs an explanation on what they mean. In the case of the developed program, there are three magic numbers: 6 which is the maximum value of a die, 6 which is the number of dice, and 1000 which is the number of trials.

In general, it makes sense to replace such magic numbers with a constant: in Python a variable with a fixed value which is never changed in the program, and which for clarity is written in uppercase letters. There are three advantages of doing this: (1) the meaning of such a value is self-documented; (2) if two of these magic numbers are the same (e.g., 6 in this program) their uses are clearly distinguished in the program; and (3) if the value changes (e.g., you want to run more trials), you only need to change it in one place in the program.

The second improvement is to pack running a single trial in a function. While the present program is rather short and may not really need functions, for more complex programs which, for instance, need deeper nested loops, packing a task which can be distinguished naturally in a function may improve readability and may help debugging.

If you do this, you have to consider what the function needs (the input) and what it produces (the output). In this case, the function does not need any input, but will produce the number of rolls needed, as that is the only thing that we need to know of a trial. With the introduction of such a function, namely provide it with the imput parameters which make it "generalized" for different numbers of dice, and different types of dice.

The whole program, with constants for magic numbers and a generalized trial function, becomes:

```
from random import randint

# Runs a single trial, in which num_dice dice are rolled, and all
# dice which do not show die_value are rerolled until they all
# show die_value. It returns the number of rolls needed.
def trial( die_value, num_dice ):
    rolls = 0
    number_of_die_value = 0
    while number_of_die_value < num_dice:
        # Roll all dice which do not show die_value yet
        for i in range( num_dice - number_of_die_value ):
            # Roll a single die
            die = randint( 1, die_value )
            # Count number of dice which show die_value
            if die == die_value:
                number_of_die_value += 1
        # Count number of rolls
        rolls += 1
    return rolls

TRIALS = 1000
DIE_VALUE = 6
NUM_DICE = 6

total_rolls = 0
for j in range( TRIALS ):
    rolls = trial( DIE_VALUE, NUM_DICE)
    total_rolls += rolls

print( total_rolls/TRIALS )
```

7.4 How to implement a program bottom-up

With bottom-up programming you go through the following steps:

- You identify the simplest tasks in the program, and implement those first and test them.
- You build more complex tasks around the simpler tasks that you already implemented, and test them.
- When you have implemented a naturally distinguishable task, you consider placing that task in a function. This is particularly helpful to avoid loops that are nested so deep that they become unreadable.

Bottom-up programming is most useful for tasks which are complex but which cannot easily be distinguished in sequential steps.

Exercises

Exercise 7.1 Design and write a program which estimates what the chance is that if you roll six regular six-sided dice, that the total value of the dice is 25 or higher. Use a bottom-up approach. Also make sure that it is easy to make the following three changes to the program: (1) you roll a different number of dice; (2) you roll a different kind of dice (e.g., 8-sided); and (3) you want the total value of the dice to be higher than something else than 25.

Exercise 7.2 List all the 4-digit numbers for which the sum of the digits is equal to the product of the digits. Use a bottom-up approach.

Exercise 7.3 List all numbers in a given range (so not just 4-digit numbers) for which the sum of the digits is equal to the product of the digits. If you wrote the code for the previous exercise in a generalised way, you have pretty much solved this. If you did not use a generalised way, you are probably going to have to add another loop somewhere.

Chapter 8

Generalization

At the end of the previous chapter I showed how you can create a function that solves a task in a more general way than for the specific case that was required to solve the particular problem that you have at hand. A typical application of functions (and other programming concepts, but in this course the focus is on functions) is to generalize over multiple problems which need the same or very similar solutions. While seldom needed for smaller problems, it can be helpful to gain a "feel" for programming to think about how you can solve a problem in a more generalized way.

8.1 Getting input with constraints

In a previous chapter I introduced a function which asks the user for a number, which we used to ask for a number between 1 and 1000. The function looked something like this:

```
def input_number_between_1_and_1000():
    while True:
        strnum = input( "Give me a number between 1 and 1000: " )
        if not strnum.isdigit():
            print( "This is not an integer" )
            continue
        num = int( strnum )
        if num < 1 or num > 1000:
            print( "This is not a number between 1 and 1000" )
            continue
        break
    return num

num = input_number_between_1_and_1000()
print( "The number is:", num )
```

Suppose that we need to write a program which asks the user for three numbers. One can be any number, as long as it is an integer, one has to be between 1 and 1000, and the last one has to be between 10 and 30. Potentially, you could write that program as follows:

```
def input_number():
    while True:
        strnum = input( "Give me a number: " )
        if not strnum.isdigit():
            print( "This is not an integer" )
            continue
        num = int( strnum )
        break
    return num

def input_number_between_1_and_1000():
    while True:
        strnum = input( "Give me a number between 1 and 1000: " )
        if not strnum.isdigit():
            print( "This is not an integer" )
            continue
        num = int( strnum )
        if num < 1 or num > 1000:
            print( "This is not a number between 1 and 1000" )
            continue
        break
    return num

def input_number_between_10_and_30():
    while True:
        strnum = input( "Give me a number between 10 and 30: " )
        if not strnum.isdigit():
            print( "This is not an integer" )
            continue
        num = int( strnum )
        if num < 10 or num > 30:
            print( "This is not a number between 10 and 30" )
            continue
        break
    return num

num1 = input_number()
num2 = input_number_between_1_and_1000()
num3 = input_number_between_10_and_30()
print( f"The numbers are {num1}, {num2}, and {num3}." )
```

While this program works as it should, I assume that you see that this approach, whereby you have three functions which are pretty much the same apart from some details, is rather inferior. If you do not see it, please consider the following:

- If you discover that you made a mistake in one of the functions, you probably have to fix it in three places rather than just one place.
- If you have to extend the program and let it ask for a number between 90 and 250, a number smaller than 120, and a number between -40 and 40, if you follow the same

approach you have to make three more copies of one of the functions and adapt those.

8.2 Generalizing the input function

To make the program more elegant, we want to make a generalized number-input function. This function should be usable in each of the cases that we need a number as input. To generalize a function, we need to identify what different cases the function needs to handle, and how we can let the function distinguish those cases based on input parameters.

First, let's consider the situation in which the function has to ask a number in a certain range, like a number between 1 and 1000, and a number between 10 and 30. You can probably see that you can do this if you give the minimum and the maximum value of the range as parameters. We then get a function like this:

```
def input_number_in_range( minvalue, maxvalue ):
    while True:
        strnum = input( f"Give number ({minvalue}-{maxvalue}): " )
        if not strnum.isdigit():
            print( "This is not an integer" )
            continue
        num = int( strnum )
        if num < minvalue or num > maxvalue:
            print( f"Number outside range {minvalue}-{maxvalue}" )
            continue
        break
    return num

num1 = input_number_in_range( 1, 1000 )
num2 = input_number_in_range( 10, 30 )
print( f"The numbers are {num1} and {num2}." )
```

Note that the `minvalue` and `maxvalue` are used in three places in the function, namely in the prompt, in the comparison, and in one of the error messages.

There is a bit of a problem in this function if the programmer calls the function with a `minvalue` that is higher than the `maxvalue`, because then the loop in the function becomes endless. To handle that, we can, for instance, let the function generate an exception (exception generation is a bit of an advanced topic, but it entails that the program will end with an error message if a certain condition occurs, in this case, when the function is called with `minvalue` greater than `maxvalue`). The function then looks as follows:

```
def input_number_in_range( minvalue, maxvalue ):
    if minvalue > maxvalue:
        raise Exception( f"Error: {minvalue} > {maxvalue}" )
    while True:
        strnum = input( f"Give number ({minvalue}-{maxvalue}): " )
        if not strnum.isdigit():
            print( "This is not an integer" )
            continue
        num = int( strnum )
```

```
        if num < minvalue or num > maxvalue:
            print( f"Number outside range {minvalue}-{maxvalue}" )
            continue
        break
    return num
```

In the solution above we can use one function to ask for a number between 1 and 1000, and between 10 and 30. However, we also need to be able to ask for "any number." How can we solve that?

Different approaches are possible. If there is a certain absolute minimum and a certain absolute maximum, we can give those as arguments. However, Python does not have any limitation on the size of integers (most other programming languages do, and this is a common approach in such languages). We could also add a third parameter, which is a boolean which indicates whether "any number is allowed" or "only numbers in a certain range are allowed." However, in that case we need to work with an extra parameter of which the functionality is confusing.

You may also consider that a range is only legal if the `minvalue` is not higher than the `maxvalue`. We could therefore define that if you give a `minvalue` which is higher than the `maxvalue`, the function does not raise an exception, but simply allows any number. We can give default values to `minvalue` and `maxvalue` such that when the function is called without these arguments, it will allow any number, while if a `minvalue` and `maxvalue` are specified, they are used to constrain the input.

Since the function is now no longer limited to a range, we change the name to `input_number()`, which looks like this:

```
def input_number( minvalue=0, maxvalue=-1 ):
    while True:
        prompt = "Give me a number: "
        if minvalue <= maxvalue:
            prompt = f"Give a number ({minvalue}-{maxvalue}): "
        strnum = input( prompt )
        if not strnum.isdigit():
            print( "This is not an integer" )
            continue
        num = int( strnum )
        if minvalue <= maxvalue:
            if num < minvalue or num > maxvalue:
                print( f"Nr outside range {minvalue}-{maxvalue}" )
                continue
        break
    return num
```

8.3 Abstraction

Abstraction is a form of generalization. I want to mention it here because it comes up sometimes, and for many it is unclear what the difference between abstraction and gener-

alization is. And frankly, the difference in computational terms is not entirely clear.

We talk about abstraction when referring to the common attributes of a group of entities which we deal with in a program. For instance, when we have a program that deals with creating identification cards for the people who have an affiliation with a university, that program only needs to know of a person their administration number, their first name, and their last name, because it only needs to know these features to create a card. Of course, persons are much more than those three features, but for the program an abstraction of a person is an entity which only has these three features. As long as an entity is given to the program which has the features of this abstraction of a person, it can create a card for the entity.

So it does not matter if the person is a student or a staff member, they can get a card from the program. Maybe in the past there was a separate program for creating cards for students and for creating cards for staff members, but since students and staff members have the features of the same, abstract person, one program is all that is necessary. You can say that the program has generalized over persons' roles.

Abstraction is not of importance for the contents of this course, as it ignores object orientation, which is an advanced topic. Once you study object orientation, however, you may encounter the notion of "abstract classes," which are entities in a program which are never instantiated by themselves, but which can be implemented by more specified classes. I will not say more about it here, as it will come up when you get to object orientation.

Exercises

Exercise 8.1 Write a function which rolls a six-sided die. Call the function 1000 times, and calculate the average value of the roll of such a die (the answer should be close to 3.5).

Exercise 8.2 While six-sided dice are the most common dice, there also exist 2-sided dice (basically, coins), 3-sided dice, 4-sided dice, 8-sided dice, 10-sided dice, 12-sided dice, and 20-sided dice. Adapt the function from the previous exercise in such a way that you can roll any of these dice based on a parameter that you give to the function.

Exercise 8.3 In certain roleplaying games you can roll dice with "advantage" or "disadvantage." This entails that you roll two of the dice, and with advantage you take the higher roll, and with disadvantage you take the lower roll. Generalize the function from the previous exercise in such a way that you can roll with advantage, with disadvantage, or normally.

Exercise 8.4 For some board games specialty dice are made. For instance, the game "Dead End Drive" uses six-sided dice with the values 2–3–3–4–4–5. "Backgammon" uses a six-sided die with the values 2–4–8–16–32–64. "Betrayal at House on the Hill" uses six-sided dice with the values 0–0–1–1–2–2. "Calendar Dice" are two six-sided dice, one with the values 0–1–2–3–4–5, and one with the values 0–1–2–6–7–8. "Formula D" has seven rather weird dice: a four-sided die with the values 1–1–2–2, a six-sided die with the values 2–3–3–4–4–4, an eight-sided die with the values 4–5–6–6–7–7–8–8, a twelve-sided die with the values 7 to 12 twice, a twenty-sided die with the values 11 to 20 twice, a thirty-sided

die with the values 21 to 30 three times, and one regular twenty-sided die with the values 1 to 20. Generalize the function that you made in the previous exercise so that it also works with such irregular dice. Note that you need some way to communicate the kind of die to the function. You may assume that a die only has integers on its sides.

Exercise 8.5 Generalize the number-input function developed in this chapter so that it also accepts floating-point numbers. You should be able to indicate to the function whether it should be limited to integers or that it can accept both integers and floating-point numbers.

Chapter 9

Implementation Tips

When students start out with programming, they still have to discover certain standardized ways of implementation. If you stick to such standardized ways, you have a much easier time designing programs as you no longer need to think about certain details which are obvious to experienced programmers. In this chapter I will discuss some of these ways. The chapter is aimed specifically at Python programming, as other programming languages may have idiosyncrasies which may make some of the ways discussed here less applicable.

9.1 Choosing data structures

Data structures are constructs which allow you to store values. When you learn Python programming, initially there are three data structures which you focus on: single-value variables (integers, strings, and floating-points), lists, and dictionaries. You may occasionally encounter two more: tuples and sets. For novice programmers it is often unclear when one should use which data structure. There are some straightforwards rules-of-thumb which one can use to make decisions in this respect.

Obviously, when you need to store a single value, you use a single-value variable. That much is clear. But when do you use a list, dictionary, tuple, or set? Tuples and sets are used in exceptional cases, so let's first look at lists and dictionaries.

A *list* is a data structure with an ordering. That means that if you have a series of values which you need to store, and you need to be able to say "this one is the first, this one is the second, this one is the third, etc." then a list is the obvious choice. Furthermore, since a list has an ordering, it can be sorted. None of the other standard data structures can be sorted. Thus, if you need sorting, you need a list.

A dictionary is a data structure which has no ordering. It is used to store values which belong to a "key." Thus you use a dictionary when you need to store a series of values, which you can look up based on some key. Often, you think of dictionaries as data structures which store "pairs," namely a value tied to a key. (Note: since Python 3.7 dictionaries seem to have an ordering, namely the insertion order is preserved; however, internally there is no ordering, and you cannot apply any methods which work with ordering, such as sorting.)

To give an example to distinguish the two, suppose that you have to read all the words in a file, and do something with them. Which data structure do you use? The answer is: that depends on what you need to do with them. If, for instance, you need to present all the words in the file in alphabetical order, then you will have to sort them at some point and thus a list is the obvious way of storing them. However, if you have to count how often each word occurs in the file, you have to store a word with a counter, i.e., you have to store key-value pairs, so a dictionary is the obvious way of storing them.

Here is some code which does both these things (if you want to practice with coding, you may wish to develop this code yourself before you look at how I do it; I have based my code on the code developed in the chapter on problem decomposition):

```
def stringclean( inputstring ):
    outputstring = ""
    for letter in inputstring.lower():
        if letter >= 'a' and letter <= 'z':
            outputstring += letter
        else:
            outputstring += " "
    return outputstring

def get_wordlist( filename ):
    with open( filename ) as fp:
        contents = fp.read()
    wordlist = stringclean( contents ).split()
    wordlist.sort()
    return wordlist

def get_wordcounts( filename ):
    with open( filename ) as fp:
        contents = fp.read()
    wordcounts = {}
    wordlist = stringclean( contents ).split()
    for word in wordlist:
        wordcounts[word] = wordcounts.get( word, 0 )+1
    return wordcounts

print( "Word list:" )
wordlist = get_wordlist( "pc_woodchuck.txt" )
print( ", ".join( wordlist ) )
print()
print( "Word counts:" )
worddict = get_wordcounts( "pc_woodchuck.txt" )
for word in worddict:
    print( f"{word}: {worddict[word]}" )
```

If this difference is clear, then you may consider when to use a set or a tuple.

A *set* is unordered, and each item in the set can occur only once. You may have noticed that in the code above, the list of words had words which occurred multiple times. Perhaps you do not want that. You may then consider using a set to store the words, because it

will guarantee automatically that each word would be in the set just once. However, a set would not be a suitable approach for this, because I said that the words should be presented in alphabetical order. Since a set cannot be ordered, you cannot sort it. So when would you use a set?

The main reason to use a set is when you have to perform set operations. You can create a union of two sets, or create the intersection of two sets. Other set operations are possible as well. Thus if, for instance, I have two files and I am asked to list all the words that occur in both files, a good solution would be to put all the words of one file in one set, and all the words of the other file in another set, and then create the intersection of those two sets. Obviously, it does not happen often that you would need to perform such operations. Therefore the application of sets is rare.

So how would I solve the issue with listing all the words of a file in alphabetical order, but only list each word once? There are numerous solutions for this problem, of which I list two. The first is that before I store a word in the list, I check if it is already in the list, and only store it if it is not there yet. This is an acceptable solution, though relatively time-intensive, as checking whether something exists in a list is an expensive operation. The second solution is to simply build the list, then convert the list to a set, then convert the set back to the list, and then sort the list. The conversion to the set I do to get rid of doubles, and the conversion back to the list I do because I need to sort the list. Both these solutions are in the following code:

```
def stringclean( inputstring ):
    outputstring = ""
    for letter in inputstring.lower():
        if letter >= 'a' and letter <= 'z':
            outputstring += letter
        else:
            outputstring += " "
    return outputstring

def get_wordlist_with_unique_test( filename ):
    with open( filename ) as fp:
        contents = fp.read()
    words = stringclean( contents ).split()
    wordlist = []
    for word in words:
        if word not in wordlist:
            wordlist.append( word )
    wordlist.sort()
    return wordlist

def get_wordlist_via_set( filename ):
    with open( filename ) as fp:
        contents = fp.read()
    wordlist = stringclean( contents ).split()
    wordset = set( wordlist )
    wordlist = list( wordset )
    wordlist.sort()
    return wordlist
```

```
wordlist = get_wordlist_with_unique_test( "pc_woodchuck.txt" )
print( ", ".join( wordlist ) )
wordlist = get_wordlist_via_set( "pc_woodchuck.txt" )
print( ", ".join( wordlist ) )
```

You may think of other solutions to the problem yourself. For instance, you can think of how you would use a dictionary with word counts to create an alphabetically ordered list of all the unique words.

Finally, a few words on tuples.

A *tuple* is an ordered data structure which is immutable. You may consider that a tuple is a list, but you cannot make changes to that list, e.g., you cannot sort a tuple. In general, almost any application for a tuple can just as well be done with a list, and a list can do much more than a tuple. The main exception is when you must use an immutable data structure somewhere. A good example is when you want to build a dictionary where each key consists of multiple values. Since the keys for a dictionary must be immutable, you cannot use lists for such keys; you must use tuples. However, this kind of application is pretty advanced, so you do not need to spend much thought on it at this time.

9.2 Building loops over sequences

Functionally, there two kinds of loops: a loop that moves through a sequence of items (such as the items in a list), and a loop which continues for an indeterminate amount of time, until a certain conditions occurs.

The first kind of loop is usually (but not necessarily) implemented by a **`for`**-loop. The other kind of loop is discussed in the next section.

If the sequence of items you have to process are in a sequential data structure (list, dictionary, set, or even tuple), you can just write **`for`** `<item>` **`in`** `<sequence>`, and the code in the body of the loop will be run for each item in the sequence. Occasionally, you may need the index of the item in the sequence, and in that case you usually write the **`for`**-loop as processing the indices with a **`range()`**. For example:

```
sequence = ['apple','pear','banana','grape','orange']

for item in sequence:
    print( item )
print()
for i in range( len( sequence ) ):
    print( sequence[i] )
```

When you run this code, you see that both loops produce the same output. The first loop, however, does not have access to the indices of the items in the list.

For the second loop, please note the following: the variable `i` takes on the values that are produced by **`range()`**. **`range()`** gets as argument the length of the sequence. The values that it produces are integers 0, 1, 2, ... up to but not including the length of the sequence. In

this case, the length of the sequence is 5 (it has five items), so the five values that **range**() produces here are 0, 1, 2, 3, and 4.

This neatly lines up with how indices work in Python, and actually in almost all programming languages: they start at zero and go up to (but not including) the length of the sequence for which you use the indices. This becomes even more clear when you try to do the same thing as the second loop does, with a **while** loop.

```
sequence = ['apple','pear','banana','grape','orange']

for i in range( len( sequence ) ):
    print( sequence[i] )
print()
i = 0
while i < len( sequence ):
    print( sequence[i] )
    i += 1
```

You can see that the **while** loop does exactly the same as the **for** loop, namely access the items of sequence by their index. Note that for the **while** loop, we start the index `i` at 0, and increase it until `i` gets equal to or becomes greater than the length of sequence.

This is the typical way of writing a loop which goes through items of a sequence by index. Start at index 0 and write a comparison which checks if this index is still smaller than the length of the sequence. Always write loops that process indices in this way. If you stick to this standard, you no longer have to spend any thought on whether you are counting one item too few or too many.

9.3 Building loops with an indeterminate end

If you have to end a loop when "some condition occurs," but you cannot tell up front how many times the loop code should be run, you must use a while loop. There are two main ways to write a while loop. The first you use if the condition is a fairly straightforward test.

For instance, suppose that I am searching for a number of 4 digits for which the sum of the digits is the same as the product of the digits (this was an exercise in a previous chapter). I have written two functions, `sum_of_digits()` and `product_of_digits()` that calculate the sum and product of the digits of a number, respectively. I can use them to write a while loop where the condition of the loop basically says: do this loop as long as you have not found a solution to the problem. I do not know exactly how many times I have to go through the loop, but once I found the solution, the loop ends (in fact, the body of the loop is executed 124 times).

```
def sum_of_digits( num ):
    strnum = str( num )
    total = 0
    for c in strnum:
        total += int( c )
    return total
```

```
def product_of_digits( num ):
    strnum = str( num )
    total = 1
    for c in strnum:
        total *= int( c )
    return total

num = 1000
while sum_of_digits( num ) != product_of_digits( num ):
    num += 1
print( num )
```

Now, there is a bit of a problem with this code. The problem is that, before I solved the problem, I did not know if there actually was a 4-digit number for which the sum of the digits equals the product of the digits. If such number does not exist, the loop is endless!

One way of solving this is using a logical operator. If extend the condition of the loop with an extra clause, namely that the number should not be higher than 9999. So I do the loop while I have not found a solution and the number is still smaller than 10000. This I can solve with a logical **and**. The main loop becomes:

```
num = 1000
while sum_of_digits( num ) != product_of_digits( num ) and num <
    10000:
    num += 1
```

While writing a loop with one or more logical operators in the condition works, such conditions tend to become unreadable quickly. That is why I often write them differently, namely by using **while True**, which means that the loop is endless, but then write conditions as the first statements in the loop which **break** out of the loop if certain conditions occur. This also tends to be easier to read as you can make positive rather than negative conditions, i.e., the loop ends when a solution is found, rather than the loop continues as long as the solution is not found. The following main loop does this:

```
num = 1000
while True:
    if num > 9999:
        break
    if sum_of_digits( num ) == product_of_digits( num ):
        break
    num += 1
```

The **while True** construction tends to be confusing to students, but it is not more than what I describe above: the conditions on which the loops ends are written at the top of the loop, with **break** statements to end the loop. It is also a good solution if you have to execute a function or do a calculation before you can test the condition. For instance, if you read lines from a file, and you should end the loop when there is nothing to read anymore, you first have to do a `readline()` before you can draw the conclusion that you are done. For example:

```
fp = open( "pc_woodchuck.txt" )
while True:
    buffer = fp.readline()
    if buffer == "": # There is nothing to read anymore
        break
    print( buffer, end="" )
```

9.4 Initializing and deinitializing

Often when you have to compute something, you need to do some initializing, like setting variables to an initial value, and some deinitializing, like using the computed values of the variables you initialized. I often see students having problems finding the right spot in the code where to initialize a variable, and where to use it.

If you know "the right spot" for one of them, you usually know the right spot for the other, as they tend be on the same indentation level. I.e., if I do some sort of calculation which produces a value in a variable, then I read out that variable right after the calculation, and the initialization of the variable almost always has to be at the same indentation level right before the calculation.

Let's give an example. Suppose that with a simulation I want to estimate the percentage of rolls of five six-sided dice that have a total value of 15 or more. This means I have to code a nested loop. Using bottom-up development, I first program the inner loop, where I roll five dice.

```
from random import randint

total = 0 # initialization
for i in range( 5 ):
    total += randint( 1, 6 )

print( total ) # deinitialization
```

When I want to calculate a total value for five dice which are rolled, I need to roll a single die five times, and add up the values. For that I need a variable in which to store the total value, which I call total. That variable is initialized at 0, and can be read out after the loop is done. In the code you can see that setting the value of total to zero and having the value of total be the total of five dice, is at the same indentation level.

Now, I said that I wanted to estimate what percentage of rolls end up with a total of 15 or higher. Let's say that I simulate rolling the dice 1000 times. That means that I have to build a loop around my previous code, which executes that code 1000 times. I need to keep track of how many of those rolls are 15 or higher. So I have to keep track of that using a second variable, which I need to initialize to 0 before the simulation, and which will have the desired value after the simulation. So the code becomes:

```
from random import randint
```

```
TRIALS = 1000

over15 = 0 # initialization 1
for j in range( TRIALS ):
    total = 0 # initialization 2
    for i in range( 5 ):
        total += randint( 1, 6 )
    if total >= 15: # deinitialization 2
        over15 += 1

print( over15/TRIALS ) # deinitialization 1
```

In this code I had to change the line where I printed the value of total to using this value to see if I need to increase `over15`, which keeps track of how many rolls were 15 or higher. This use of total is (of course) still at the same indentation level as the initialization of total. In the same vein, the initialization of `over15` is at the same level as the use of `over15`, in the last line of the code.

I often see students doing this wrong. Two typical errors that are made in the code that I developed above are initializing `total` too early and initializing it too late. If you initialize `total` too early, i.e., at the same spot where you initialize `over15`, `total` will never be reset to zero in the code and thus will be increased after every roll, which means that you end up with 0.999 or 1.000. If you initialize it too late, i.e., underneath the line `for i in range( 5 )`, you set total to zero before every individual die roll, and you end up with 0.000.

It is a rule-of-thumb that applies almost always that, as I said, initialization and deinitialization are at the same indentation level.

9.5 Capturing errors

During coding, try to capture errors. Anything that can go wrong, should lead to an error message. Even if you think that something can never go wrong, still program an error message. If it does not go wrong, you will never see the message and it won't take any processing time, but sometimes you will be surprised.

For instance, suppose you calculate a fraction, e.g., all numbers on a list of numbers that are bigger than 10. You know that the fraction should always fall in the range [0,1]. However, if you are going to use that fraction later in the program and it will lead to problems if the fraction is outside that range, it makes sense to at least build in a check right after calculating the fraction to make sure that it does fall in the range, and produce an error message otherwise.

You do that because, even if you are an experienced programmer and feel certain that you are able to calculate such a fraction correctly, you may have made a typo and produced the wrong result. Everyone makes such mistakes, so it makes sense to build in these checks regardless how experienced you are. You may need a few seconds to build it in, but it occasionally may save you hours of debugging.

Exercises

Exercise 9.1 You read a file with some contents, and you have to do something with those contents. In the following situations, which data structure would you use to store the contents of the file? Note: sometimes you may wish to use multiple data structures. Note 2: Sometimes there is a "best answer," but not always.

- The file contains a list of items ordered by a client. Each item is listed with a quantity and a price. You must determine the total cost of the order.
- The file contains a list of words that are legal in a particular language. You must use these words to spellcheck a text.
- The file contains a list of matches in an eSports event. For each match is listed which two competitors engaged in the match, and who of them won the match. You must use the match information to produce a ranked list of all the competitors, from best-scoring to worst-scoring.
- The file contains a list of bachelor courses with for each course the number of the course and the number of study points (ECTS). You have a series of files, each file for one student, which contains a list of the courses the student took (by course number), and their final grade for the course. You must determine which students are ready to receive their bachelor diploma (i.e., which students have a total of 180 study points of courses they passed).
- The file contains a text which has, in several places, links to other texts which must be inserted in the given text. You have to produce the text which has all these links replaced by the texts that are to be inserted.

Exercise 9.2 This exercise and the following exercises in this chapter all use a file which contains a dictionary of English words, stored in a file. A standard dictionary of this type is the file "wordsEn.txt," which can easily be found on the Internet.

The file "wordsEn.txt" contains a list of words, one word per line. Make a program that produces a list of all 5-letter words which contain an 'x'.

Exercise 9.3 Make a program that uses the file "wordsEn.txt" to produce a list of all 5-letter words of which the letters are in alphabetical order. E.g., the word "abort" has all its letters in alphabetical order as 'a' < 'b' < 'o' < 'r' < 't'.

Exercise 9.4 Make a program that uses the file "wordsEn.txt" to produce a list of all 5-letter words of which the letters are in alphabetical order and no letters are the same or right next to each other in the alphabet. E.g., the word "forty" meets the requirements as the letters are in alphabetical order and there is at least one other letter in the alphabet between any two of the letters.

Exercise 9.5 Make a program that uses the file "wordsEn.txt" to produce a list of all 5-letter words of which the letters are in alphabetical order and at least two pairs of letters are right next to each other in the alphabet. E.g., the word "ghost" meets the requirements as the letters are in alphabetical order and the 'g' is next to the 'h', and the 's' is next to the 't'.

Exercise 9.6 Make a program that uses the file "wordsEn.txt" to produce a list of all 5-letter words which also have an anagram that is a legal 5-letter word. E.g., "earth" and "heart" would both be on the list, as they are anagrams of each other. An extra nice solution shows the anagrams next to each other, so "crate," "cater," "react," "caret," and "trace" (and perhaps also "recta" and "carte," if they are in the dictionary) are all on one line.

Exercise 9.7 Make a program that uses the file "wordsEn.txt" to produce a list of all words (regardless their number of letters) of which the letters are in alphabetical order. However, words that are part of other words on the list should be excluded. E.g., the word "most" should not be on the list as it is part of the word "almost." There are many ways to approach this exercise. Once you have found one, see if you can make it more efficient.

Chapter 10

Recursion

Recursion entails that functions can call themselves to make certain problems easier to solve. While it is an advanced topic, I thought that I would still spend a chapter on it, to try to explain it in a way that makes intuitively clear what recursion is about and why it works.

10.1 Recursion in real life

In an earlier chapter I discussed sorting. An example I used was sorting books on bookshelves. I explained that a typical way that people do this is first distribute the books over piles, for instance based on letters of the alphabet. Let's look at this example in a bit more detail.

You have a large number of books, and you know that you can easily sort a pile of 10 books or less. You want the books on the shelves by the author's last name. So you begin by splitting the books over 26 piles, based on the first letter of the author's last name. Once you have done this, some of these piles will be 10 books or less; e.g., there may be no books at all on the pile marked by X, and just a few on the pile marked by U. However, many of the piles will be more than 10 books.

Now you repeat the procedure for all the piles with more than 10 books, now using the second letter of the author's last name. For instance, the pile marked by S may have more than 10 books, so you now make piles Sa, Sb, Sc, ..., Sz. Many of these piles will remain empty, some of them may hold 10 books or less, and maybe some of them will still have more than 10 books.

So you repeat the procedure with those piles which have more than 10 books, but now based on the third letter of the author's last name. For instance, the pile marked St will be divided over piles Sta, Stb, Stc, ..., etc.

When does the procedure end? The easy answer is that it will end when all piles have 10 books or less. There is a small problem with this answer, as it may be the case that there is a pile with more than 10 books which all have the same last name. If we assume that in that case we do not want to further split up the pile as we no longer need to further sort it (which may not actually be the case, as there may be multiple authors with the same last

name and we might still want to split it up further, but I will leave that possibility out), the answer is that it will end when all piles have either 10 books or less, or consist of only books with one and the same last name for the author.

You can imagine code for this as follows:

```
function sort_books( books ):
    divide books over piles based on first letter of author name
    for each of the piles do the following:
        if pile contains > 10 books and multiple author names:
            divide books over piles based on 2nd letter of name
            for each of the piles do the following:
                if pile contains > 10 books and multiple author names:
                    divide books over piles based on 3rd letter of name
                    ...etc.
place the books on the shelves in the order of the piles
```

A big problem with this code is that we do not know how many loops we need to nest for this. If the maximum length of an author's last name is 20, then 20 nestings would suffice. However, notice that we are doing almost the same thing in each loop implementation, namely:

- We check whether we need to further split up a pile, and
- If we need to further split up a pile, we do so on the next letter of the last name

So, you can imagine that you have a function that gets a pile of books, and a number of a letter of the last name, and splits up the pile.

```
function split_pile( books, nr_of_letter ):
    split books into piles based on nr_of_letter of author name
    for each pile:
        if pile contains > 10 books and multiple author names:
            split_pile( pile, nr_of_letter + 1 )
```

As you can see, this is a recursive call, as the function `split_pile()` calls itself.

This is how recursion works. You have a problem which you solve by turning it into one or more easier problems (a big pile of books is turned into a series of smaller piles of books). The easier problems need the same solution approach as the harder problem. If an easier problem is still too hard, you turn it into one or more easier problems again, which again need the same solution approach. You continue doing this until the problems have become so easy that they basically have solved themselves.

What is important is that this procedure is guaranteed to end at some point. In our case, the procedure will stop as soon as all piles are 10 books or less, or contain only books with the same last name. Because a pile can always be split up further if there are multiple last names in a pile, this procedure is guaranteed to end.

10.2 Guessing a number

In the chapter on top-down programming, one of the exercises asks you to develop a program which lets the computer guess a number, that you memorize. You have to tell the computer "higher," "lower," or "correct."

If you make this a smart program, it will try to exclude half the possible numbers with each guess. For instance, if the number is between 1 and 1000, the computer can first guess 500. If the actual number is lower, the range is 1–499 (499 numbers), and if it is higher, then the range is 501–1000 (500 numbers). Suppose that the number is higher, then the computer's next guess would be the middle of the remaining range, so 750. If the actual number is lower, then the range is 501–749. Next, the computer asks the middle of that range, so 625. Etc. This approach guarantees that the computer needs no more than the square root of the maximum number as guesses (10 guesses for the range 1–1000).

You can implement this in an iterative way. I.e., you keep track of the lower end and higher end of the range. The computer guesses the middle of the range. If the actual number is lower, the higher end of the range is moved to the middle (minus 1). If the actual number is higher, the lower end of the range is moved to the middle (plus 1). You continue doing this until the number is guessed or the range spans a single number.

Code for this could be the following:

```
def get_answer():
    while True:
        answer = input( "(H)igher, (L)ower, (C)orrect: " ).upper()
        if answer not in "HLC":
            continue
        return answer

print( "Memorize a number between 1 and 100" )
lo = 1
hi = 100
while True:
    if lo > hi:
        raise Exception( "You are trying to cheat." )
    mid = (lo + hi) // 2
    if lo == hi:
        break
    print( "I guess the number is", mid )
    answer = get_answer()
    if answer == 'C':
        break
    elif answer == "H":
        lo = mid + 1
    else:
        hi = mid - 1

print( "The number is", mid )
```

You can also think of implementing this recursively (though I hasten to add that that is not the preferred approach, for many reasons).

A "guess the number" function gets a range of possible numbers. It will guess the middle of the range. If that is correct, the function is finished and returns the guessed number. If it is not correct, if the actual number is lower, it will play "guess the number" (i.e., a recursive call) with the smaller range which is lower than the guessed number. If the actual number

is higher, it will play "guess the number" (i.e., a recursive call) with the smaller range which is higher than the guessed number. In both cases where the number is not correct, it will return the return value of the recursive call.

This procedure is guaranteed to end, as upon every recursive call the length of the range is halved, and if the range has a length of 1 the number is guessed (it may be possible that the range gets length zero at some point, if the human gave incorrect answers to the guesses).

In the following recursive implementation of the guess-the-number game, I use a list to store the numbers, and the list is halved on every recursive call. This is not an efficient way of coding as these lists take a lot of memory, but it seemed to me the clearest way to explain the example.

```
def get_answer():
    while True:
        answer = input( "(H)igher, (L)ower, (C)orrect: " ).upper()
        if answer not in "HLC":
            continue
        return answer

def guess_number( numbers ):
    if len( numbers ) < 1:
        raise Exception( "You are trying to cheat." )
    if len( numbers ) == 1:
        return( numbers[0] )
    mid = len( numbers ) // 2
    print( "I guess the number is", numbers[mid] )
    answer = get_answer()
    if answer == 'C':
        return numbers[mid]
    elif answer == "H":
        return guess_number( numbers[mid+1:] )
    else:
        return guess_number( numbers[:mid] )

print( "Memorize a number between 1 and 100" )
print( "The number is", guess_number( list( range( 1, 101 ) ) ) )
```

Exercises

Exercise 10.1 In the "guess the number" game, can you keep track of the number of guesses made without using a global variable? Hint: let the function return two values, namely the correct number and the number of guesses, instead of only one.

Exercise 10.2 A list is sorted if the first half of the list is sorted, and the second half of the list is sorted, and the last element of the first half is lower than the first element of the second half. And, of course, a list of only one element is sorted. This is a recursive definition. Write a recursive function that checks if a list is sorted.