LEARN PYTHON WITH JUPYTER

Develop computational thinking while learning to code

SERENA BONARETTI

Dear coder,

Thank you for your choosing *Learn Python with Jupyter*! I hope the book will help you develop computational thinking while learning to code in Python!

The **e-book** version of *Learn Python with Jupyter* is **open and free**, and it will remain open and free. The **printed** version is available on the **Amazon** shops: `.com`, `.co.uk`, `.de`, `.fr`, `.es`, `.it`, `.nl`, `.pl`, `.se`, `.com.be`, `.ie`, `.ca`.

You can read more about the **characteristics** of the book at `www.learnpythonwithjupyter.com` and in the Jupyter Blog post at `www.tinyurl.com/LPWJmedium`.

If you have any comments or questions, feel free to **email me** at serena.bonaretti.research@gmail.com. I'd be happy to hear from you. You can also leave feedback at `www.tinyurl.com/LPWJfeedback`.

Thank you for learning with me,
Serena

Learn Python with Jupyter

Develop computational thinking while learning to code

Serena Bonaretti

www.learnpythonwithjupyter.com

Main editing and proofreading by John Batson.
Additional editing and proofreading by Serena Bonaretti, with the support of ChatGPT.
Cover by Serena Bonaretti.

www.learnpythonwithjupyter.com

To Christof, who loves R

Eccoci nuovamente insieme per imparare a leggere e a scrivere.
Io direi, però, di più: per imparare a conoscere meglio il mondo e noi stessi.
Here we are again together to learn how to read and write.
Actually, I would go further: to learn to better understand the world and ourselves.
—Alberto Manzi, Non è mai troppo tardi, *It's never too late*

Simple is better than complex.
—Tim Peters, The Zen of Python

Content

Part 6. Overview of lists and the for loop			
Chapter	Computational thinking	Syntax	In more depth
21. Overview of lists *p.* 163	■ Arithmetic operations on list elements ■ List concatenation and replication ■ List assignment ■ Adding and removing list elements ■ List sorting and searching	■ List methods: `.clear()`, `.copy()`, `.count()`, `.extend()`, `.reverse()`, `.sort()`	*Why not use a* `for` *loop to remove list elements?* *p.* 172
22. Overview of the for loop *p.* 176	■ `for` loop as a repetition of commands ■ `for` loop through indices, elements, and indices and elements ■ List comprehension ■ Tuples ■ Nested `for` loop	■ Built-in functions `list()` and `enumerate()`	*Basics of Markdown* *p.* 184
23. Lists of lists *p.* 187	■ Lists of lists ■ Slicing lists of lists ■ `for` loop to browse lists of lists ■ Flattening lists of lists	■ (No new syntax)	*Digital images* *p.* 191
Part 7. Dictionaries and overview of strings			
Chapter	Computational thinking	Syntax	In more depth
24. Inventory at the English bookstore *p.* 197	■ Dictionary items, keys, and values ■ Slicing dictionary values ■ Modifying dictionary values ■ Adding and removing dictionary items	■ Dictionaries ■ Dictionary methods: `.items()`, `.keys()`, `.values()`, `.update()`, `.pop()`	*Lists of dictionaries* *p.* 202
25. Trip to Switzerland *p.* 206	■ Initializing an empty dictionary ■ Four ways to modify a dictionary value that is a list ■ `for` loop to browse dictionaries ■ Use of comma separation or `.format()` in `print()`	■ Dictionary method `.get()` ■ List method `.format()`	*Dealing with KeyError* *p.* 210
26. Counting, compressing, and sorting *p.* 213	■ Counting elements ■ Compressing information ■ Sorting dictionaries according to keys or values	■ Dictionary method `.get(key, initial value)` ■ Built-in function `sorted()`	*Remaining dictionary methods* *p.* 218
27. Overview of strings *p.* 220	■ "Arithmetic" operations on strings ■ Replacing or removing substrings ■ Searching and counting substrings ■ Converting strings to a list and vice versa ■ f-strings ■ Rounding numbers	■ String methods `.count()`, `.find()`, `.join()`, `.replace()`, `.split()`, and `.swapcase()` ■ Built-in function `round()`	*Escape characters* *p.* 230
Part 8. Functions			
Chapter	Computational thinking	Syntax	In more depth
28. Printing thank you cards *p.* 237	■ Function as a unit of code ■ Calling a function ■ Function inputs	■ Function definition ■ Keyword `def` ■ Function inputs: parameters and arguments, and default values ■ Docstrings for function definition and parameters in Numpy style ■ Function call	*Why is function documentation important?* *p.* 244

29. Login database for an online store *p. 246*	■ Function outputs ■ Modularization: Main function and satellite functions	■ Keyword `return` ■ Docstrings for function returns ■ Tuples	*What is None?* *p. 253*
30. Free ticket at the museum *p. 256*	■ Use of `if/else` construct to raise an error ■ Creation of conditions to check variable types and values ■ Return based on conditions ■ Return values	■ Built-in function `isinstance()` ■ Types `str`, `int`, `list`, `dict` ■ Keyword `raise` ■ Exceptions `TypeError()` and `ValueError()` ■ Example in docstring definition ■ Docstring for returned values	*How can I avoid interrupting the flow?* *p. 263*
31. Factorials *p. 265*	■ Iterative vs recursive functions ■ Recursive thinking ■ Base case and recursive case	(No new syntax)	*When do we use recursive functions?* *p. 269*
32. How can I reuse functions? *p. 271*	■ Function reuse ■ Creating, modifying, and structuring a module ■ Various ways to import a module ■ Package as a group of modules ■ IPython as a coding engine ■ Using complementary tools when coding (IDE, Jupyter Notebook, and terminal)	■ Keywords `as` and `from` ■ Module `sys` and its command `sys.path.append()` ■ Jupyter extension `autoreload`	*What is: if __name__ == "__main__"?* *p. 280*
Part 9. Last bits of basic Python			
Chapter	Computational thinking	Syntax	In more depth
33. Birthday presents *p. 285*	■ Opening and reading a text file ■ Creating and writing a text file ■ Calculating basic statistics ■ Organizing functions in pipelines	■ Keyword `with` ■ Built-in function `open()` with the parameters `"r"` and `"w"` ■ Variable `file` with its methods `.read()` and `.write()` ■ Built-in functions `min()`, `max()`, and `sum()`	*How do I organize folders and files?* *p. 292*
34. What's more in Python? *p. 295*	■ Using sets as intermediators for list operations ■ Creating anonymous functions ■ Reproducing random numbers ■ Calculating computational time	■ Tuple methods `.count()` and `.index()` ■ Sets and their methods `.union()` and `.intersection()` ■ Keyword `lambda` ■ Built-in function `map()` ■ Function `.seed()` from the module `random` ■ Module `time` and its function `.time()`	*What is pip install?* *p. 303*
Part 10. Object-oriented programming			
Chapter	Computational thinking	Syntax	In more depth
35. Let's build an online store! *p. 307*	■ Procedural programming vs. object-oriented programming ■ Classes and objects ■ Attributes and methods	■ Keyword `class` ■ Built-in methods `__init()__` and `__str()__` ■ Dot notation to call attributes and methods	*Python data types are classes!* *p. 316*

About this book

What will I learn in *Learn Python with Jupyter*? In this book, you will learn to code in Python using Jupyter Notebook. Even more importantly, you will develop computational thinking, which is the way we think when coding.

What makes *Learn Python with Jupyter* different? Learning a programming language is, in many respects, learning a new *language*. When we study a new *spoken* language—such as Italian or Korean—we usually use two types of books. We learn with a *course* book that shows us how to handle everyday situations—such as greeting people or ordering food at a restaurant—and guides us in learning a new way of thinking, which is typical of that language and culture. In parallel, we refine our knowledge of the language syntax using a *grammar* book, where each chapter focuses on a specific part of speech—such as nouns or adjectives—listing detailed rules with brief examples. Most books for learning programming languages follow the grammar-book approach, that is, each chapter explains a separate element of syntax—such as mathematical operations or the `for` loop—providing little guidance on how to combine coding elements. *Learn Python with Jupyter*, on the other hand, is designed as *course* book that helps you develop computational thinking while learning to code in Python.

Is *Learn Python with Jupyter* for me? If you have never coded before, if you are following online courses or videos but feel that you struggle to connect fragmented concepts, or if you need to better structure your Python and coding knowledge, this book is for you. Also, if you are training to become a scientist but are not very strong in coding, if you are transitioning to the Jupyter/Python environment from another programming language, or if you are a teacher looking for material, this book can be for you.

How is *Learn Python with Jupyter* structured? The book is divided into 12 parts. The first two parts provide an introduction to coding and the computational environment we will use—that is, the Jupyter/Python environment. In the following ten parts, you will learn how to think computationally and write Python code. Each of these ten parts contains two to five chapters, for a total of thirty-eight chapters. The topic progression is designed to support you in developing computational thinking while focusing on syntax and strategies, progressing from concrete concepts to more abstract reasoning.

How are chapters structured? Each chapter starts with a coding example embedded in a story, followed by thorough explanations focused on computational thinking and syntax. A few exceptions are the chapters that introduce basic concepts—such as Chapters 1 and 7—or those that summarize and extend what has been explained up to that point—such as Chapters 21 and 27. All chapters contain several theory and coding exercises. They finish with a recap summarizing the chapter's main concepts and an *In more depth* sections with coding strategies or curiosities.

Why is code embedded in stories? Stories provide context and favor long-term memorization. They are extensively used in learning foreign languages, and, in many respects, a programming language is a foreign language. Moreover, stories—rather than mathematical examples—can better support learners from non-scientific backgrounds.

Why learning with Jupyter notebooks? Jupyter notebooks are among the best available tools for combining code with stories—or, more technically, with narrative. They are also enjoyable to work with, visually appealing, and user-friendly as they run in a web-browser, which is a familiar environment for anyone using a computer.

Why is there code pronunciation? When we code, we pronounce or mumble code within ourselves, and occasionally aloud with a colleague. Although coding has a strong vocal component, there is no standard for code pronunciation. The pronunciation proposed in this book is the optimized result of more than a thousand hours of one-on-one interaction with students of various mother tongues.

What kinds of exercises are in *Learn Python with Jupyter*? In this book, you will find more than 700 theory and coding exercises. Theory exercises are meant to strengthen code comprehension and syntax precision, whereas coding exercises are meant to make you practice and thus learn by doing. All exercise solutions are on `www.learnpythonwithjupyter.com`.

What is on the website? On `www.learnpythonwithjupyter.com`, you can find the Jupyter notebooks associated with each chapter, so you can experiment while learning. You can also register to a portal where you can not only find the exercise solutions but also propose alternative solutions and ask questions. Finally, you can find work derived from this book and some additional material.

How do I use *Learn Python with Jupyter* to learn coding? Set up the computational environment as explained in the *Getting ready* part and get familiar with it. Then, proceed with the chapters. For each chapter, download the corresponding notebook at `www.learnpythonwithjupyter.com`. While reading the chapter, make sure that you understand the syntax and computational thinking, play with the code in the notebook, and complete the theory exercises to strengthen your comprehension. Then, read the recap to revisit the main concepts introduced in the chapter and the *In more depth* section, which will give you useful hints. Finally, take your time to solve the coding exercises. Avoid looking at solutions in the portal before completing an exercise, as this reduces your learning. Ultimately, be aware that missing the understanding of one chapter may compromise your understanding of the following chapters.

How is the language used in *Learn Python with Jupyter*? The language is colloquial, engaging, and simple—but precise. Great care has been taken to provide clear examples, precise definitions, and thorough explanations.

Why is *Learn Python with Jupyter* an inclusive book? It is well known that biases exist across genders, social classes, and geographical areas regarding who learns to code. In this book, every effort has been made to promote inclusivity, from the color palette, which avoids traditionally masculine or feminine tones, to the use of names and nationalities that are diverse and globally representative.

Why is there a free electronic version of *Learn Python with Jupyter*? Education should be a right for everyone. However, many people in the world lack access to education because of limited financial resources. To allow access to knowledge, the electronic version of *Learn Python with Jupyter* is and will remain free. The printed version is for those who prefer having a physical copy of the book and wish to support the author's work.

INTRODUCTION

In this short part, we will talk about coding environments, language syntax, and computational thinking. If you are eager to start coding, just skip it and come back later!

What do we need to learn when learning to code?

Coding is a lot about telling a computer what to do. We, human beings, need to write commands that computers understand, and to do so, we need to learn to think differently. We have to start from scratch and master a new way of communicating, consisting of *concise* and *logical* instructions. To be able to write these instructions for a computer, we need to learn at least three things: a coding environment, a language syntax, and computational thinking. Let's see what they are!

A **coding environment** is the place **where we can write and execute code**. There are several environments for coding in Python. In this book, we will use the Jupyter environment, which, since its release in 2015, has become increasingly used both in industry and academia (Figure 1 (left)). It allows integrating code with narrative and is ideal for learning to code, writing code interactively, and analyzing and visualizing data. Other very common coding environments are integrated development environments (IDEs), which typically include various components, such as a script editor, a variable environment panel, and a console where code is tested and executed—do not worry if you do not understand all these terms yet, they will become clear. For Python, popular IDEs include Spyder—represented in Figure 1 (middle) and with which you will become familiar in Chapter 32—Visual Studio Code, and PyCharm. Finally, the most basic environment is Python IDLE, which is included in the Python installation. It consists of a shell—which looks very similar to a terminal—where one can type and execute commands (Figure 1 (right)). We will not use it in this book.

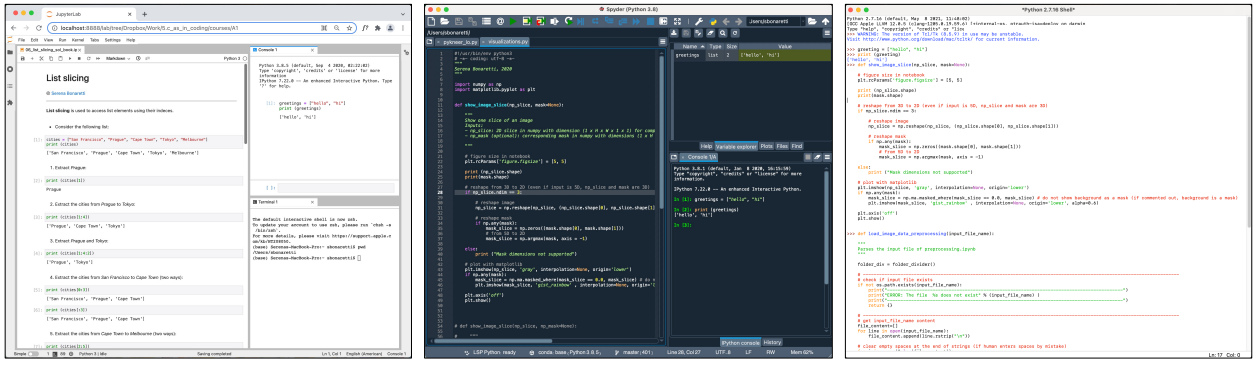


Figure 1. Three environments to code in Python: the Jupyter environment (left), Spyder (middle), and Python IDLE (right).

A **language syntax** is a **set of rules defining how to write commands**. You are already very familiar with at least one syntax, which is your native language syntax. In your mother tongue, you know words, punctuation, and how to arrange these elements in sentences to create paragraphs and entire texts. In coding, the pattern is similar. We need to know data types and operators, how to arrange them in `if/else` constructs and loops, and ultimately combine these elements into functions and classes. In Table 1, you can see a schematic summary of the elements and syntax that you will learn in this book. Don't worry if you do not understand most of it—everything will become more and more clear as we progress through the book.

Data types (*Words*)	Operators (*Punctuations*)	Constructs and loops (*Sentences*)	Unit of code (*Paragraphs*)	Software (*Texts*)
String, list, integer, float, Boolean, tuple, dictionary, set	Assignment, membership, arithmetic, comparison, logical	`if/else` construct, `for` loop, `while` loop	Functions	Classes (object-oriented programming)

Table 1. Components of a programming language, from the most basic (left) to the most complex (right). In the column titles, the words in between parentheses show the parallelism with the syntax of a natural language.

Finally, **computational thinking** is the **way we think when coding**. Every time we approach a new subject, we need to learn *how to think* in that subject and develop specific skills. Some of the abilities that you will develop in this book are:

- *Divide and conquer*, which consists of decomposing a problem into sub-problems, solving the sub-problems, and combining the sub-problem solutions to into the solution of the main problem.
- *Pattern recognition*, which means recognizing in a new problem features of a previously solved problem so that you can apply a similar solution.
- *Solution generalization*, which consists of generalizing solutions from specific cases to broader situations.
- *Creating algorithms*, which means conceiving and implementing a series of sequential instructions to solve a problem.

In the next part *Getting ready*, you will download, install, and learn how to use the Jupyter/Python environment. Then, starting from Chapter 1, you will begin developing computational thinking while learning Python syntax. Let's start this exciting journey!

GETTING READY

In this part, we will set up the Jupyter/Python environment and learn how to use it. Let’s start this exciting journey!

The Jupyter/Python environment

An easy way to think about the Jupyter/Python environment is to consider it as a Russian doll—those wooden dolls of decreasing size nested one inside another (Figure 2). The largest doll is **JupyterLab**, a web-based environment in which we can open, organize, and work on files of various types. In JupyterLab, there is **Jupyter Notebook**, a web-based application where we can write code with narrative. Jupyter Notebook supports several programming languages, one of which is **Python**. And finally, Python is enriched by an extraordinary amount of **modules** and **packages** that allow us to add useful functionalities to code. Let's access the Jupyter/Python environment and see how it works!

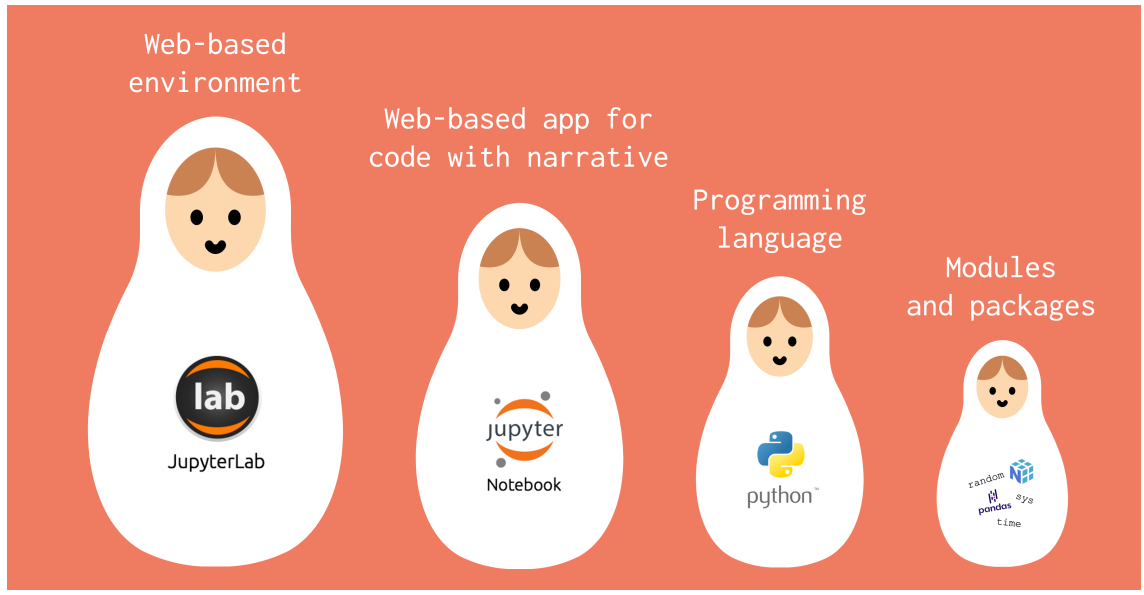


Figure 2. The Jupyter/Python environment represented as a Russian doll, where each component includes in the following ones.

Accessing the Jupyter/Python environment

You can access JupyterLab, Jupyter Notebook, Python, and some of Python's common packages in at least three different ways. The first way does not require any installation but you need to be connected to the internet while coding, whereas the other two ways require an installation but you do not need to be connected to the internet while coding. Choose the way that fits you the best!

1. *The very easy way*: *JupyterLite*. Open your browser and go to `https://jupyter.org/try-jupyter`. That's it, you are in the Jupyter/Python environment! As mentioned before, you do not need to install anything, but you have to be connected to the internet.

2. *The easy way*: *Anaconda*. Go to the Anaconda website, `www.anaconda.com/products`, and click *Download*[1]. Once downloaded, install Anaconda like any other software: click *Next* when required, and leave the default options (unless you have specific requirements). The installation might take a few minutes. When Anaconda is installed, open the Anaconda Navigator by double-clicking its icon, which

[1]Websites and download procedures are subject to frequent changes. If the procedure described here has changed, refer to the instructions provided on the website

looks like the icon in Figure 3, box 1. Once opened, you will see all the software contained in Anaconda, including JupyterLab (Figure 3, box 2).

3. *The advanced way: pip install.* Open the terminal on your computer and execute the command `pip install jupyterlab`. Let the installation do its work!

Now that you have access to JupyterLab, let's learn how it works!

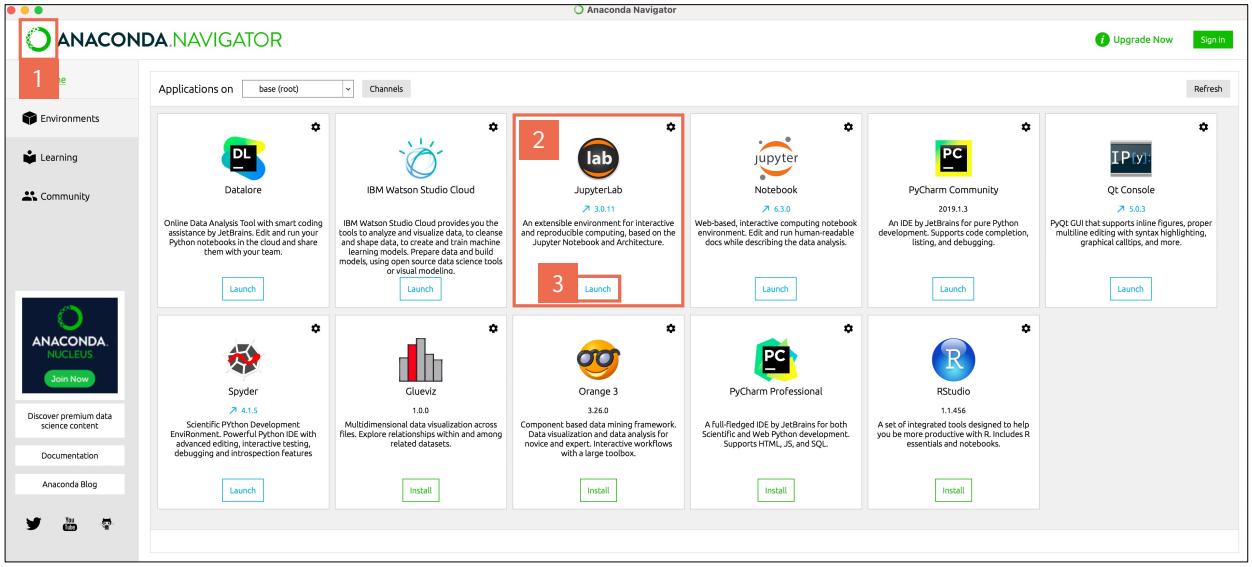


Figure 3. Anaconda interface: (1) icon, (2) JupyterLab, and (3) JupyterLab launch button.

JupyterLab

JupyterLab is an environment where we can code in an organized and efficient way. If you are using JupyterLite, you are already in JupyterLab. If you installed Anaconda, open JupyterLab by clicking the *Launch* button in the JupyterLab tile in Anaconda (Figure 3, box 3). If you installed JupyterLab using the terminal, run the command `jupyter lab` to launch it. In all cases, you should see something similar to Figure 4.

Here are the most relevant features of JupyterLab and some suggestions on how best to use them:

- *JupyterLab is a web-based environment.* JupyterLab runs in your browser. If you are using JupyterLite, you are connected to the internet and the address shown is the one you used to open JupyterLite. If you installed JupyterLab using Anaconda or via the terminal, you will notice that the address contains `localhost` (Figure 4, box 1), which means that you are working locally, that is, on your computer, and not to the internet.
- *Top bar* (Figure 4, box 2). The items in the top bar, such as *File*, *Edit*, *View*, etc., are quite intuitive and similar to many other software. We will describe the most relevant items throughout the book, but feel free to start exploring them now! Notice that when clicking some top bar items (for example, *File*), some of the items that appear might be light gray because they are disabled (for example, *Save As..*). This is because they refer to Jupyter Notebook, which we will describe in the next section. Finally, a fun feature of JupyterLab is that you can set a dark theme by going to *Settings*, then *Themes*, and then *JupyterLab Dark*.

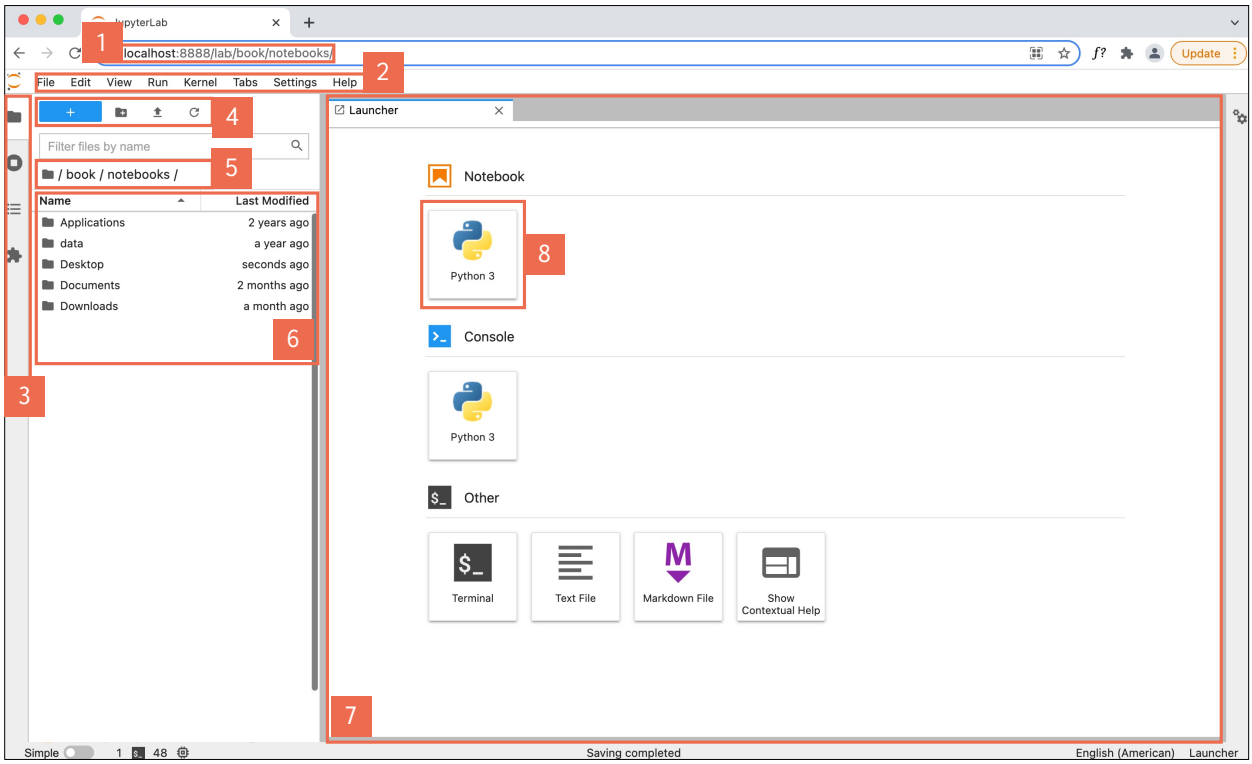


Figure 4. JupyterLab interface, containing: (1) local URL, (2) top bar, (3) lateral tabs, (4) folder browser top bar, (5) folder browser, (6) folder content, (7) launcher, and (8) Jupyter Notebook launch button.

- *Browsing and opening files.* On the left side of JupyterLab, you can find a panel with a few vertical tabs (Figure 4, box 3). When you click a tab, a panel opens on right; when you re-click, the panel closes—having a closed panel can be convenient on a small screen. Let's click the top tab, the one with the folder icon, to open the file browser panel. At the top of this panel, there is a top bar (Figure 4, box 4), containing four buttons: a + to open the launcher (Figure 4, box 7); a folder with a + to create a new folder; an upward arrow to upload a file; and a circular arrow to refresh the content of the current directory—in coding, we often say **directory** instead of folder. Below the top bar, there is a box to search for files, and then the path of the **working directory** (Figure 4, box 5)—that is, the folder where we open and save files. Finally, there is a panel showing the directory content (Figure 4, box 6). In JupyterLab, we can open an existing file only from this panel, and not by double-clicking a file in a computer folder. This is why it's important to know how to navigate folders within JupyterLab. To go back to a previous folder, we click the folder name (for example, *book* in Figure 4, box 5). To go into a sub-folder—a folder inside the current folder—we double-click on the sub-folder listed in the panel (such as *data* in Figure 4, box 6).
- *Launching tools.* The launcher is where you can open new notebooks, consoles, terminals, text files, etc. (Figure 4, box 7). As an alternative, you can open new files and tools from the top bar (Figure 4, box 2) by clicking *File*, then *New*, and then selecting the file type you want. Now it's time to open a Jupyter Notebook!

Jupyter Notebook

To open a Jupyter Notebook, go to the launcher and click the notebook icon (Figure 4, box 8). A new file named `Untitled.ipynb` appears in the browser panel (Figure 5, box 1). The file extension `.ipynb` stands for interactive python notebook. To give the notebook an appropriate name, right-click on `Untitled.ipynb` in the browser panel. Then, click *Rename*, and choose any name you want—for example, `practicing_cells.ipynb`. As you might have noticed, by right-clicking on the file name, you can perform several other common file actions, such as delete, cut, copy, duplicate, and more.

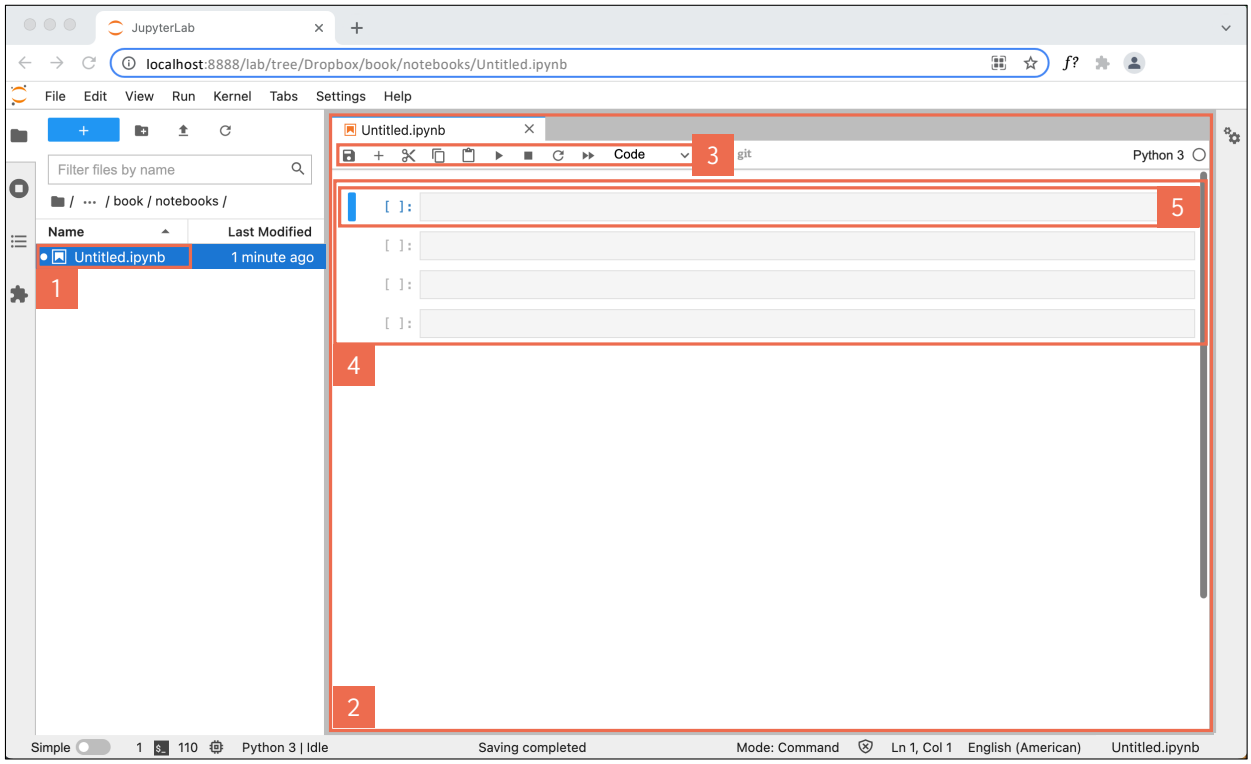


Figure 5. A Jupyter Notebook opened in JupyterLab. (1) Notebook in the folder content panel, (2) Jupyter Notebook, (3) Jupyter Notebook top bar, (4) cells, (5) currently active cell.

The file content is visible in the launcher area (Figure 5, box 2). A Jupyter Notebook is essentially a file containing a sequence of **cells**, that is, grey rectangles like the ones you see in Figure 5, box 4. Each cell can contain code or narrative, as we will see in a bit. The **blue bar** on the left side of a cell (Figure 5, box 5) indicates that the current cell is the **active cell**. In the presence of multiple cells, we can make a cell active by clicking on the square brackets [ ] on the cell left side. When a cell is active, we can perform several operations in various ways, either by keyboard commands, via the notebook top bar (Figure 5, box 3; enlarged in Figure 6), the JupyterLab top bar (Figure 4, box 2), or by right-clicking in the cell. All these possibilities might look redundant, but they are available for coders with different habits—some prefer using keyboard commands, others prefer clicking with the mouse. If there are too many options for you, just choose one and stick to that! Here are some useful cell operations and how to perform them:

- *Creating a cell*: To create a cell below the active cell, press B, for below, or the plus button in the notebook top bar (Figure 6, item 2). The newly created cell becomes the active cell. We can also create a new cell above the active cell by pressing A, for above (there is no corresponding top bar

button).

- *Deleting a cell*: To delete the active cell, press D twice, or click on the scissor button (Figure 6, item 3).
- *Copying a cell*: To copy the active cell, first press C and then V (without *Command* or *Control*!), or item 4 in Figure 6 to copy, and then item 5 to paste.
- *Undoing or redoing cell operations*: To undo a cell operation (for example, if you deleted a cell by mistake), press Z, or go to *Edit*, and then *Undo cell operation* in JupyterLab top bar (Figure 4, box 2). Similarly, to redo a cell operation, simultaneously press *Shift* and Z, or go to *Edit* and then *Redo cell operation* in JupyterLab top bar.
- *Moving cells*: Left-click on the square brackets [ ] of the active cell, and while holding down the mouse button, move the cell up or down. Release when you reach the wanted position. As an alternative, go to *Edit* in the JupyterLab top bar (Figure 4, box 2) and then click on *Move Cells Up* or *Move Cells Downs*.
- *Add line numbers*. Line numbers are very useful when coding—we will refer to the code by its line number throughout the whole book. To add line numbers, go to *View* in the JupyterLab top bar (Figure 4, box 2), and then click *Show Line Numbers*.
- *Other operations*. You can split or merge cells, enable or disable scrolling for output, and more by going to the JupyterLab top bar (Figure 4, box 2), and then see the options in *Edit*, or by right-clicking inside a cell and browsing the options that appear. Explore them!

Figure 6. Jupyter Notebook top bar: (1) save notebook, (2) add cell, (3) cut cell, (4) copy cell, (5) paste cell, (6) run cell, (7) interrupt kernel, (8) restart kernel, (9) restart kernel and run whole notebook, and (10) define cell as code or markdown. The button (6) is described in Chapter 6, whereas the buttons (7) to (9) are described in Chapter 7. All the other buttons are described in this chapter.

And finally, time to talk about cell content! As we mentioned before, a cell can contain two things: code or narrative. By default, Jupyter Notebook cells are code cells. To transform a cell into a **text cell**, press M on the keyboard, or click the drop-down menu in the Jupyter Notebook top bar (Figure 6, item 10), and select *Markdown*. **Markdown** is a simplified version of HTML, the coding language used to create websites. This is why the Jupyter environment is web-based: to use the rich features of web browsers! Writing the narrative in a notebook is fundamental to embedding code into explanations that make workflows easy to understand. You can learn how to write in Markdown in the *In more depth* section in Chapter 22. And last but not least, cells can contain **code**. The remainder of the book will be about that.

Downloading and working with the book material

Throughout the rest of the book, you will find 38 chapters. For each chapter, there is a Jupyter Notebook, whose file name begins with the corresponding chapter number. Each notebook contains the code discussed in the text so that you can practice and follow along as you read. Let's see how you can download and work with the notebooks:

- *Downloading the notebooks*: Go to `www.learnpythonwithjupyter.com` and download the notebooks. Save them in a new folder called `Notebooks`.
- *Opening the notebooks in JupyterLab*: If you are using JupyterLite, go to the top bar of the file browser panel (Figure 4, box 4), click the icon to upload files, represented by an upward arrow (third icon from the left), and upload the notebooks. If you are using JupyterLab installed locally, just navigate to the folder `Notebooks` using the folder browser (Figure 4, box 5). In either case, click the notebook's file name to open it.
- *Saving the notebooks*: If you are using JupyterLite, go to *File* and then *Download* to save your work. If you are using JupyterLab installed locally, just click the *Save* button in the top bar (Figure 6, box 1).
- *Creating notebooks for coding exercises*: In each chapter of the book, you will find coding exercises. Create a separate folder called *Exercises*, and as you progress through the book, add notebooks with your solutions to this folder. Creating your own notebooks will strengthen your organizational skills and allow you to become more familiar with the Jupyter/Python environment.

Finally, if you feel like going a step further, create your folders on a **cloud service** to ensure you do not lose your files if your computer breaks or has issues—yes, computers are machines and they can break! You can use cloud services such as Google Drive (`www.google.com/drive`), Dropbox (`www.dropbox.com`), or any others that you prefer. Using these tools is straightforward. Download and install the program on your computer. Once installed, you will see a new folder. Inside this folder, create `Notebooks` and `Exercises` and add your notebooks there. All your files will then be automatically synchronized and saved in the cloud.

At this point, we are really ready. Let's start coding!

PART 1
CREATING THE BASICS

It's finally time to write code! In the two chapters of this part, you will learn the basic elements that we will use throughout the book. You will learn about strings—that is, a data type that contains text—and the concatenation operation, which is used to combine strings. You will also learn how to ask questions and how to print information. And, most importantly, you will learn what a variable is. Let's go!

1. Text, questions, and art

Strings, input(), and print()

In this first chapter, you will learn three basic elements of code writing that we will use throughout the book: how to represent text in Python, how to ask a question to a person, and how to provide information. Ready? Open Jupyter Notebook 1 and follow along!

1. Writing text: Strings

In coding, we use the word string to refer to text. We can define strings as follows:

Strings are **text** in between quotes

Let's look at the two examples below. On the left side is the code from the first cell of Jupyter Notebook 1. On the right side is the corresponding pronunciation. Read the code out loud:

```
[]:  1  "This is a string"
```

```
This is a string
```

```
[]:  1  'Everything you write between quotes is a
        string'
```

```
Everything you write between quotes
is a string
```

Now consider the following statements. Are they true or false?

✎ True or false?

1. A string contains text. T F
2. A string is in green in Jupyter Notebook. T F
3. Quotes can be either single or double. T F

Computational thinking and syntax

Let's analyze the code above in detail! In each cell, there is a string containing text. By text, we mean any **character** we can type on the keyboard: letters, numbers, symbols, and even the space! Quotes can be **double quotes** `" "`, like in the top example, or **single quotes** `' '`, like in the bottom example. We must make sure you do not mix them: if we start a string with an opening double quote, we must end it with a closing double quote; if we start with an opening single quote, we must end the string with a closing single quote. Strings are a **data type** and are one of the core elements of the Python language. In Jupyter Notebook, Python strings are in **red**.

Let's run the first cell. **Running a cell** means **executing the code** in that cell. In the notebook, position the mouse anywhere inside the cell. If you haven't done it already, click the mouse left button. The cursor will become a blinking vertical bar. Then, move to the keyboard. If you are on a macOS, press *Shift* and *Return* at the same time. If you are on a Windows, press *Shift* and *Enter* at the same time—if

not explicitly labelled, *Enter* is the key on the right side of the keyboard with an angled arrow. As an alternative, you can click the *Start* button in the Jupyter Notebook top bar (Figure 6, box 6, in the part *Getting ready*).

This is how the first cell looks after running it:

```
[1]: 1 "This is a string"
'This is a string'
```
This is a string

When we run a cell, two things occur. First, a number appears in between the square brackets on the left side of the cell. In this case, the number is 1 because this is the first cell we ran. Second, we execute the code. In this case, we get to see the content of the cell; that is, `'This is a string'`. Jupyter Notebook shows the string in between single quotes, even when the string is written in between double quotes. As mentioned above, single and double quotes are equivalent.

Let's run the second cell. Like before, left-click anywhere inside the cell. Then, press *Shift* and *Return* if on macOS, or *Shift* and *Enter* if on Windows, or click the *Start* button in the Jupyter Notebook top bar. Here is what we get:

```
[2]: 1 'Everything you write between quotes is a
       string'
'Everything you write between quotes is a string'
```
Everything you write between quotes is a string

The number 2 appeared in between the square brackets on the left side of the cell, showing that this is the second cell we ran. As is becoming clear, the **number on the left** side between square brackets indicates the **order of execution** of the cells. Finally, we see the string contained in the cell: `'Everything you write between quotes is a string'`.

2. Asking questions: input()

In all programming languages there are ways to ask questions to a person, whom we usually call the **user**. This is a very important feature because it allows the interaction between a computer and a human being. What does this mean? Let's look at the code! Read the two cells below out loud (pronunciation on the right):

```
[]: 1 input("What's your name?")
```
input What's your name?

```
[]: 1 input("Where are you from?")
```
input Where are you from?

What does the code inside the cells do? Get a first hint by solving the following exercise.

✎ Match the sentence halves

1. "What's your name?" is	a. it is colored green.
2. input() is a built-in function and	b. by round brackets.
3. When running a cell containing input()	c. a string.
4. A built-in function is always followed	d. we can answer a question.

Computational thinking and syntax

Let's understand how these lines of code work! Let's run the first cell. We will get a text box:

```
[*]: 1 input("What's your name?")
     What's your name?
```
input What's your name?

Type your name in the rectangle (I will write mine!):

```
[*]: 1 input("What's your name?")
     What's your name? Serena
```
input What's your name?

And now press *Return* or *Enter* on the keyboard. Here is what happened (you will see your name, of course!):

```
[3]: 1 input("What's your name?")
     What's your name? Serena
     'Serena'
```
input What's your name?

The number on the left side of the cell turned to 3 as expected. But while answering the question, instead of the number 3, there was a **star symbol** (`*`). This indicates that a cell has started to run but has not finished yet. To complete the cell run and execute the code, we have to press *Return* or *Enter* after typing the answer. If the cell **run is not completed**, the **code in the cell will not execute** and the **following cells cannot be run**. Now, let's look at the code. We know that `"What's your name?"` is a string because it is text between quotes and it is colored red. What about **`input()`**? It allows us **to ask a question** and creates a **text box**—that is, a white rectangle—where a user can insert some text. `input()` is a **built-in function**, which, for now, can be defined as follows:

A **built-in function** is a command that performs a **specific task**

We can recognize if a code element is a built-in function by two characteristics. First, in Jupyter Notebook built-in functions are always green. Second, they are always followed by **round brackets ()**—in this book, we will call them round brackets instead of parentheses to differentiate from other types of brackets that we will encounter in the following chapters. In between the round brackets, we often write an **argument**—this terminology will become clearer when you learn about functions. For `input()`, the argument is a **string** containing the question we want to ask. Finally, built-in functions are very useful, as they contain code written by the creators of a programming language to facilitate common tasks and make coding more efficient.

Let's run the next cell:

```
[*]: 1 input("Where are you from?")
     Where are you from?
```
input Where are you from?

Enter your country of origin in the text box (I will type mine!):

```
[*]: 1 input("Where are you from?")
     Where are you from? Italy
```
input Where are you from?

Now press *Return* or *Enter* on the keyboard. You will see an output similar to the following (you will see your country of origin!):

```
[4]:  1  input("Where are you from?")
      Where are you from? Italy
      'Italy'
```

input Where are you from?

What happens here is similar to what happened in previous cell 3. The built-in function `input()` shows the question—that is, the string—and creates a text box in which we can type the answer. After typing the answer, we press *Return* or *Enter* to complete the code execution. In a more technical way, we can say that the built-in function `input()` creates a text box in Jupyter Notebook in which we can answer the question contained in the string we give as an argument.

One question before continuing: where do you think we see `input()` in action in everyday life? Every time we are asked to type something on a device, there is a function similar to `input()` behind the scenes! For example, when we write our name to register for a newsletter, enter the amount we want to withdraw from an ATM, or fill out an online form.

Finally, it is important to mention that when we write code, **we wear two hats**—that is, we have two roles: we are at the same time coder and user! While writing code, we wear the **coder hat**: we create code to perform a task, design the code structure, and define user messages. While testing code, we wear the **user hat**: we check whether the code does what is expected, is easy to use, and whether the user interaction is pleasant. When coding, we switch hats continuously!

3. ASCII art: print()

We now know how to ask a question to a user, but how do we provide them with information? Look at the following cell containing some ASCII art, a form of digital art by which we create images using the symbols on a keyboard:

```
[]:  1  print("/\_/\  ")
     2  print(">^.^<  ")
     3  print(" / \   ")
     4  print("(___)__")
```

What do you think will be printed to the screen? The answer is straightforward, but before running the cell, analyze the code by completing the following exercise.

✎ True or false?

1. print() is a string. T F
2. print() can have a string as an argument. T F
3. In coding, we print row by row. T F

Computational thinking and syntax

Let's run the cell. Here is what we get:

[5]:
```
print("/\_/\  ")
print(">^.^<  ")
print(" / \   ")
print("(___)__")
```
```
/\_/\
>^.^<
 / \
(___)__
```

The little cat we created using keyboard symbols gets displayed to the screen. To do so, we used the **built-in function `print()`**, which **displays on screen the argument** we provide — in this case a string. You might ask: But when we ran the cells 1 and 2, we could see the content of the strings; why do we need `print()`? The fact that we could see the strings from cells 1 and 2 is a feature of Jupyter Notebook. After running a cell, **Jupyter Notebook displays the content of the last line** but not the content of the previous lines. If we delete the `print()` function from the code in cell 5, Jupyter Notebook will display only the very last string at line 4:

[5]:
```
"/\_/\  "
">^.^<  "
" / \   "
"(___)__"
```
```
'(___)__'
```

As you may have noticed, in a Jupyter Notebook cell, we can write **several lines of code** and these lines will **be executed sequentially**. In other words, when we run a cell, Python first executes line number 1, then line 2, and so on, until the last line of the cell is reached. Thus, we print to the screen row by row, exactly like a printer! In addition, **in a string, spaces matter**. Spaces are **characters**, so a space is an element of a string and occupies its own place. On the other hand, spaces do not matter between code elements. For example, both lines below are executed without error messages:

[5]:
```
print ("(___)__")
print( "(___)__" )
```
```
'(___)__'
'(___)__'
```

However, the Python style guidelines recommend **avoiding a space between a function and the subsequent round brackets**.

When writing code with some repetition, it is good practice to keep some **parallelism** between the lines of code. Compare the code written in cell 5 as we did above:

[]:
```
print("/\_/\  ")
print(">^.^<  ")
print(" / \   ")
print("(___)__")
```

to the same code written without aligning closing quotes and closing round brackets, as below:

```
[]:
1 print("/\_/\")
2 print(">^.^<")
3 print(" / \")
4 print("(___)__")
```

We can see that, in the second case, the code looks somehow harder for a human to read. Instead, when we align quotes, brackets, and other symbols, we create code that is more **readable** and **less prone to errors**. In the following chapters, you will learn more tricks to minimize the amount of errors when writing code.

One last question before the recap: where do you think we see the built-in function `print()` in action in everyday life? Every time we see a message on a device! For example: 'Registration completed', or 'Thank you for your purchase', or 'Logout successful'. In the underlying code, there is a function similar to `print()`!

Recap

- The type string is text between quotes.
- `input()` is a built-in function to ask a user to enter a value.
- `print()` is a built-in function to display a value to screen.

Our fingers have memory

When learning to code, it is very important to **type every single command**, resisting the temptation of copying/pasting. Typing helps us **memorize** commands in at least two ways. First, when typing a command we mentally spell it, so we repeat it in our minds, and thus we memorize it. Second, our fingers can memorize typing patterns. For example, when typing `print()`, our fingers will automatically remember to type the round brackets right after `print`. Similarly to a pianist who does not look at the keyboard but at the sheet music while playing, we want to look not at the keyboard but at the screen while coding. This way of typing is called **touch typing** (or blind typing). It helps us be faster and **minimize the amount of errors** we make because

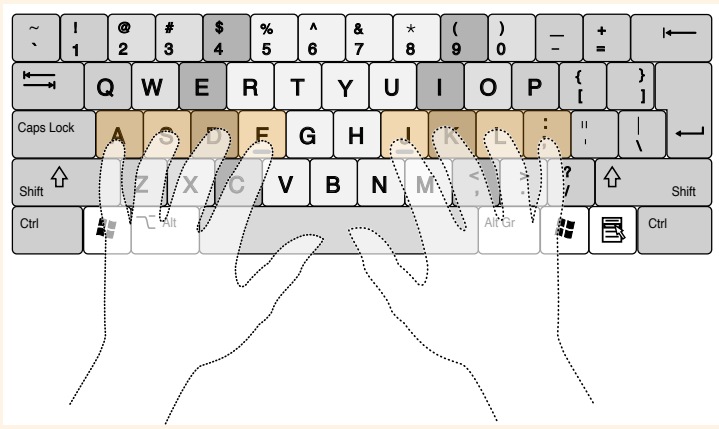


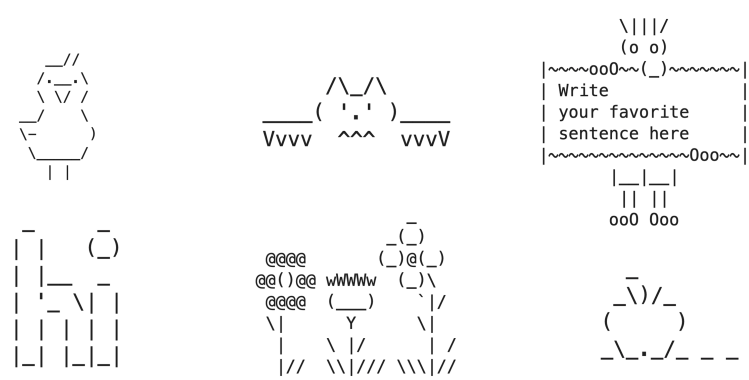
Figure 1.1. Starting position of the fingers on a computer keyboard when touch typing.

we do not have to keep moving our eyes between the keyboard and the screen. How can we learn touch typing? It is very easy; it just requires some practice. The idea is that each finger presses some specific keys of the keyboard, as in Figure 1.1. We position the left index finger on the letter F and the right index finger on the letter J — the two little bumps on these keys define the starting point. The remaining fingers will go on the keys in the same row. For the left hand, the middle finger will go on the letter D, the ring finger on S, and the small finger on A. Similarly, for the right hand, the middle finger will go on the letter K, the ring finger on L, and the small finger on the semicolon. What about the letters G and H that are in between? When needed, the left index finger will move from F to G, and the right index finger from J to H. The fingers will then move upward and downward for the other letters, maintaining the same reciprocal positions. There are plenty of websites to learn touch typing in a fun way, such as `www.typing.com` and `www.typingclub.com`. They are free, and creating an account is optional. They provide gradual exercises starting from typing single letters, to syllables, to words, up to whole sentences. Give it a try?

Ready for some coding exercises? Create a new notebook and solve the following exercises below. If you do not remember how to create a new notebook or new cells, have a look at the *Getting Ready* part.

Let's code!

1. *Writing strings*. Write a string using double quotes. Then, run the cell and observe what happens. Then write a string using single quotes. Run the cell and observe what happens.
2. *Asking questions*. Write two questions using the built-in function `input()` and then answer them.
3. *ASCII art*. Reproduce at least one of the following pieces of ASCII art:



2. Registering to an event

Variables, assignment, and string concatenation

Let's continue building our basics by learning about variables, assignment, and string concatenation. What are they? Let's find out together using Notebook 2! Read the story below aloud and try to understand what the code does:

1. Creating the badge

- You are organizing an event, and you have created the following registration form for the participants:

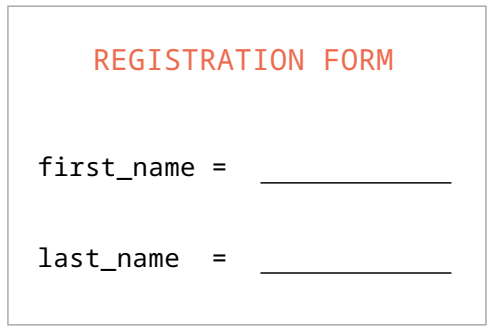


Figure 2.1. Registration form for the event participants.

- The first participant comes in and fills out the form:

```
[]:  1  first_name = "Elena"                    first name is assigned Elena
     2  last_name = "Notch"                     last name is assigned Notch
```

- Then you print her first name and last name on a badge:

```
[]:  1  print(first_name)                       print first name
     2  print(last_name)                        print last name
```

What does the code in these cells do? Let's get some hints by completing the following exercise.

✎ True or false?

1. The command first_name = "Elena" assigns the string "Elena" to the variable first_name. T F
2. The command print(first_name) will print Elena. T F
3. The command print(last_name) will print last_name. T F

Computational thinking and syntax

Any guesses about what the code does? Let's run the first cell:

```
[1]:  1  first_name = "Elena"                  first name is assigned Elena
      2  last_name = "Notch"                   last name is assigned Notch
```

At line 1 we create a **variable** called `first_name`. To the variable `first_name` we **assign** the string `"Elena"`, which is the **value**. Similarly, at line 2 we create a variable called `last_name`, to which we assign the string `"Notch"` as a value. In general, we can assign any value to a variable. For example, we can register the second guest, Fernando Pérez, by writing:

```
[]:   1  first_name = "Fernando"               first name is assigned Fernando
      2  last_name = "Pérez"                    last name is assigned Pérez
```

We can also register the third guest, Guido van Rossum, by writing:

```
[]:   1  first_name = "Guido"                  first name is assigned Guido
      2  last_name = "van Rossum"               last name is assigned van Rossum
```

As you can see, the variable names remain the same (`first_name` and `last_name`), whereas the assigned values can be different (`"Elena"` or `"Fernando"` or `"Guido"`, `"Notch"` or `"Pérez"` or `"van Rossum"`). We can define variables as follows:

A **variable** is a label associated to a **value**

In Python, variables are **lowercase**. When composed of multiple words, these are connected by **underscore**, like in `first_name`. In Jupyter Notebook, variables are **black**. The symbol **=** is called **assignment operator**. This has nothing to do with the *equals* we learned in math! *equals* has a different symbol in coding, as we will see in Chapter 9. In coding we use the symbol = **to assign a value to a variable**, and we pronounce it as *is assigned*. This is a very important concept to remember, and it's one of the most counter-intuitive! Symbols are colored **purple** in Jupyter Notebook.

Let's now run the second cell:

```
[2]:  1  print(first_name)                     print first name
      2  print(last_name)                      print last name
     Elena
     Notch
```

As you might expect, at line 1 we print to the screen the value assigned to the variable `first_name`, which is `Elena`. At line 2 we print the value assigned to the variable `last_name`, which is `Notch`.

A curiosity: Do you know who the other participants are? Fernando Pérez is the co-creator of Jupyter Notebook together with Brian Granger and Guido van Rossum is the creator of Python!

2. Adding participants' information

Time to put together what we have learned so far! Let's continue with our story.

- After receiving the badge, each participant has to go to a computer to enter additional information. First, they re-enter their name:

```
[]:   1  name = input("What's your name?")     name is assigned input What's your name?
```

- Then they add their food requests:

```
[]:  1  food_requests = input("Any food requests?")
```

food requests is assigned input Any food requests?

- Finally they receive the following confirmation message on the computer screen:

```
[]:  1  print("Name: " + name)
     2  print("Food requests: " + food_request)
     3  print(name + ", your request " +
        food_requests + " has been recorded!")
```

print Name: concatenated with name
print Food requests: concatenated with food request
print name concatenated with , your request concatenated with food requests concatenated with has been recorded!

What happens in this code? Let's get some hints by completing the following exercise!

✎ True or false?

1. The answer to the question What's your name? is assigned to the variable name. T F
2. The question Any food requests? is asked before the question What's your name? T F
3. If the answer to the first question is Giulia and the answer to the second question is Vegetarian, then the third print will show: Giulia, your request Vegan has been recorded! T F
4. The symbol + can combine a string and a variable containing a string. T F

Computational thinking and syntax

Let's run the first cell:

```
[3]:  1  name = input("What's your name?")
      What's your name? Elena
```

name is assigned input What's your name?

The built-in function `input()` creates a text box where we enter `Elena`. After we press *Enter* on the keyboard, `Elena` is assigned to the variable `name`. Thus, we can say that `name` is a variable of type string whose value is `Elena`.

Let's run the second cell:

```
[4]:  1  food_requests = input("Any food requests?")
      Any food requests? No nuts
```

food requests is assigned input Any food requests?

Similarly to cell 3, `input()` creates a text box where we enter `No nuts`, which will be assigned to the variable `food_requests` as a string.

Let's now run the last cell of this notebook. What prints do you expect?

[5]:
```
1 print("Name: " + name)
2 print("Food requests: " + food_requests)
3 print(name + ", your request " +
  food_requests + " has been recorded!")
```

```
Name: Elena
Food requests: No nuts
Elena, your request No nuts has been recorded!
```

```
print Name: concatenated with name
print Food requests: concatenated with
food requests
print name concatenated with , your
requests concatenated with food requests
concatenated with has been recorded!
```

At line 1, we print the union of the string `"Name: "` and the value assigned to the variable `name`—that is, the string `Elena`—and we obtain `Name: Elena`. The symbol **`+`** is called **concatenation operator** and allows us to **join strings** together. We can pronounce it as *concatenated with*—not *plus*! At line 2, we do the same, that is, we print the concatenation of the string `"Food requests: "` with the variable `food_requests`, which contains the string `"No nuts"`, to obtain `Food requests: No nuts`. Finally, at line 3, we print the result of the concatenations of four components comprising two variables of type string—`name` and `food_requests`—with two strings—`", your request "` and `" has been recorded!"`. As you can see, we can concatenate as many strings and variables of type string as we want! One last note: as you might remember from the ASCII art in the previous chapter, spaces matter. Therefore, we need to add **spaces at the beginning or end of strings to separate words** in the print. For example, at line 1, we add a space at the end of the string `"Name: "`—without space, we would have printed `Name:Elena`. Similarly, at line 2, we add a space at the end of `"Food requests: "` to avoid printing `Food requests:No nuts`. Finally, at line 3, in the string `", your request "` we add a comma and a space at the beginning and a space at the end, and in the string `" has been recorded!"` we add a space at the beginning. Without these spaces, the print would have been `Elena,your requestNo nutshas been recorded!`.

You have now learned the very basics on which you will build your coding skills and knowledge. Now take a few minutes to complete the following exercise, which will help you clearly summarize the syntax you have learned so far!

✎ Fill in the gaps

Fill in the gaps by inserting what each word is and its color in Jupyter Notebook. See the example in the first sentence:

1. input() is a ___built-in function___ and is colored ___green___ .
2. Also print() is a _____ and is colored _____ .
3. name is a _____ and is colored _____ .
4. "Food requests: " is a _____ and is colored _____ .
5. = is the _____ and is colored _____ .
6. + is the _____ and is colored _____ too.

Recap

- In coding, we assign values to variables.
- The symbol = is the assignment operator (and not the *equals* symbol!), and it can be pronounced *is assigned*.
- The symbol + is the concatenation operator when dealing with strings (and not the *plus* symbol!), and it can be pronounced *concatenated with*.

Dealing with NameError and SyntaxError

When we write code, we inevitably make mistakes, and we get error messages. **Getting error messages is normal when coding!** The most important thing is to learn how to read them so that we can fix errors quickly and keep coding. There are different kinds of errors, and we'll learn how to recognize and fix them throughout the book. Let's start with name errors and syntax errors.

Name error. Let's look at the following code with an error message:

[5]:
```
1 print("Name: " + ame)
2 print("Food requests: " + food_requests)

3 print(name + ", your request " +
  food_requests+ " has been recorded!")
```

```
print Name:  concatenated with ame
print Food requests:  concatenated
with food requests
print name concatenated with ,
your request concatenated with
food requestsconcatenated with has
been recorded!
```

```
---------------------------------------
NameError       Traceback (most recent call last)
Cell In[5], line 1
----> 1 print("Name: " + ame)
      2 print("Food requests: " + food_requests)
      3 print(name + ", your request " + food_requests + " has been recorded!")
NameError: name 'ame' is not defined
```

When encountering an error, we have to perform two steps:

1. **Read the last line of the message**, which tells us what type of errors we have made.
2. **Look for the green arrow**, which shows us the line where the error is.

As the last line of the message says, we are dealing with a **name error**: `NameError: name 'ame' is not defined`. This is a very common error message. It means that there is no variable called `'ame'` in your code. This error message usually pops up in two cases: when we misspell a variable name, or when we forget to run a previous cell containing the variable. In this example, we misspelled the variable `'name'`. This variable is present at lines 1 and 3. Which line should we correct? The arrow pointing at line number 1 shows us that the error is at line 1, where we can see that we typed `'ame'` instead of `'name'`. So we correct the typo, rerun the cell, and keep coding!

↳

Syntax error. Let's look at this code with an error message:

```
[5]:
1  print("Name: " name)
2  print("Food requests: " + food_requests)

3  print(name + ", your request " +
   food_requests+ " has been recorded!")
```

```
print Name:  name
print Food requests:  concatenated
with food requests
print name concatenated with ,
your request concatenated with
food requestsconcatenated with has
been recorded!
```

```
Cell In[5], line 1
    print("Name: " name)
                  ^
SyntaxError: invalid syntax
```

In this case we have made a **syntax error**. Like before, the first thing we do is to read the last line of the message, which says: `SyntaxError: invalid syntax`, meaning that we have forgotten some symbol or punctuation. Where is the error? To answer this question, first we look at the very first line of the message—`Cell In[5], line 1`—which indicates the cell and line where the error occurred—that is, cell 5, line 1, in this case. Second, we look at the **position of the hat symbol** ^ below the command, which specifies the location of the error within the command. In our case, we forgot to add the the concatenation operator. So we add it, rerun the cell, and keep coding!

Ready to exercise? Let's go!

Let's code!

1. *At the gym.* You are the manager of a gym and you have to register a new person. What variables would you create? Write three variables, assign a value to each of them (make sure they are strings!), and print them.
2. *At a bookstore.* You are the owner of a bookstore and you want to create a book catalog. You start with the first book: Code Girls by Liza Mundy. You create two variables, `book title` and `author`, assign them the actual title and author, and print them. Then, pick a book of your choice, create the two variables again, assign the corresponding values, and print them.
3. *Where are you from?* Ask a person what country he comes from and where he lives. Then print three sentences like in cell 5 of the code in this chapter.
4. *What's your favorite song?* Ask a person her favorite song and favorite singer. Then print three sentences like in cell 5 of the code in this chapter.

PART 2

INTRODUCTION TO LISTS AND THE IF/ELSE CONSTRUCT

In this part, you will learn about lists, which are simply sequences of elements of various types—for example, strings. You will also learn how to manipulate them, that is, how to add, remove, or replace one or more elements. And finally, you will learn about the `if/else` construct, which allows you to execute code based on conditions. Ready? Let's go!

3. In a bookstore

Lists and if... in... / else...

What does a list look like? And how do we use `if/else` construct? To answer these questions, let's open Jupyter Notebook 3 and begin! Read the following example aloud and try to understand it:

- You are the owner of a bookstore. On the programming shelf there are:

```
[]:  1  books = ["Learn Python", "Python for all", "Intro
         to Python"]
     2  print(books)
```

```
books is assigned Learn Python,
Python for all, Intro to Python
print books
```

- A new customer comes in, and you ask what book she wants:

```
[]:  1  wanted_book = input("Hi! What book would you like
         to buy?")
     2  print(wanted_book)
```

```
wanted book is assigned input Hi!
What book would you like to buy?
print wanted book
```

- You check if you have the book, and you reply accordingly:

```
[]:  1  if wanted_book in books:
     2      print("Yes, we sell it!")
     3  else:
     4      print("Sorry, we do not sell that book")
```

```
if wanted book in books
print Yes, we sell it!
else
print Sorry, we do not sell that book
```

What does the code above do? Get some hints by completing the following exercise.

✎ True or false?

1. On the programming shelf there are 2 books. T F
2. If the customer wants a book that is in the programming shelf, you print: Yes, we sell it! T F
3. The if/else construct allows us to execute commands based on conditions. T F

Computational thinking and syntax

Let's analyze the code line by line, starting with the first cell:

```
[1]:  1  books = ["Learn Python", "Python for all", "Intro
          to Python"]
      2  print(books)
      ['Learn Python', 'Python for all', 'Intro to Python']
```

```
books is assigned Learn Python,
Python for all, Intro to Python
print books
```

On line 1 there is a variable called `books`, to which we assign a sequence of elements of type string: `"Learn Python"`, `"Python for all"`, and `"Intro to Python"`. The elements are separated by commas and they are in between **square brackets**. A variable with this syntax is called **list**. In our code, `books` is a **variable of type list whose elements are of type string**. In other words, we can say that `books` is a **list of strings**. A list is defined as follows:

A **list** is a sequence of **elements** separated by commas `,`
and in between square brackets `[]`

As its name says, a list is literally a list of elements, similar to a shopping list or a to-do list. It can contain elements of various types, such as strings, numbers, etc. For now, we will consider only lists of strings.

Let's run the second cell:

```
[2]:
1  wanted_book = input("Hi! What book would you like to buy?")
2  print(wanted_book)

Hi! What book would you like to buy? Learn Python
Learn Python
```

```
wanted book is assigned input Hi! What book would you like to buy?
print wanted book
```

You are now familiar with the code in this cell. Briefly, on line 1 we created a variable called `wanted_book`, which contains the user's answer to the question: `Hi! What book would you like to buy?` Then, on line 2, we printed the value contained in the variable `wanted_book`.

Let's run the third cell:

```
[3]:
1  if wanted_book in books:
2      print("Yes, we sell it!")
3  else:
4      print("Sorry, we do not sell that book")

Yes, we sell it!
```

```
if wanted book in books
print Yes, we sell it!
else
print Sorry, we do not sell that book
```

Here, we finally meet the `if/else` construct. Let's learn how it works by starting from lines 1 and 2. These lines say `if wanted_book`, which is `"Learn Python"`, is in `books`, which is `["Learn Python", "Python for all", "Intro to Python"]` (line 1), print `"Yes, we sell it!"` (line 2). In more details, in line 1 we check whether the value assigned to the variable `wanted_book` is one of the elements of the list `books`. If that is the case, then we move to line 2 and print a positive answer to the user.

What if `wanted_book` is not in the list? Let's rerun cell 2 and enter a book that is not in the list:

```
[4]:
1  wanted_book = input("Hi! What book would you like to buy?")
2  print(wanted_book)

Hi! What book would you like to buy? Basic Python
Basic Python
```

```
wanted book is assigned input Hi! What book would you like to buy?
print wanted book
```

In this case, what do you expect when running the cell below? Let's run it:

```
[5]:
1  if wanted_book in books:
2      print("Yes, we sell it!")
3  else:
4      print("Sorry, we do not sell that book")

Sorry, we do not sell that book
```

```
if wanted book in books
print Yes, we sell it!
else
print Sorry, we do not sell that book
```

We start again from line 1, where we read `if wanted_book`, which now is `"Basic Python"`, is in `books`, which is `["Learn Python", "Python for all", "Intro to Python"]`. But this time, `"Basic Python"` is not in the list `books`. So we skip line 2, go directly to line 3—where there is `else`—and proceed to line 4, where we print the string `"Sorry, we do not sell that book"`.

As you can deduce from the example above, in an `if/else` construct, code is executed depending on the **truthfulness of a condition**. If the condition in the `if` line is met, or true, we execute the underlying code. Otherwise, if the condition in the `if` line is not met, or false, then we execute the code under `else`. Therefore, we can define the `if/else` construct as follows:

> An **`if/else`** construct **checks** whether a **condition** is true or false,
> and **executes** code **accordingly**:
> if the condition is met, the code under the **`if`** condition is executed;
> if the condition is not met, the code under **`else`** is executed

Let's now focus on the syntax. An `if/else` construct is composed of four parts, explained below:

- **`if` condition** (line 1) contains a condition that determines code execution. It consists of three components: (1) the **keyword `if`** colored **bold green** in Jupyter Notebook, (2) the condition itself, and (3) the colon punctuation mark **:**.
- **Statement** (line 2) contains the code that gets executed if the condition at line 1 is met.
- **`else`** (line 3) implicitly contains the alternative to the condition on line 1. This line is always composed of the **keyword `else`** followed by the colon **:**.
- **Statement** (line 4) contains the code that gets executed if the condition at line 1 is not met.

Note: **`else` and its following statements are not mandatory**. There are cases when we do not want to do anything if the conditions are not met. Some examples of this scenario are provided in the following chapters.

Before concluding, let's zoom even more into these lines and focus on two more aspects: membership conditions and indentation. In coding, we can use various types of conditions, and you will see these throughout the book. In this case, we have a **membership condition**—`wanted_book in books` (line 1)—where we check whether a variable contains one of the elements of a list. In a membership condition, we write: (1) variable name, (2) **`in`**, and (3) the list in which we want to find the element. **`in`** is a **membership operator**. In Jupyter Notebook, `in` is colored **bold green**, like keywords. In general, make sure not to confuse keywords, in bold green, with built-in functions, in fainter green.

Notice that the statements under the `if` condition (line 2) and under the `else` (line 4) are always indented, which means shifted toward the right. An **indentation** consists of 4 spaces, or 1 tab. In Jupyter Notebook, when pressing *Enter* or *Return* after writing the `if` or `else` lines, the cursor is always automatically placed at the right indented position. Finally, under an `if` or an `else` condition, we can write **several commands**—as we will see in the next chapters—but they must be indented correctly to be executed.

✎ Complete the table

`Up to this point, you have already learned quite a lot of syntax. Complete the following table to summarize the syntax you know so far.`

Code element	What it is	What it does
books	A variable of type list	It contains a sequence of strings
wanted_book		
"Learn Python"		
if		
in		
else		
=		
+		
input()		
print()		

Recap

- Lists are a Python type that contains a sequence of elements (for example, strings) separated by commas and in between square brackets.
- The `if/else` construct allows us to execute code based on conditions.
- The membership operator `in` verifies whether an element is in a list.
- In Python, we use indentation for statements below `if` or `else`.

Let's give variables meaningful names!

One of the fundamental criteria when writing code is **readability**. It is important to write code that is easy to read both for our future selves and for others. One of the ways to make code readable is to create **meaningful variable names**. As an example, let's consider the code we analyzed in this chapter. On line 1 of cell 2 we created the variable `wanted_book`:

```
[2]:  1  wanted_book = input("Hi! What book would you like
         to buy?")
```

```
wanted book is assigned input Hi! What book would
you like to buy?
```

Instead of `wanted_book`, we could have named the variable `answer`:

```
[2]:  1  answer = input("Hi! What book would you like to
         buy?")
```

```
answer is assigned input Hi! What book would you
like to buy?
```

The name `answer` is logically consistent because this variable contains the answer to the question `"Hi! What book would you like to buy?"`. However, `answer` is not the best choice because it is a very generic variable name. Variable names should be pertinent, **representing the information they contain**. Consider having 10 `input()` commands in the code. What do we call the corresponding variables? We don't want to call them `answer_1`, `answer_2`, ..., `answer_10`; it would

be hard to remember what we assigned to `answer_7`, for example. Or, if we later decide to reshuffle some questions, then we will have to rename the variables to make sure the numbers increase sequentially. This would generate a lot of confusion and increase the possibility of errors.

Back to the previous example, the name `answer` would also not be meaningful in the following line of code from cell 3:

```
[3]: 1 if answer in books:                    if answer in books
```

It does not make much sense to look for an answer in a list of books! But it makes more sense to look for a wanted book in a list of books:

```
[3]: 1 if wanted_book in books:               if wanted book in books
```

Let's code!

For each of the following scenarios, create code similar to that presented in this chapter.

1. *In an art gallery.* You are the owner of an art gallery. Write a list of some paintings you sell. A new customer comes in, and you ask what painting she wants to buy. You check whether you have that painting and reply accordingly.
2. *In a travel agency.* You are the owner of a travel agency. Write a list of some travel destinations you sell tickets for. A new customer comes in, and you ask where he wants to go. You check whether you offer that travel destination and reply accordingly.
3. *In a chemical lab.* You are the manager of a lab. On a shelf there some jars containing chemicals. Write a list containing the names of the chemicals. One of the lab members comes to you and you ask what chemical she wants. You check in your system whether you have that chemical and reply accordingly.
4. *In a tea room.* You are the owner of a tea room. Write a list of teas you offer. A new customer comes in, and you ask what tea he wants. You check on the menu whether you serve that tea and reply accordingly.

4. Grocery shopping

List methods: .append() and .remove()

What are methods? And what do `.append()` and `.remove()` do? To answer this questions, open Jupyter Notebook 4 and follow along. Let's start with the following example:

- You are going to a grocery store where you have to buy some food:

```
[]:  1  shopping_list = ["carrots", "chocolate", "olives"]
     2  print(shopping_list)
```

```
shopping list is assigned carrots, chocolate, olives
print shopping list
```

- Right before leaving home, you ask yourself if you have to buy something else. If the item is not in the list, you add it:

```
[]:  1  new_item = input("What else do I have to buy?")
     2  if new_item in shopping_list:
     3      print(new_item + " is/are already in the
            shopping list")
     4      print(shopping_list)
     5  else:
     6      shopping_list.append(new_item)
     7      print(shopping_list)
```

```
new item is assigned input What else do I have to buy?
if new item in shopping list
print new item concatenated with is/are already in the shopping list
print shopping list
else
shopping list dot append new item
print shopping list
```

- Finally, you ask yourself if you have to remove an item. If so, you remove the item from the list:

```
[]:  1  item_to_remove = input("What do I have to
        remove?")
     2  if item_to_remove in shopping list:
     3      shopping_list.remove(item_to_remove)
     4      print(shopping_list)
     5  else:
     6      print(item_to_remove + " is/are not in the
            shopping list")
     7      print(shopping_list)
```

```
item to remove is assigned input What do I have to remove?
if item to remove in shopping list
shopping list dot remove item to remove
print shopping list
else
print item to remove concatenated with is/are not in the shopping list
print shopping list
```

To get a better idea of what happens in this code, match the sentence halves in the following exercise.

Match the sentence halves

1. The variable shopping_list contains	a. we remove it from the shopping list.
2. If the new item is not in the shopping list	b. to remove an element from a list.
3. If the item to remove is in the shopping list	c. "carrots", "chocolate", and "olives".
4. The method .append() allows us	d. we add it to the shopping list.
5. The method .remove() allows us	e. to add an element at the end of a list.

Computational thinking and syntax

Let's dig into the code by running the first cell:

[1]:
```
1  shopping_list = ["carrots", "chocolate", "olives"]

2  print(shopping_list)
```
```
['carrots', 'chocolate', 'olives']
```
```
shopping list is assigned carrots, chocolate, olives
print shopping list
```

We start with a list called `shopping_list`, which contains three strings: `"carrots"`, `"chocolate"`, and `"olives"` (line 1). Then, we print the shopping list to the screen (line 2).

What does `.append()` do? Let's run the second cell:

[2]:
```
1  new_item = input("What else do I have to buy?")

2  if new_item in shopping_list:
3      print(new_item + " is/are already in the
       shopping list")

4      print(shopping_list)
5  else:
6      shopping_list.append(new_item)
7      print(shopping_list)
```
```
What else do I have to buy? carrots
carrots is/are already in the shopping list
['carrots', 'chocolate', 'olives']
```
```
new item is assigned input What else do I have to buy?
if new item in shopping list
print new item concatenated with is/are already in the shopping list
print shopping list
else
shopping list dot append new item
print shopping list
```

In this cell, we ask the user to input a new item to buy, and the answer is saved in the variable `new_item` (line 1). Then, we act according to the value contained in `new_item`. If `new_item` is already in `shopping_list` (line 2), we print a message saying that the item is already in the shopping list (line 3). To make the message more precise, we concatenate the string in `new_item` with the string `"is/are already in the shopping list"`. Then, we print the shopping list to check that the item is actually in the list (line 4).

What if the item is not in the shopping list? Let's rerun the cell and enter an item that is not in the list:

[3]:
```
1  new_item = input("What else do I have to buy?")

2  if new_item in shopping_list:
3      print(new_item + " is/are already in the
       shopping list")

4      print(shopping_list)
5  else:
6      shopping_list.append(new_item)
7      print(shopping_list)
```
```
What else do I have to buy? apples
['carrots', 'chocolate', 'olives', 'apples']
```
```
new item is assigned input What
else do I have to buy?
if new item in shopping list
print new item concatenated with
is/are already in the shopping
list
print shopping list
else
shopping list dot append new item
print shopping list
```

This time, we entered `apples` in the text box created by `input()` (line 1). Because `apples` is not in the shopping list (line 2), we skip the commands at lines 3 and 4 and jump directly to the `else` (line 5) to execute the commands below. We add the new item to the list (line 6), and we print the list to check whether we added the element correctly (line 7).

How did we add a new element to a list? Let's have a closer look at line 6. Here, **the method `.append()`** adds the element `new_item` to the `shopping_list`. Note that `.append()` always **adds an element at the end of a list**. As we said, `.append()` is a method. But what is a method? A preliminary definition—we'll redefine it when we talk about object-oriented programming in the last part of the book—is as follows:

> A **method** is a built-in function for a **specific** variable **type**

You can recognize that methods are functions because they are followed by round brackets. However, a method has its own syntax, which is composed of four elements: (1) variable name, (2) dot, (3) method name, and (4) round brackets. In the round brackets, there can be an **argument**, such as `new_item` in this case. Different data types have different methods. For example, `.append()` can be used for lists but not for strings. Lists have a total of eleven methods, and we will learn all of them throughout this book. Methods are colored **blue** in Jupyter Notebook.

What does `.remove()` do? Let's run the last cell:

[4]:
```
1  item_to_remove = input("What do I have to
   remove?")
2  if item_to_remove in shopping list:

3      shopping_list.remove(item_to_remove)

4      print(shopping_list)
5  else:
6      print(item_to_remove + " is/are not in the
       shopping list")

7      print(shopping_list)
```
```
What do I have to remove? olives
['carrots', 'chocolate', 'apples']
```
```
item to remove is assigned input
What do I have to remove?
if item to remove in shopping
list
shopping list dot remove item to
remove
print shopping list
else
print item to remove concatenated
with is/are not in the shopping
list
print shopping list
```

This time, we ask the user what item they want to remove (line 1). If the item to remove is in the shopping list (line 2), then we remove the item (line 3) and print the resulting list (line 4). How do we remove an item? We use **`.remove()`**, which is the **list method to remove an item from a list**. The

syntax is the same as for `.append()` and any other method: list name followed by dot, method name, and round brackets, which can contain an argument. As an argument, `.remove()` takes the element to be removed from the list.

Finally, what if we answer the question `"What do I have to remove?"` with an element that is not in the list? Let's have a look:

[5]:
```
1 item_to_remove = input("What do I have to remove?")
2 if item_to_remove in shopping list:
3     shopping_list.remove(item_to_remove)
4     print(shopping_list)
5 else:
6     print(item_to_remove + " is/are not in the shopping list")
7     print(shopping_list)
```

```
What do I have to remove? grapes
grapes is/are not in the list
['carrots', 'chocolate', 'apples']
```

```
item to remove is assigned input What do I have to remove?
if item to remove in shopping list
shopping list dot remove item to remove
print shopping list
else
print item to remove concatenated with is/are not in the shopping list
print shopping list
```

In the text box created by `input()`, we entered `grapes` (line 1), which is not in `shopping_list` (line 2). Therefore, we skip lines 3 and 4 and jump to the `else` at line 5. There, we print a message saying that `item_to_remove` is not in the shopping list (line 6) and print the shopping list for a final check (line 7).

✎ Complete the table

In Python, we use a lot of punctuation marks. Sum up what you have learned so far by completing the following table:

Punctuation symbol	What it's called	What it does
'' or ""	Single quotes or double quotes	They contain a strings
( )		
[ ]		
:		
,		
.		

Recap

- The method `.append()` adds an element at the end of a list.
- The method `.remove()` removes an element from a list.

Why do we print so much?

When coding, it is important to **keep control of variable's values**. And particularly when learning to code, every time we create or modify a variable, it's important to make sure the code does what it is intended to do. Printing is an easy way to check that variable modifications correspond to our intentions. As an example, consider the code in cell 4, and let's focus on the `if` condition and its statements. Let's first rewrite the code without the printing command:

```
[4]:
1  item_to_remove = input("What do I have to remove?")
2  if item_to_remove in shopping list:
3      shopping_list.remove(item_to_remove)

What do I have to remove? olives
```

```
item to remove is assigned input
What do I have to remove?
if item to remove in shopping list
shopping list dot remove item to remove
```

How do we know that the code actually worked correctly? That is, how do we know whether `'olives'` was actually removed from `shopping_list`? We can assume that it happened, but we cannot be sure until we see it with our eyes. So, we need to print. Let's rewrite the code by adding `print()` back to line 4:

```
[4]:
1  item_to_remove = input("What do I have to remove?")
2  if item_to_remove in shopping list:
3      shopping_list.remove(item_to_remove)
4      print(shopping_list)

What do I have to remove? olives
['carrots', 'chocolate', 'apples']
```

```
item to remove is assigned input
What do I have to remove?
if item to remove in shopping list
shopping list dot remove item to remove
print shopping list
```

Because we printed, we can make sure that `'olives'` is not in the `shopping_list` and our code accomplished what we intended. Always print extensively when coding! You can always remove the `print()` function later on.

Let's code!

1. For each of the following scenarios, create code similar to the one presented in this chapter.
   a. *Organizing an event.* You are organizing an event. Write a list of what you need to buy. Then ask your co-organizer what else you have to buy. If the item is not in the list, add it. Finally, ask your co-organizer if there is anything you need to remove from the list. If so, remove the item from the list.
   b. *Favorite cities.* Write a list containing names of cities. Ask a friend their favorite city. If the city is not in the list, add it. Then, ask your friend if they do not like one of the cities you listed. If so, remove the city from your list.
2. *Shoe store.* You are the owner of a shoe store, and you have to place a new order for the next summer season. You go to the storage room, and you create a list of the remaining shoes: sneakers, boots, ballerinas. You know that in summer your customers will want sandals, so you add them to the list. However, they are not going to buy boots, so you remove them from the list. After you get the new supplies, a new customer comes in. You ask what shoes he wants to try, and he replies that he'd like to try sandals. You check in your list and reply accordingly. Then you ask if he wants to have a look at something else, and he replies that he'd like to try boots. You check in your list again and reply accordingly.

3. *Currency exchange office.* You work at a currency exchange office. The available currencies are Euros, Canadian Dollars, and Yen, whereas the Swiss Franc is unavailable, so you will have to order it. Create a list of available currencies and a list of currencies to order. A new customer comes in; you ask what currency she wants. After she replies, you check in the list of available currencies. If the currency she wants is available, you tell her that you have it, remove the currency from the list of available currencies, and add the currency to the list of currencies to order. If the currency she wants is not available, you tell her that you do not have that currency, and add the currency to the list of currencies to order.

5. Customizing the burger menu

List methods: .index(), .pop(), and .insert()

Let's learn three more list methods: `.index()`, `.pop()`, and `.insert()`. Open Jupyter Notebook 5, and read the following example aloud.

- You are at a food court, ready to order. Today's menu includes a burger, a side dish, and a drink:

```
[]:  1  todays_menu = ["burger", "salad", "coke"]
     2  print(todays_menu)
```

```
todays menu is assigned burger, salad, coke
print todays menu
```

- You are happy with burger and coke, but you want to change the side dish from `salad` to `fries`. To do so, you:

  1. Look at the position of the side dish in the menu:

```
[]:  1  side_dish_index = todays_menu.index("salad")
     2  print(side_dish_index)
```

```
side dish index is assigned todays menu dot index salad
print side dish index
```

  2. Remove `salad` from the side dish position:

```
[]:  1  todays_menu.pop(side_dish_index)
     2  print(todays_menu)
```

```
todays menu dot pop side dish index
print todays menu
```

  3. Add `fries` to the side dish position:

```
[]:  1  todays_menu.insert(side_dish_index, "fries")
     2  print(todays_menu)
```

```
todays menu dot insert at side dish index fries
print todays menu
```

What happens in this code? Get some hints by completing the following exercise.

✎ True or false?

1. The method .index() gives us the position of an element in a list. T F
2. The position of salad is 2. T F
3. We remove the element in position side_dish_index and insert a new element in the same position. T F
4. .index(), .pop(), and .insert() are three string methods. T F

Computational thinking and syntax

Let's analyze the details of the code! Let's run the first cell:

```
[1]:  1  todays_menu = ["burger", "salad", "coke"]
      2  print(todays_menu)
['burger', 'salad', 'coke']
```

```
todays menu is assigned burger, salad, coke
print todays menu
```

We create a list called `todays_menu` containing three elements of type string—`"burger"`, `"salad"`, and `"coke"` (line 1)—and we print it (line 2).

In the second cell, we meet the new list method `.index()`. What does it do? Let's run the cell:

```
[2]:
1  side_dish_index = todays_menu.index("salad")
2  print(side_dish_index)
1
```

```
side dish index is assigned todays menu dot index salad
print side dish index
```

The method **`.index()`** looks for the element `"salad"` in the list `todays_menu` and **gives us its position**. More technically, we say that `.index()` **takes** the **argument** `"salad"` and **returns** its index. The position of `"salad"` is then assigned to the variable `side_dish_index` (line 1), which we print (line 2). Note that in coding, we use the two synonyms **index** and **position** interchangeably.

Why is `"salad"` in position 1 and not 2? This is because in Python we count elements starting **from 0**, as you can see in Figure 5.1: `"burger"` is in position 0, `"salad"` in position 1, and `"coke"` in position 2.

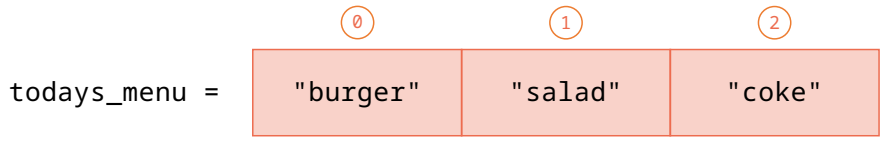


Figure 5.1. Representation of the list `todays_menu`: each rectangle is a list element, and the number above is the corresponding index.

Finally, note that an element position is a number. In Python, **zero**, **positive**, and **negative whole numbers** are called **integers**, abbreviated as **int**. In our example, the variable `side_dish_index` contains the number 1, and it is of **type integer**.

Let's discover what `.pop()` does by running the next cell:

```
[3]:
1  todays_menu.pop(side_dish_index)
2  print(todays_menu)
['burger', 'coke']
```

```
todays menu dot pop side dish index
print todays menu
```

The method **`.pop()`** removes the element in position `side_dish_index` from the list `todays_menu`. In other words, `.pop()` takes `side_dish_index` as an argument and **removes the element at that index**, which is `"salad"`. In the previous chapter, we saw another method that deletes an element from a list: `.remove()`. What is the difference between the two methods? The method **`.remove()`** deletes an element of a certain **value**, whereas **`.pop()`** deletes an element **in a specific position**.

And finally, let's learn the method `.insert()`. Let's run the last cell:

```
[4]:
1  todays_menu.insert(side_dish_index, "fries")
2  print(todays_menu)
['burger', 'fries', 'coke']
```

```
todays menu dot insert at side dish index fries
print todays menu
```

The method **`.insert()`** allows us to **add an element at a specific index**. It takes two arguments: (1) the index where we want to insert the new element and (2) the value of the new element. In this case, we want to insert at position `side_dish_index`, which is position 1, the string `"fries"`. Similarly, in the

previous chapter we saw another method to add an element to a list: `.append()`. What's the difference? The method **`.append()`** adds an element **at the end** of a list, whereas **`.insert()`** adds an element **in a specific position** of a list.

When dealing with lists, in some cases, it is more convenient to work directly on the elements and use methods like `.append()` and `.remove()`. In other cases, it is more appropriate to work on elements' positions, so we use methods such as `.index()`, `.pop()`, and `.insert()`. Note that `.append()`, `.remove()`, `.pop()`, and `.insert()` modify a list. On the other side, `.index()` gives us the position of an element in a list, and we can save this information in a separate variable. Lastly, `.append()`, `.remove()`, `.index()`, and `.pop()` take only one argument, whereas `.insert()` takes two arguments, which are position and new element.

✎ Complete the table

So far you have learned five list methods. Summarize what they do by completing the following table.

List method	What it does
`.append()`	
`.remove()`	
`.index()`	
`.pop()`	
`.insert()`	

Recap

- The method `.index()` returns the position of an element in a list.
- The method `.pop()` removes an element in a certain position from a list.
- The method `.insert()` adds an element in a certain position to a list.
- Indices (or positions) of elements are whole numbers that start from 0 and increase in increments of one unit; they are of type integer.

We code in English!

During a coffee break, a colleague once told me, "Isn't it crazy that when English speaking people code, they actually do it in their own mother tongue? I mean, when they say *if*, they actually mean *if*!" I had never thought about it. For me, an Italian mother tongue, `if` was just a keyword composed of two symbols. Reading `if book in books` or `ab book in books` was exactly the same! I had learned to look at keywords and variable names as abstract symbols with no intrinsic meaning; they were just entities with a specific function. After that conversation, I mentally translated keywords and variable names into my mother tongue, and everything acquired much more meaning and made so much more sense! I grasped the importance of variable

names (they actually have a meaning in English!), and thus, I started writing commands like `if book in books`, instead of `if variable_1 in list_1`. Now, when I code, I mainly think in English. But that translation process helped me acquire more awareness and make my code much more readable. In Chapters 4 and 5, we learned five list methods. Their names actually have a meaning in English. `remove`, `insert`, and `index` are pretty straightforward. To remember that `append` adds new elements at the end of a list, one can think of the appendix of a book, which is always at the end, or of the appendix in the intestine, which is somewhere at the end of the abdomen. To remember `pop`, one can think of making popcorn, like little explosions, that here remove elements from a certain position. Whether English is your native tongue or not, remember that we code in English!

Let's code!

1. For each of the following scenarios, create code similar to that presented in this chapter:
   a. *Getting a new bike.* You go to a bike store to buy your new bike. There you find a bike you like: it is blue, electric, and has gears. Write a list with these characteristics. You are happy with the bike being electric and having gears, but you would like to change its color. To do so, you (1) look at the position of the blue color in the bike option list, (2) remove the blue color, and (3) add the color you want.
   b. *Ordering a T-shirt online.* You are ordering a new T-shirt online. You find a T-shirt you like, which is red, with a round neck, and with a print *add your text here*. Write a list with these characteristics. Now you want to add your own text to the T-shirt. To do so, you (1) look at the position of *add your text here*, (2) remove *add your text here*, and (3) add the text you want to be printed on your T-shirt. After completing the exercise, can you think of an alternative way to change the T-shirt print?
2. *Steve Jobs.* Given the following list:

   ```
   steve_jobs = ["somebody", "learn", "use", "a computer", "it teaches us"]
   ```

   Find out a famous quote by Steve Jobs by doing the following:
   a. Add the new string `"think"` at the end of the list.
   b. Add `"should"` in position 1.
   c. Add `"how to"` in position 3. Then also add it in position 7.
   d. Replace `"use"` with `"program"`.
   e. Add `"because"` after `"a computer"`.
   f. Replace `"somebody"` with `"everybody"`.
   g. Add `" - Steve Jobs"` at the end.
3. *Grace Hopper.* Do you know why we say *debugging* in coding? Let's find out! Given the following list:

   ```
   grace_hopper = ["In 1946", "a moth", "caused", "a malfunction", "in an early",
   "electromechanical", "computer"]
   ```

   Modify it by doing the following:
   a. Replace `"In 1946"` with `"From then on"`.
   b. Add `"we said"` after `"computer"`.
   c. Remove the string in position 5 (6th element) and add `"with a"` in the same position.
   d. Remove the string in position 3 (4th element).
   e. Substitute (or replace) `"a moth"` with `"when anything"`.
   f. Remove `"in an early"`.
   g. Add `"it had bugs in it"` at the end of the list.
   h. Substitute `"caused"` with `"went wrong"`.
   i. Add `" - Grace Hopper"` at the end of the list.

6. Traveling around the world

List slicing

In the previous two chapters, you learned five methods to manipulate lists: `.append()`, `.remove()`, `.index()`, `.pop()`, and `.insert()`. These list methods are very convenient and easy to remember; however, they can make code quite cumbersome. In Python, there is an alternative and more compact way to change, add, and remove list elements, which you will see in the next chapter. This alternative method is based on slicing, whose meaning and rules are presented in this chapter. Ready to get to know everything about slicing? Open Jupyter Notebook 6 and follow along!

First of all, what is slicing?

Slicing means **accessing** list elements through their **indices**

If you have a sweet tooth, the word "slicing" immediately reminds you of a slice of cake. And in fact, there is quite a similarity between slicing a cake and slicing a list! In the first case, you extract one or more cake slices for your guests—and yourself! In the second case, you extract one or more list elements to be used in subsequent lines of code.

- Let's meet the list we will slice:

```
[1]:  1  cities = ["San Diego", "Prague", "Cape Town", "Tokyo",
      "Melbourne"]
      2  print(cities)
['San Diego', 'Prague', 'Cape Town', 'Tokyo', 'Melbourne']
```

```
cities is assigned San
Diego, Prague, Cape
Town, Tokyo, Melbourne
print cities
```

In this cell, there is a list called `cities` containing five strings: `"San Diego"`, `"Prague"`, `"Cape Town"`, `"Tokyo"`, and `"Melbourne"` (line 1), and we print it (line 2).

How are we going to slice `cities`? The **syntax** for slicing is very easy. It consists of the **list name followed by opening and closing square brackets**, like this: `cities[]`. In between the square brackets, we write the **positions of the elements** we want to slice. For this reason, it's crucial to be aware of the positions of each element within a list. In the list `cities`, the elements have the following positions:

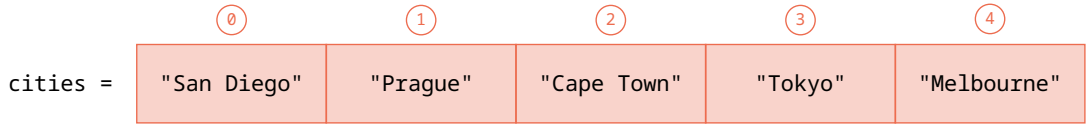


Figure 6.1. Representation of the list `cities`: each rectangle is a list element, and the number above is the corresponding index.

Now, how do we write element positions in between the square brackets? There are various rules depending on how many elements we want to slice, where they are, and in which direction we want to extract them. We are going to learn all these rules in the coming pages.

A last note before starting: to better learn about slicing, I suggest this method. Every time you read a slicing task (for example: Slice `"Prague"`), cover the following code with a piece of paper. Try to guess

the code, and compare your guess with the solution. Then carefully read the explanation. Make sure you fully understand the current example before proceeding to the next one. Enough words, time to slice!

1. Slice `"Prague"`:

```
[2]: 1 print(cities[1])                    print cities in position one
     'Prague'
```

In this cell, we slice (or access) `"Prague"`, which is in position `1`, and we print it. As you can see, when we slice **one single element** from a list, we write the position of the element itself in between the square brackets. Thus, we can summarize this syntax as **`list_name[element_position]`**, and we can read it as *list name in position element position*.

Note: For simplicity, in this example and those that follow, we just print the sliced elements. However, one could assign a sliced element to a variable, like this:

```
[2]: 1 sliced_city = cities[1]             sliced city is assigned cities in position one
     2 print(sliced_city)                  print sliced city
     'Prague'
```

We will assign sliced list elements to variables in the following chapters. For now, let's focus on understanding how slicing works!

2. Slice the cities from `"Prague"` to `"Tokyo"`:

```
[3]: 1 print(cities[1:4])                  print cities in positions from one to four
     ['Prague', 'Cape Town', 'Tokyo']
```

In this cell, we slice and print three consecutive elements—`"Prague"`, `"Cape Town"`, and `"Tokyo"`— that are at positions 1, 2, and 3, respectively. To do so, in between the square brackets, we write two numbers, separated by a colon `:`. The **first number** is the **position of the first element** we want to slice, and we call it **start**. In this case, the start is `1`, which corresponds to `"Prague"`. The **second number** is the **position of the last element** we want to slice, to which we must add 1. We call it **stop**. The stop always follows the **plus one rule**, which simply says that **we must add 1 to the position of the last element we want to slice** (you can learn the reasoning behind this rule in the *In more depth* section at the end of this chapter). In this example, the position of the last element (`"Tokyo"`) is 3, to which we must add 1 because of the plus one rule, so the stop is `4`. We can summarize the syntax to slice consecutive elements as **`list_name[start:stop]`**, and we can read it as *list name in positions from start to stop*.

3. Slice `"Prague"` and `"Tokyo"`:

```
[4]: 1 print(cities[1:4:2])                print cities in positions from one to four
                                           with a step of two
     ['Prague', 'Tokyo']
```

In this case, we want to slice and print two non-consecutive elements—`"Prague"` and `"Tokyo"`— which are at positions 1 and 3, respectively. In the code above, you might recognize that `1` is the start, `4` is the stop (because of the plus-one rule), and `2`? That is the **step**! As you can see, `"Tokyo"` is positioned 2 steps after `"Prague"`: there is 1 step from `"Prague"` to `"Cape Town"`, and 1 step from `"Cape Town"` to `"Tokyo"`, for a total of 2 steps. Therefore, the syntax to slice **non-consecutive elements** is an extension

of the rule we saw in the example above: **`list_name[start:stop:step]`**, which you can read as *list name from start to stop with step*. We can call it the **three-s rule**, where the three s's are the initials of start, stop, and step, respectively.

The most convenient aspect of the three-s rule is that we can simplify it in several situations. For example, we didn't write the step in the example 2, where we sliced the cities from `"Prague"` to `"Tokyo"`. Do you know why? Because **when elements are consecutive, the step is 1**—`"Cape Town"` is 1 step after `"Prague"`, and `"Tokyo"` is 1 step after `"Cape Town"`—and when the step is 1 we can simply omit it. Obviously, we could have written the code specifying the step as follows:

```
[3]: 1 print(cities[1:4:1])
['Prague', 'Cape Town', 'Tokyo']
```

```
print cities in positions from one to four
with a step of one
```

However, adding the step here is a redundancy, so we simply avoid it.

4. Slice the cities from `"San Diego"` to `"Cape Town"`:

```
[5]: 1 print(cities[0:3])
['San Diego', 'Prague', 'Cape Town']
```

```
print cities in positions from zero to three
```

Here we have to slice consecutive elements. So, we specify the start, which is `0` for `"San Diego"`, and the stop, which is `3` for `"Cape Town"`, but we can omit the step because it is `1`. Interestingly, in this case we can simplify the three-s rule even more! Because the **start** coincides with the **first element of the list**, we can omit it:

```
[6]: 1 print(cities[:3])
['San Diego', 'Prague', 'Cape Town']
```

```
print cities in positions from the beginning
of the list to three
```

5. Slice the cities from `"Cape Town"` to `"Melbourne"`:

```
[7]: 1 print(cities[2:5])
['Cape Town', 'Tokyo', 'Melbourne']
```

```
print cities in positions from two to five
```

Again, we have to slice consecutive elements. Therefore, we specify the start, which is `2` for `"Cape Town"`, and the stop, which is `5` (because of the plus-one rule) for `"Melbourne"`, but we omit the step because it is `1`. Once more, we can further simplify the three-s rule! How? The **stop** coincides with the **last element of the list**, so we can omit it:

```
[8]: 1 print(cities[2:])
['Cape Town', 'Tokyo', 'Melbourne']
```

```
print cities in positions from two to the
end of the list
```

So far, we have seen the three-s rule applied in its entirety (cell 3), and without start (cell 4), stop (cell 5), and step (cell 2). How else can we simplify it? Let's look at the following example. How do you think the code will look?

6. Slice `"San Diego"`, `"Cape Town"`, and `"Melbourne"`:

```
[9]: 1 print(cities[0:5:2])
['San Diego', 'Cape Town', 'Melbourne']
```

```
print cities in positions from zero to five
with a step of two
```

This time, the elements to slice are not consecutive. We start at `0`, which is the position of `"San Diego"`, we stop at `5` (because of the plus-one rule) for `"Melbourne"`, and we specify the step, which is `2`, because we are slicing every second element. However, as you might have guessed, because the start coincides with the beginning of the list, and the stop coincides with the last element of the list, we can omit both, and rewrite the code above as follows:

```
[10]: 1 print(cities[::2])
['San Diego', 'Cape Town', 'Melbourne']
```

`print cities in positions from the beginning of the list to the end of the list with a step of two`

You have now mastered the three-s rule and learned how to simplify it. How else can we play with it? Let's look at this further representation of the list `cities`:

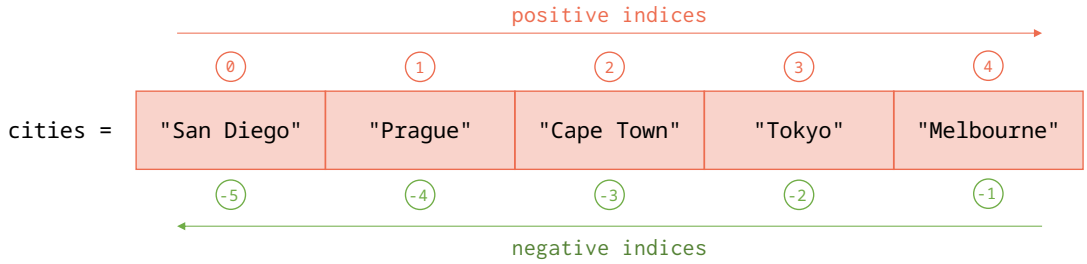


Figure 6.2. In a list, indices can be positive (from left to right) or negative (from right to left).

In Python, each element of a list can be identified by a positive or a negative index. We use **positive indices** when we consider elements **from left to right** and **negative indices** when we consider elements from **right to left**. Positive indices start from 0 and increase of 1 unit (0, 1, 2, etc.). Negative indices start from -1 and decrease of 1 unit (-1, -2, -3, etc.). Note that negative indices do not start from 0 to avoid ambiguity: the element in position 0 is always the first element of the list starting from the left side. When are negative indices convenient? For example, when we are dealing with a very long list. In that case, it would be tedious to count through all elements starting from 0. So we can just count backwards starting from the last element!

How do we use negative indices in slicing? Let's have a look!

7. Slice `"Melbourne"`:

```
[11]: 1 print(cities[4])
Melbourne
```

`print cities in positions 4`

In this example, we extracted `"Melbourne"` as we learned in example 1: by writing its positive index, which is `4`, in between the square brackets. However, `"Melbourne"` is the last element of the list; therefore, it is much more convenient to use its negative index to slice it, like this:

```
[12]: 1 print(cities[-1])
Melbourne
```

`print cities in position minus one`

The advantage of using the negative index is that we do not need to count through all the list elements

to get to know the position of `"Melbourne"`. Since `"Melbourne"` is the last element of the list, we can just write `-1`. This saves us time and eliminates possible errors due to miscounting.

8. Slice all the cities from `"Prague"` to `"Tokyo"` using negative indices:

```
[13]: 1 print(cities[-4:-1])
['Prague', 'Cape Town', 'Tokyo']
```

```
print cities in positions from minus four to minus one
```

This is in an alternative to the code in cell 2. There, we extracted the cities from `"Prague"` to `"Tokyo"` using positive indices, whereas here we want to use negative indices. It might look intimidating, but the reasoning is always the same. The first element we want to extract is `Prague`, which is in position `-4`, therefore the start is `-4`. The last element we want to extract is `Tokyo`, which is in position `-2`, thus the stop is `-1` because of the plus one rule. Like in the previous example, using negative indices can be very convenient when extracting elements from the end of a long list.

In this example, we saw how to use negative indices for the start and the stop. What about the step? A **negative step** allows us **to slice elements in reverse order**, which means from the right to the left. Negative steps can be used with both positive or negative start and stop. This might sound confusing, but we'll clarify it the next three cells. Slicing in reverse order is a very powerful feature, and it's the last thing you need to know to master slicing. Let's have a look!

9. Slice all the cities from `"Tokyo"` to `"Prague"` using positive indices (reverse order):

```
[14]: 1 print(cities[3:0:-1])
['Tokyo', 'Cape Town', 'Prague']
```

```
print cities in positions from three to zero
with a step of minus one
```

When slicing—and coding, in general—it is extremely important **to be aware of the result we expect**. When slicing in reverse order, having the result in mind can really avoid confusion. So, let's start from there. We want to print `"Tokyo"`, `"Cape Town"`, and `"Prague"`. The first element is `"Tokyo"`, which is in position 3, so the start is `3`. The last element is `"Prague"`, which is in position 1. When we slice in reverse order, instead of the plus one rule, we have to use the **minus one rule**, which says that **we must subtract 1 from the position of the last element we want to slice**. Why? This is very intuitive. As we know, for the stop, we always want to take the **next position**. When slicing in **direct order**, the next position is on the **right side** of the last element. Therefore, we add 1 to its index. On the other side, when slicing in **reverse order**, the next position is on the **left side** of the last element. Therefore, we subtract 1 from its index. Now, back to our example. The last element is `"Prague"`, which is in position 1. And because of the minus one rule, the stop is `0`. Finally, we need to define the step. Because the elements are consecutive, the step should be 1, but because we are going in reverse order, we have to put a minus in front of it, so the step becomes `-1`.

In summary, when slicing in reverse order, we have to: (1) make sure we have the first and the last elements clearly in our minds, (2) apply the minus one rule to the stop, and (3) use a negative step.

Let's raise the bar even more now! Look at the next example.

10. Slice all the cities from `"Tokyo"` to `"Prague"` using negative indices (reverse order):

```
[15]: 1 print(cities[-2:-5:-1])
['Tokyo', 'Cape Town', 'Prague']
```

```
print cities in positions from minus two to
minus five with a step of minus one
```

When using negative indices for the start and the stop, the rules are exactly the same as when using positive indices. The first element we want to slice is `"Tokyo"`, which is in position -2, so the start is

`-2`. The last element is `"Prague"`, which is in position `-4`. Because of the minus one rule, we have to subtract 1 from `-4`, therefore the stop is `-5`. And finally, because we are slicing consecutive elements in reverse order, the step is `-1`. As you can now imagine, using negative indices can be very convenient when slicing elements at the end of a very long list in reverse order.

11. Slice all the cities (in reverse order):

```
[16]: 1 print(cities[::-1])
```

```
['Melbourne','Tokyo', 'Cape Town',
'Prague', 'San Diego']
```

`print cities in positions from the beginning of the list to the end of the list with a step of minus one`

The first element to slice is `"Melbourne"`, which is the last element of the list. Therefore, we can omit the start. The last element to slices is `"San Diego"`, which is the first element of the list. Therefore, we can omit the stop too. We just must write the step, which is `-1` because we are slicing consecutive elements in reverse order. Easy to remember!

Last note. Learning slicing might feel overwhelming at first because of all the rules, the use of positive and negative indices, and thinking of lists in direct and reverse order. However, learning slicing properly is fundamental not only because it is **often used in coding**, but also because it allows you to exercise your brain and **strengthen your logical thinking**. Take your time to learn the rules and do the exercises below. You will greatly benefit from it in the following chapters!

✎ Complete the table

Complete the following table to create an overview of slicing in your own words:

Slicing syntax	What it does
`list_name[index]`	
`list_name[start:stop:step]`	
`list_name[:stop:step]`	
`list_name[start::step]`	
`list_name[start:stop]`	
`list_name[negative_index]`	
`list_name[::negative_step]`	
`list_name[::-1]`	

Recap

- To slice one element, we use the rule: `list_name[element_position]`.
- To slice multiple elements, we use the three-s rule: `list_name[start:stop:step]`, where:
  - We can omit start when we slice from the first element of a list, stop when we slice to the last element of a list, and step when we slice consecutive elements of a list.
  - The stop follows the plus one rule when slicing from left to right (direct order), and the minus one rule when slicing from right to left (reverse order).

- The values of `element_position`, `start`, `stop`, and `step` can be:
  - Positive: when considering elements from left to right (direct order).
  - Negative: when considering elements from right to left (reverse order).
- Negative steps are used to invert lists.

Why the plus one rule?

So far, we have learned that each list element is associated with an index or position. However, in Python, each element is actually considered **between two positions**, as represented in Figure 6.3.

Figure 6.3. List representation where each element is in between indices.

Let's re-consider example 2, where we extracted the cities from `"Prague"` to `"Tokyo"`:

```
[3]: 1 print(cities[1:4])
['Prague', 'Cape Town', 'Tokyo']
```

`print cities in positions from one to four`

Using the representation above, we can see that the start is `1` because that is the index that precedes `"Prague"`, the first element to slice. And the stop is `4` because that is the index that follows `"Tokyo"`, the last element to slice.

For many people, considering elements in-between indices is pretty straightforward. For other people, considering that elements have one single index—as we have done so far—is easier. My recommendation is to pick one representation and stick to that. In this book, we will continue to represent list elements with one single index.

Let's code!

1. *Fruits and veggies.* Given the following list:

   ```
   fruits_and_veggies = ["peppers", "apricots", "carrots", "apples", "zucchini", "grapes",
   "cabbage", "oranges", "asparagus", "pears"]
   ```

   Use slicing to extract:

   a. The produce between apples and grapes (included);
   b. All the vegetables;
   c. All the fruits;
   d. The vegetables between carrots and asparagus (included);
   e. The fruits between apples and oranges (included).

2. *Clothes, stationery, and electronics.* Given the following list:

   ```
   objects = ["mobile", "t_shirt", "pencil", "laptop", "hat", "ruler", "tv", "pants", "pen"]
   ```

   Use slicing to extract:

   a. All the clothes;
   b. All the stationery;
   c. All the electronics;
   d. The second and the last stationery items;
   e. The first and the last electronics items;
   f. The first and the second clothing items.

3. *Interior design.* Given the following list:

   ```
   interior_design = ["sofa", "curtain", "lamp", "table", "carpet", "plant", "armchair",
   "blanket", "vase"]
   ```

   Use slicing to extract the following elements in direct order (from left to right), once using positive indices and once using negative indices:

   a. All furniture;
   b. All textiles;
   c. All decorative elements;
   d. The pieces composed of 5 letters (count them by hand, no coding required).

4. *Botanic garden.* Given the following list:

   ```
   botanic_garden = ["tulip", "pine", "poppy", "palm", "rose", "oak", "daisy", "eucalyptus"]
   ```

   Use slicing to extract the following elements, once in direct order (from left to right) and once in reverse order (from right to left):

   a. All flowers;
   b. All trees;
   c. All flowers and trees starting with `p` (find them by hand, no coding required);
   d. `"pine"`, `"rose"`, and `"eucalyptus"`;
   e. All flowers and trees.

5. *Travel agency.* You are the owner of a travel agency and these are the destinations you offer:

   ```
   destinations = ["Boston", "Madrid", "Shanghai", "Cairo", "Mexico City", "Copenhagen", "Seoul",
   "Casablanca", "Lima", "Vienna", "Bangkok", "Nairobi", "Buenos Aires", "Athens", "Manila", "Cape
   Town"]
   ```

   You also have a list containing additional destinations you want to offer in the future:

   ```
   future_destinations = ["Tunis"]
   ```

   a. A new customer comes in and you ask where he would like to go. He replies: Berlin. You check whether Berlin is part of the destination list. If Berlin is part of the list, you say that you sell tickets for Berlin. If Berlin is not part of the destination list, you: (1) tell the customer that you do not sell tickets for Berlin, (2) tell him what European cities are in the destination list, and (3) add Berlin to the list of future destinations.
   b. Because tickets for Berlin are not available, your customer is now thinking about going to Asia. So you tell him the destinations in Asia. He tells you that he forgot the last two Asian places you mentioned; so you tell him again. Then, he says he would have enjoyed going to Hong Kong. But Hong Kong is not an available destination, so you add it to the list of future destinations.
   c. Now you ask your customer if he is interested in going to the American continent, and he replies: Toronto. You check whether Toronto is part of the list. Similarly to what you did for Berlin, if Toronto is part of the list, you say that you sell tickets for Toronto. If Toronto is not part of the destination list, you: (1) tell your customer that you do not sell tickets for Toronto, (2) tell him what cities on the American continent are in the destination list, and (3) add Toronto to the list of future destinations.

d. The customer is still undecided. You think he might be interested in a trip to Africa, so you tell him all the destinations in Africa. He finally tells you that he wants to go to Cape Town! So you replace Cape Town from the list of destinations with Tunis from the list of future destinations, and remove Tunis from the future destination list.
e. The customer is finally gone, and you want to create a flyer with all the destinations you offer. To do so, you add the three new future destinations to the list of current destinations (in what order?), and you print the destinations you offer for each continent. While doing so, you notice that Africa only has four destinations. So you add one African destination to the destination list before printing out the African destinations. And, finally, you close the shop, go home, and enjoy your evening after a hard day of work!

7. Senses, planets, and a house

Changing, adding, and removing list elements using slicing

Now that you know everything about slicing, let's see how to use it to manipulate lists—that is, how to change, add, or remove list elements. Download and open Jupyter Notebook number 7 and follow along. Similarly to the previous chapter, cover the code in these pages with a sheet of paper. First, try to guess the commands to execute, and then compare with the code below. Don't forget to read the code aloud!

1. Senses

Let's first learn how to **change list elements** using **slicing** and **assignment**.

- Given the following list:

```
[1]:  1  senses = ["eyes", "nose", "ears", "tongue",      senses is assigned eyes, nose, ears,
          "skin"]                                         tongue, skin
      2  print(senses)                                    print senses
['eyes', 'nose', 'ears', 'tongue', 'skin']
```

The list `senses` contains five strings: `"eyes"`, `"nose"`, `"ears"`, `"tongue"`, and `"skin"` (line 1), and we print it (line 2).

- Replace `"nose"` with `"smell"`:

```
[2]:  1  senses[1] = "smell"                              senses in position one is assigned smell
      2  print(senses)                                    print senses
['eyes', 'smell', 'ears', 'tongue', 'skin']
```

To change **one list element**, we **assign the new value to the list sliced in the element's position**. In this case, the element we want to replace—`"nose"`—is in position 1. So, we slice the list in position 1, and we assign the new string `"smell"` (line 1). Then, we print the list to check whether the change is correct (line 2).

At this point, you might ask: Why do I have to learn list manipulation using slicing when I already know how to do it with methods? For at least three reasons! First reason: to **reduce the possibility of errors**. The code at line 1 is an alternative to the code we learned in Chapter 5, where we used three methods to replace an element, that is:

```
[]:  1  nose_index = senses.index("nose")                 nose index is assigned senses dot index nose
     2  senses.pop(nose_index)                            senses dot pop nose index
     3  senses.insert(nose_index, "smell")                senses dot insert at position nose index smell
```

By using slicing, we reduce the number of commands from three to one, and we do not need to create an extra variable—that is, `nose_index`. By writing less code, we minimize the possibility of making errors! Second reason: **slicing makes code writing faster**. Imagine you have to replace 4 elements. With slicing, you would have to write just 4 lines of code; instead, with list methods, the number of lines required would be 12—that is, 3 lines for each of the 4 elements! And finally, the third reason: transitioning from list methods to list slicing allows us to shift from a more concrete to a more **abstract**

way of thinking. As you know, when using list methods, we use a coding language that is more similar to a natural language. Method names, in fact, are words in the English vocabulary, such as remove, insert, etc. Instead, when slicing, we use numbers—which represent element positions—and thus we use (numerical) symbols in place of words. As you can see, we are building more and more the abstract thinking that coding requires. So let's keep going!

- Replace `"tongue"` and `"skin"` with `"taste"` and `"touch"`:

```
[3]:  1  senses[3:5] = ["taste", "touch"]
      2  print(senses)
['eyes', 'smell', 'ears', 'taste', 'touch']
```

```
senses in positions from three to five
is assigned taste, touch
print senses
```

To change **several elements** in a list, first we **slice the elements we want to substitute**, and then we **assign them a list containing the new values**. In this case, we want to replace two elements, so we slice using the three-s rule. The start is the position of `"tongue"`, which is `3`, and the stop is the position of `"skin"`, which is 4, but it becomes `5` because of the plus one rule. The step is `1`, so we can omit it. To the sliced list, we assign a list containing the new elements, which are the strings `"taste"` and `"touch"` (line 1). Finally, we print the list to make sure that the change occurred correctly (line 2).

- Replace `"eyes"` and `"ears"` with `"sight"` and `"hearing"`:

```
[4]:  1  senses[0:3:2] = ["sight", "hearing"]
      2  print(senses)
['sight', 'smell', 'hearing', 'taste', 'touch']
```

```
senses in positions from zero to three
with a step of two is assigned sight,
hearing
print senses
```

Like in the previous example, we want to replace several elements. So, we begin by slicing the list. The start is the position of `"eyes"`, which is `0` (and can be omitted). The stop is the position of `"ears"`, which is 2, but it becomes `3` because of the plus one rule. The two elements are not consecutive, thus we have to write the step, which is `2`. Finally, we assign the list containing the two strings we want to add: `"sight"` and `"hearing"`. Note that the two elements we want to replace are not consecutive, but Python takes care of placing `"sight"` and `"hearing"` in the right positions (line 1). At the end, we print the final list to check the changes we made (line 2).

2. Planets

To **add new elements** to a list, we can use **slicing combined with list concatenation and assignment**. How? Let's have a look at the following examples!

- Given the following list:

```
[5]:  1  planets = ["Mercury", "Mars", "Earth", "Neptune"]
      2  print(planets)
['Mercury', 'Mars', 'Earth', 'Neptune']
```

```
planets is assigned Mercury,
Mars, Earth, Neptune
print planets
```

We begin with the list `planets`, which contains four strings: `"Mercury"`, `"Mars"`, `"Earth"`, and `"Neptune"` (line 1), and we print it (line 2).

- Add `"Jupiter"` at the end of the list:

```
[6]:
1  planets = planets + ["Jupiter"]

2  print(planets)
['Mercury', 'Mars', 'Earth', 'Neptune', 'Jupiter']
```

```
planets is assigned planets
concatenated with Jupiter
print planets
```

To add **an element at the end of a list**, we (1) embed it in a list, (2) concatenate it to the original list, and (3) assign the result to the original list. It's less complicated than it sounds! Let's start from the far right of line 1. We take the new element `"Jupiter"`—which is a string—and we enclose it in square brackets to **transform it into a list**: `["Jupiter"]`. Why do we need to change `"Jupiter"` data type? Because we want to add it to the list `planets` using concatenation. And, as in string concatenation, we can concatenate only strings with strings, in **list concatenation**, we can concatenate only **lists with lists**. Note that list concatenation works the same way as string concatenation. Finally, we **assign the result of the operation to the original list** `planets` to actually change it. It is common to say that we **reassign** the result to the original list. This whole operation constitutes an alternative to the method `.append()`. Finally, we print the modified list to check the correctness of our code (line 2).

You may have realized that in this example there is no slicing! This is because it's a special case, where we add an element at the end of a list—it would be similar if we added an element at the beginning of a list. In our example, we could write `planets[0:4] + ["Jupiter"]`, where `planets[0:4]` slices all the elements in the list, but that would be redundant. Let's see slicing in action in the next two cells!

- Add `"Venus"` between `"Mars"` and `"Earth"`:

```
[7]:
1  planets = planets[0:2] + ["Venus"] +
   planets[2:5]

2  print(planets)
['Mercury', 'Mars', 'Venus', 'Earth',
'Neptune', 'Jupiter']
```

```
planets is assigned planets in positions
from zero to two concatenated with Venus
concatenated with planets in positions from
two to five
print planets
```

In this case, we want to add an element **in the middle** of a list. To do so, we (1) split the list in two segments at the position where we want to insert the new element, (2) insert the new element as a list by concatenating it with the two list segments, and (3) assign the result to the original list. Like before, it's easier than it sounds! We want to **split the list** between `"Mars"` and `"Earth"`. The first list segment will contain `"Mercury"` and `"Mars"`. Thus, we slice `planets` starting from position `0`, corresponding to `"Mercury"`, and stopping in position `2` for the plus one rule—`"Mars"` is in position 1. The second list segment will contain `"Earth"`, `"Neptune"`, and `"Jupiter"`. So, we slice starting from position `2`, corresponding to `"Earth"`, and stopping in position `5` for the plus one rule—`"Jupiter"` is in position 4. In between the two list segments, we **concatenate a new list** containing the string `"Venus"`—like before, we have to change `"Venus"` from a string to a list. We conclude the operation by **assigning the concatenation result to the original list** (line 1). As you may have realized, this line is an alternative to the method `.insert()`. Finally, we print the obtained list to check the correctness of the operation (line 2). A possible way to think about the whole procedure is to consider a list like a toy train, where each list element is a car. When we want to insert a new car, for example a restaurant car, we split the train into two parts in the position where we want the new car to be. Then, we add the first part of the train to the left side of the restaurant car, and the second part of the train to the right side of the restaurant car.

- Add `"Uranus"` and `"Saturn"` between `"Neptune"` and `"Jupiter"`:

```
[8]:
1  planets = planets[:5] + ["Uranus", "Saturn"] +
   planets[5:]
2  print(planets)
```

```
['Mercury', 'Mars', 'Venus', 'Earth', 'Neptune',
'Uranus', 'Saturn', 'Jupiter']
```

```
planets is assigned planets in
positions from the beginning of the
list to five concatenated with Uranus,
Saturn concatenated with planets in
positions from five to the end of the
list
print planets
```

To insert **several consecutive elements in the middle of a list**, we use the same approach as the one above. We slice the first part of the list `planets` from the beginning—this time, we omit the start for simplicity—to `5`, which corresponds to the position of `"Neptune"` plus 1. Then, we concatenate the two new elements `"Uranus"` and `"Saturn"` **embedded in a list**. Finally, we concatenate the remaining part of the list `planets`, starting from the position of `"Jupiter"`, which is `5`, and stopping at the end of the list—we omit the stop for simplicity (line 1). Now a trick! You may have noticed that the stop of the first list segment coincides with the start of the second list segment—they are both `5`. This is because of the plus one rule applied to the stop of the first list segment. Thus, when adding new elements using slicing, we can **just count the stop of the first list segment**. The start of the second list segment will automatically be the same!

An important note before continuing: in the past three examples, we started **analyzing code from the right side of the assignment operator**. Focusing on that side is quite common because it is where we define variable changes and operations. Sometimes, we can even start writing code on the right side of the assignment operator, and then type the appropriate variable name on the left side. It's very common to start **analyzing or writing code backwards**!

3. A house

To **delete** list elements, we can use the **keyword `del` combined with list slicing**. This is very easy. Let's have a look!

- Given the following list:

```
[9]:
1  house = ["kitchen", "dining room", "living room",
   "bedroom", "bathroom", "garden", "balcony",
   "terrace"]
2  print(house)
```

```
['kitchen', 'dining room', 'living room', 'bedroom',
' bathroom', 'garden','balcony','terrace']
```

```
house is assigned kitchen,
dining room, living room,
bedroom, bathroom, garden,
balcony, terrace
print house
```

We start with a list called `house` containing 8 strings (line 1), and we print it (line 2).

- Delete `"dining room"`:

```
[10]:
1  del house[1]
2  print(house)
```

```
['kitchen', 'living room', 'bedroom', ' bathroom',
'garden', 'balcony','terrace']
```

```
del house in position one
print house
```

To **delete one element** in a list, we can use **`del` followed by the list sliced at the position of the element we want to delete**. In this case, we want to remove the string `"dining room"`, which is in position 1, so we write the keyword `del` followed by `house[1]` (line 1). **`del`** is a **keyword** that allows us to **delete a variable or some elements in a variable**—in this case, some elements in a list. Like the other keywords we have seen so far—for example, `if` and `else`—`del` is written in bold green in Jupyter Notebook. As you may have realized, using `del` and slicing is an alternative to using the list methods `.pop()` or `.remove()`. After removing the element, we print the list for checking (line 2).

- Delete `"garden"` and `"balcony"`:

```
[11]: 1  del house[4:6]
      2  print(house)
['kitchen', 'living room', 'bedroom', ' bathroom', 'terrace']
```

```
del house in positions
from four to six
print house
```

To **delete consecutive elements** from a list, we use the same syntax as above: we write the keyword `del` followed by the list sliced at the positions of the elements we want to delete. In this example, the start is the position of `"garden"`, which is `4`, and the stop is the position of `"balcony"`, which is 5 and becomes `6` because of the plus one rule (line 1). Then, we print the reduced list (line 2).

- Delete `"kitchen"`, `"bedroom"` and `"terrace"`:

```
[12]: 1  del house[::2]
      2  print(house)
['living room', ' bathroom']
```

```
del house in positions from the beginning of the
list to the end of the list with a step of two
print house
```

To **delete non-consecutive elements** in a list, we use the same procedure as the one above once more: we write the keyword `del`, followed by the list sliced at the positions of the elements we want to remove. In this example, the start corresponds to `"kitchen"`, which is the first element of the list, so we can omit it. The stop corresponds to `"terrace"`, which is the last element in the list, so we can omit it as well. And the step is `2` because we want to delete every second element (line 1). Finally, we print the remaining list (line 2).

- Delete `"house"`:

```
[13]: 1  del house
      2  print(house)
---------------------------------------
NameError          Traceback (most recent call last)
Cell In[13], line 2
       1 del house
----> 2 print (house)
NameError: name 'house' is not defined
```

```
del house
print house
```

Finally, we want to delete the whole house! To **delete a variable**, we write the keyword **`del` followed by the variable name**—`house` in our example (line 1). This time, we get an error when we print `house`. It's a name error, indicating that the variable does not exist anymore (line 2). This is a good error, telling us that we succeeded in our aim: we deleted the whole variable `house`!

✎ Complete the table

In the previous four chapters, you learned how to manipulate lists using methods or slicing. Complete the table below to compare the two different techniques:

List operation	List methods	List slicing
Adding an element at the beginning of a list		
Adding an element in the middle of a list		
Adding an element at the end of a list		
Changing an element in a list		
Deleting an element in a list		

- What is different if you want to add, change, or delete several elements? Write your answer here:

Recap

- To change list elements, we can use slicing and assignment.
- To add list elements, we can combine slicing, concatenation, and assignment.
- To delete list elements, we can use the keyword `del` and slicing.
- List concatenation is performed using the `+` symbol and works the same way as string concatenation.

What is a Jupyter Notebook kernel?

The **kernel** is the component of Jupyter Notebook that **executes code**. When we run a cell, the kernel tells Python to execute computations and save variables. Every notebook has its own kernel. And when we open a notebook, a new kernel is automatically created and is ready to execute code. Now you may ask: Why do we care about the kernel? Because sometimes we need to interrupt it or restart it to continue running code. Let's see what this means.

Interrupting the kernel. Consider two cells containing code. In the first cell, we ask a question using the function `input()`. In the second cell, we print the variable containing the answer. We want to execute the code, so we run the first cell. On the left side, we get the star symbol between the square brackets, indicating that the code is being executed. But before entering the answer, we mistakenly run the second cell! Now the second cell also gets the star symbol between the

square brackets on the left side, like this:

```
[*]: 1 name = input("What's your name?")
     What's your name?
```

name is assigned input What's your name?

```
[*]: 1 print(name)
```

print name

In this case, the situation is frozen and no code gets executed! So we need to interrupt the kernel. To do that, we can either go to the JupyterLab top bar, then to *Kernel*, and then *Interrupt Kernel*, or we can go to the Jupyter Notebook top bar and press the interrupt kernel button—that is, item 7 in Figure 7.1.

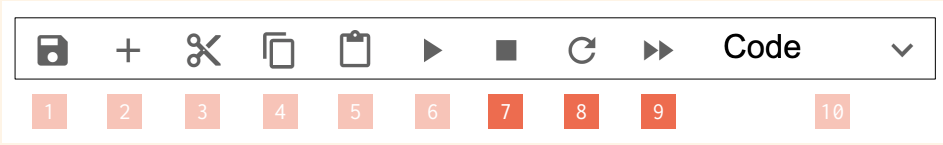


Figure 7.1. Jupyter Notebook top bar: (1) save notebook, (2) add cell, (3) cut cell, (4) copy cell, (5) paste cell, (6) run cell, (7) interrupt kernel, (8) restart kernel, (9) restart kernel and run whole notebook, and (10) define cell as code or markdown.

After interrupting the kernel, the star symbols in between square brackets disappear, and we can run each cell again.

Restarting the kernel. Consider the list `house` from this chapter. Let's say that we want to delete the element `"dining room"`, as we did in one of the examples above. But, by mistake, we type the wrong slicing index—that is, `0` instead of `1`—deleting `"kitchen"` in place of `"dining room"`, like this:

```
[9]: 1 house = ["kitchen", "dining room",
       "living room", "bedroom", "bathroom",
       "garden", "balcony", "terrace"]
```

house is assigned kitchen, dining room, living room, bedroom, bathroom, garden, balcony, terrace

```
[10]: 1 del house[0]
      2 print(house)
      ['dining room', 'living room', 'bedroom',
      'bathroom', 'garden', 'balcony','terrace']
```

del house in position zero
print house

We want to restore the original variable `house` and rerun the corrected version of our code—`del house[1]`— to obtain the correct result. How do we go back? By restarting the kernel! To do that, we can either go to the JupyterLab top bar, then *Kernel*, and then *Restart Kernel*; or we can go to the Jupyter Notebook top bar and press the curved arrow (item 8 in Figure 7.1). Then, we can rerun the cells of the notebook. As an alternative, we can restart the kernel and rerun all notebook cells at once by going to the JupyterLab top bar, then *Kernel*, and then *Restart Kernel and Run all Cells*, or to the Jupyter Notebook top bar and pressing the symbol with two arrow tips (item 9 in Figure 7.1). You may ask: do I really have to restart the kernel every time I make a mistake? Not really. In this case, one could just rerun the first cell to bring the variable `house` back to its original value, and then rerun the second cell with the corrected code. However, when dealing with multiple variables or making several mistakes with a single variable, it is good practice to reset the kernel and restart from scratch.

Let's code!

1. *Stephanie Shirley.* Do you know the story of Stephanie Shirley? Let's see what she did! Given the following list:
   `stefanie_shirley = ["In 1962", "Stephanie Shirley", "founded", "a software company", "employing", "only women", "working from home"]`
   Do the following using list slicing:
   a. Replace `"founded"` with `"thrived"`.
   b. Remove the element in position 0 (first element).
   c. Replace `"employing"` with `"transferred ownership"`.
   d. Add `"and over the years"` between `"thrived"` and `"a software company"`.
   e. Replace `"only women"` with `"to her staff"`.
   f. Insert `"gradually"` in position 4 (fifth element).
   g. Replace `"a software company"` with `"she"`.
   h. Add `"70 millionaires"` at the end of the list.
   i. Remove `"Stephanie Shirley"`.
   j. Replace `"working from home"` with `"creating"`.
   k. Insert `"The business"` at the beginning of the list.
   Redo the same using list methods.
2. *Tim Berners-Lee.* What did Tim Berners-Lee invent? Let's find it out! Given the following list:
   `tim_bernerslee = ["Tim Berners-Lee", "invented", "the World Wide Web", "in 1989", "at CERN in Geneva", "info.cern.ch", "was", "the address of", "the world's first website and Web server"]`
   Do the following using list slicing:
   a. Remove `"info.cern.ch"`.
   b. Replace `"was"` with `"consists of"`.
   c. Remove the element in position 1 (second element).
   d. Add `"all over the world"` at the end of the list.
   e. Replace `"the world's first website and Web server"` with `"about 75 million servers"`.
   f. Remove the element in position 0 (first element).
   g. Replace `"in 1989"` with `"Nowadays"`.
   h. Remove the element in position 0 (first element).
   i. Replace `"at CERN in Geneva"` with `"it is estimated that"`.
   j. Add `"the internet"` in position 2 (third element).
   k. Remove the element in position 4 (fifth element).
   Redo the same using list methods.
3. *Alan Turing.* What happened thanks to Alan Turing's contributions? Let's discover it! Given the following list:
   `alan_turing = ["Turing", "created", "an electromechanical machine", "to crack", "the Nazi Navy's", "Enigma Code"]`
   Do the following using list slicing:
   a. Replace `"the Nazi Navy's"` with `"shortened the war"`.
   b. Insert `"by two years"` in position 5 (sixth element).
   c. Replace `"an electromechanical machine"` with `"his contribution"`.
   d. Add `"saving millions of lives"` to the end.
   e. Replace `"created"` with `"that"`.
   f. Remove `"to crack"`.
   g. Replace `"Turing"` with `"It is estimated"`.
   h. Remove the element in position 5 (sixth element).
   Redo the same using list methods.

PART 3

INTRODUCTION TO THE FOR LOOP

In this part, you will learn about the for loop, which is one of the two loops in coding—the other is the while loop. You will learn the syntax of the for loop and how to use it to search elements in a list, modify a list, and automatically create new lists. Let's go!

8. My friends' favorite dishes

for... in range()

The `for` loop is one of the most important constructs in coding because it allows us to repeatedly execute commands. What does this mean and how does it work? Time to open Jupyter Notebook 8 and answer these questions! Read the following example out loud and try to understand it:

- Here are a list of my friends and a list of their favorite dishes:

```
[]:  1  friends = ["Geetha", "Luca", "Daisy", "Juhan"]
     2  dishes = ["sushi", "burgers", "tacos", "pizza"]
```

```
friends is assigned Geetha, Luca, Daisy, Juhan
dishes is assigned sushi, burgers, tacos, pizza
```

- These are all my friends:

```
[]:  1  print("My friends' names are:")
     2  print(friends)
```

```
print My friends' names are:
print friends
```

- These are my friends one by one:

```
[]:  1  for index in range(0, 4):
     2      print("index:" + str(index))
     3      print("friend:" + friends[index])
```

```
for index in range from zero to four
print index: concatenated with str index
print friend: concatenated with friends in position index
```

- These are all their favorite dishes:

```
[]:  1  print("Their favorite dishes are:")
     2  print(dishes)
```

```
print Their favorite dishes are:
print dishes
```

- These are their favorite dishes one by one:

```
[]:  1  for index in range(0, 4):
     2      print("index:" + str(index))
     3      print("dish:" + dishes[index])
```

```
for index in range from zero to four
print index: concatenated with str index
print dish: concatenated with dishes in position index
```

- These are my friends, with their favorite dishes one by one:

```
[]:  1  for index in range(0, 4):
     2      print("My friend " + friends[index] +
            "'s favorite dish is " + dishes[index])
```

```
for index in range from zero to four
print My friend concatenated with friends in position index concatenated with 's favorite dish is concatenated with dishes in position index
```

Get some hints about what the code does by completing the next exercise.

Match the sentence halves

1. The for loop allows us	a. a start and a stop as an argument.
2. The variable index	b. how many times commands are repeated.
3. In the first loop, the variable index	c. to repeat the indented commands.
4. The built-in function range() determines	d. changes value at each loop.
5. The built-in function range() can take	e. is assigned the value 0.

Computational thinking and syntax

Let's start by running the first cell:

```
[1]:
1  friends = ["Geetha", "Luca", "Daisy", "Juhan"]
2  dishes = ["sushi", "burgers", "tacos", "pizza"]
```

```
friends is assigned Geetha, Luca, Daisy, Juhan
dishes is assigned sushi, burgers, tacos, pizza
```

There are two lists—`friends` and `dishes`—and each contains four strings.

Let's run the second cell:

```
[2]:
1  print("My friends' names are:")
2  print(friends)
```

```
My friends' names are:
['Geetha', 'Luca','Daisy','Juhan']
```

```
print My friends' names are:
print friends
```

We print the string `"My friends' names are:"` (line 1) and the content of the list `friends` (line 2).

Let's now run the third cell, which contains the first `for` loop:

```
[3]:
1  for index in range(0, 4):
2      print("index:" + str(index))
3      print("friend:" + friends[index])
```

```
index: 0
friend: Geetha
index: 1
friend: Luca
index: 2
friend: Daisy
index: 3
friend: Juhan
```

```
for index in range from zero to four
print index: concatenated with str index
print friend: concatenated with friends in position index
```

The code prints the position and the value of each list element by repeating lines 2 and 3 four times. How does this happen? Let's start from line 1, which is the **header** of the `for` loop. It consists of five components:

- **`for`**: The keyword starting a `for` loop. Like all keywords, it is bold green in Jupyter Notebook.
- **`index`**: A variable that is assigned a different value at each loop iteration (we'll talk more about this in a bit).
- **`in`**: A membership operator, the same that you learned in the construct `if...in/else` in Chapter 3.
- **`range()`**: A built-in Python function. You can recognize this as a function because it is followed by round brackets and is colored green in Jupyter Notebook—like `input()` and `print()`. We'll talk more

about `range()` in a bit too.
- `:` that is, the colon punctuation.

To better understand what this line does, let's begin from the built-in function `range()`. It takes two arguments: `0` and `4`. They are two integers that we can call—guess what?—start and stop! So, what does `range()` do? Create a separate cell in the notebook, and write and run the following code:

```
1  list(range(0, 4))                    list range from zero to four
[0,1,2,3]
```

The built-in function **`range()` returns a sequence of integers spanning from the start (included) to the stop (excluded because of the plus one rule)**. In this example, the integers go from `0` to `3`, and—guess what again?—they **correspond to the indices of the elements** of the list `friends`! Why is there `list()`? This is another built-in function that we write here for a proper print. Don't worry too much about it for now. Let's focus on understanding the `for` loop!

What do we do with the list of integers created by `range()`? We assign them to the variable `index`! At each code repetition—or **loop**, or **iteration**—`index` is subsequently assigned a number created by `range()`. That is, in the first loop, `index` is assigned `0`; in the second loop, `index` is assigned `1`; and so on. The variable `index` could be called any name—for example, `loop_id` or `iteration_number`—but it is common to call it `index`. Now, what can we do with the variable `index`? At least two things!

First, we can print `index` **to keep track of which loop is getting executed**, like we do at line 2. In the first loop, `index` is assigned `0`, so we print `"index: 0"`. In the second loop, `index` is assigned `1`, so we print `"index: 1"`—and so on. Why is `str()` here? Because we can concatenate only strings with strings, and `index` is an integer! So, we need to change the variable type of `index` from integer to string. And to do that, we can use the built-in function **`str()`**, which **transforms a variable into a string**.

Second, we use `index` **to automatically slice list elements one by one**. As you now know, `index` changes at every iteration, and it is assigned values that go from the beginning of a list—that is, `0`—to the end of a list—in this case `3`. Let's look at line 3 of the cell above. In the first loop, when `index` is assigned `0`, `friends[index]` is the same as `friends[0]`—that is, `"Geetha"`. In the second loop, when `index` is assigned `1`, `friends[index]` is the same as `friends[1]`, i.e., `"Luca"`. And so on.

The lines below the header—in this example, lines 2 and 3—are called the **body** of the `for` loop. They are always **indented**, and there can be as many as we want. They get executed for a number of times determined by the sequence of numbers created by the function `range()`.

Before moving to the next cell, let's review what the code in cell 3 does. We have to go through the three lines of code for a total of four times, like this:
- In the first iteration: at line 1, `index` takes the value `0`; at line 2, we print `index: 0`; and at line 3, we print `friends` in position `0`, that is, `friend: Geetha`.
- In the second iteration: at line 1, `index` takes the value `1`; at line 2, we print `index: 1`; and at line 3, we print `friends` in position `1`, that is, `friend: Luca`.
- In the third iteration: at line 1, `index` takes the value `2`; at line 2, we print `index: 2`; and at line 3, we print `friends` in position `2`, that is, `friend: Daisy`.
- In the fourth iteration: at line 1, `index` takes the value `3`; at line 2, we print `index: 3`; and at line 3, we print `friends` in position `3`, that is, `friend: Juhan`.

Being aware of what happens at each loop is fundamental to make sure that our code does what we expect. Any time you are uncertain about what is happening in a `for` loop, **think about your code line**

by line and iteration by iteration, like we did right above. If the code is particularly complicated, you can also **create a table**, where you can keep track of each line at each iteration, like this:

Loop	`for index in range(0,4):`	`print("index:"+str(index))`	`print("friend:"+friends[index])`
First	`index is 0`	`index: 0`	`friend: Geetha`
Second	`index is 1`	`index: 1`	`friend: Luca`
Third	`index is 2`	`index: 2`	`friend: Daisy`
Fourth	`index is 3`	`index: 3`	`friend: Juhan`

Before going to the next cell, let's define the `for` loop:

> A **`for` loop** is the **repetition** of a group of commands
> for a **determined** number of times

This definition summarizes the two main features of a `for` loop:

1. We execute the **commands that are in the body** of the `for` loop several times.
2. The **number of times is known** and is determined by a sequence of numbers created by the built-in function `range()`.

Let's continue with cell 4:

```
[4]:  1  print("Their favorite dishes are:")
      2  print(dishes)
```

```
Their favorite dishes are:
['sushi', 'burgers', 'tacos', 'pizza']
```

```
print Their favorite dishes are:
print dishes
```

We print the string `"Their favorite dishes are:"` (line 1) and the content of the list `dishes` (line 2).

Let's run cell 5, which contains another `for` loop:

```
[5]:  1  for index in range(0, 4):
      2      print("index:" + str(index))
      3      print("dish:" + dishes[index])
```

```
index: 0
dish: sushi
index: 1
dish: burgers
index: 2
dish: tacos
index: 3
dish: pizza
```

```
for index in range from zero to four
print index: concatenated with str index
print dish: concatenated with dishes in
position index
```

The header is the same as that of the `for` loop we met at cell 3, including the start and the stop of the built-in function `range()`. Also, line 2—where we print the index value at each iteration—is the same. This time, at line 3 we print the dish names one by one. Once again, let's go through the code one iteration at a time:

- In the first iteration: at line 1, `index` takes the value `0`; at line 2, we print `index: 0`; and at line 3, we print `dishes` in position `0`, that is, `dish: sushi`.

- In the second iteration: at line 1, `index` takes the value `0`; at line 2, we print `index: 1`; and at line 3, we print `dishes` in position `1`, that is, `dish: burgers`.
- In the third iteration: at line 1, `index` takes the value `0`; at line 2, we print `index: 2`; and at line 3, we print `dishes` in position `2`, that is, `dish: tacos`.
- In the fourth iteration: at line 1, `index` takes the value `0`; at line 2, we print `index: 3`; and at line 3, we print `dishes` in position `3`, that is, `dish: pizza`.

Finally, let's run the last cell:

```
[6]:
1  for index in range(0, 4):
2      print("My friend " + friends[index] +
       "'s favorite dish is " + dishes[index])
```

```
for index in range from zero to four
print My friend concatenated with friends
in position index concatenated with 's
favorite dish is concatenated with dishes
in position index
```

```
My friend Geetha's favorite dish is sushi
My friend Luca's favorite dish is burgers
My friend Daisy's favorite dish is tacos
My friend Juhan's favorite dish is pizza
```

Once again, there is a `for` loop. The header is the same as that in the two previous examples: we create a sequence of integers that go from 0 to 3, and we provide them to the variable `index`, one by one at each iteration (line 1). The body of the `for` loop is constituted of one line of code, where we print a sentence composed of four parts, concatenated to each other. The first and the third parts are two strings—`"My friend "` and `"'s favorite dish is "`. The second and the fourth parts are the elements of the lists `friends` and `dishes` sliced at position `index` (line 2). As you'll notice, we can use `index` to **simultaneously slice several lists of the same length at the same position** within one `for` loop.

Last note: beyond the start and the stop, the built-in function `range()` can also take a step as an argument, like this:

```
[6]:
1  for index in range(0, 4, 1):
```

```
for index in range from zero to four with
a step of one
```

As for the start and the stop, the step also works exactly the same way as it does in slicing (Chapter 6). In this chapter examples, we omitted the step because it is `1`—that is, we take all the elements of the list. You will play with different step values in the coding exercises at the end of this chapter.

✎ Fill in the gaps

Complete the following sentences to summarize the `for` loop syntax and functionality in your own words:

1. A for loop is _____ .
2. A for loop header is _____ .
3. A for loop body is _____ .
4. for is a _____ and is colored _____ in Jupyter Notebook.

5. index is a _____ and is colored _____ in Jupyter Notebook. It receives _____ .
6. range() is a _____ and is colored _____ in Jupyter Notebook. It can take three arguments: _____ , _____ , and _____ . It returns _____ .
7. An iteration or loop is _____ .

Recap

- A `for` loop is the repetition of commands for a defined number of times.
- When the `for` loop is used to slice a list, the number of times coincides with the list length.
- The generic syntax of a `for` loop header is: `for index in range(start, stop, step):`.
- The body of a `for` loop is indented and can contain as many lines of code as needed.
- `range()` is a Python built-in function that creates a sequence of integers spanning from the start (included) to the stop (excluded).
- `str()` is a Python built-in function that converts a variable into a string.

Dealing with IndexError and IndentationError

When executing a `for` loop, we might encounter two errors: index errors and indentation errors. Let's see why they happen and how to fix them!

Index error. Let's modify the example in cell 3 by changing the stop to 5 (instead of 4). When we run the cell, we get the following error.

[3]:
```
1 for index in range(0, 5):
2     print("index:" + str(index))
3     print("friend:" + friends[index])
```

```
for index in range from zero to five
print index: concatenated with str index
print friend: concatenated with friends
in position index
```

```
index: 0
friend: Geetha
index: 1
friend: Luca
index: 2
friend: Daisy
index: 3
friend: Juhan
index: 4
---------------------------------------
IndexError       Traceback (most recent call last)
Cell In[3], line 3
       1 for index in range (0, 5):
       2     print("index: " + str(index))
----> 3     print("friend: " + friends[index])
IndexError: list index out of range
```

↳

Let's decipher the error message. As you know from Chapter 2, we start reading from the last line, which informs us about the type of error: `IndexError: list index out of range`. This means that we are **trying to slice a list in a position that does not exist**. Where do we do this? Let's look for the arrow. It points to line 3, where we slice `friends` in position `index`. What's the value of `index`? From the last line of the printouts, we can see that `index` is `4`. Thus, we are trying to slice the list `friends` in position `4`, which does not exist. Fixing this error is easy: we just correct the stop in `range()` to `4`.

Indentation error. The indentation error is very easy to recognize and fix. Let's look into this example:

```
[3]:  1  for index in range(0, 4):
      2  print("index:" + str(index))
```

```
for index in range from zero to four
print index: concatenated with str index
```

```
  Cell In[3], line 2
    print("index: " + str(index))
    ^
IndentationError: expected an indented block
```

Again, we start reading from the last line of the error message, which says: `IndentationError: expected an indented block`. This means that we **did not indent a line of code**. Where? The message says `line 2` at the end of its first line. The fix is straightforward: we just indent line 2. A last note: Jupyter Notebook (and other editors) help us avoid the indentation error by positioning the cursor correctly when we press *Enter* after a line terminated by a colon (`:`)—that is, after a `for` loop header (this chapter); an `if/else` condition (Chapter 3); and a `while` loop header (Chapter 17), a function definition (Chapter 28), or a class definition (Chapter 35), as you will see in the coming chapters.

Let's code!

1. For each of the following scenarios, create code similar to that presented in this chapter.
   a. *Capitals of the world.* Write two lists, one containing countries of the world and the other containing their capital cities. First, print all the countries as a list and all the countries one by one. Then, print all the cities as a list and all the cities one by one. Finally, print each country with its capital.
   b. *Animals of the world.* Write two lists, one containing animals of the world and one containing the continents (or countries) where they live. First, print all the animals as a list and all the animals one by one. Then, print all the continents as a list and all the continents one by one. Finally, print each animal with the continent where it lives.
2. *Mountains and rivers.* Given the following list:

   ```
   mountains_rivers = ["everest", "mississipi", "yosemite", "nile", "mont blanc", "amazon"]
   ```

   Print:
   a. All elements as a list;
   b. All elements one by one using a `for` loop;
   c. Mountains using slicing;
   d. Mountains one by one using a `for` loop (tip: remember that `range()` can have three arguments: start, stop, and step);
   e. Rivers using slicing;
   f. Rivers one by one using a `for` loop (what start do you use?);

g. All elements in reverse order using slicing;
h. All elements in reverse order, one by one, using a `for` loop (what start, stop, and step do you use?).

3. *Wild animals.* Given the following list:

```
wild_animals = ["eagle", "bear", "parrot", "tiger", "pelican", "coyote"]
```

Print:

a. All animals as a list;
b. All animals one by one using a `for` loop;
c. Mammals using slicing;
d. Mammals one by one using a `for` loop;
e. Birds using slicing;
f. Birds one by one using a `for` loop (what start do you use?);
g. All animals in reverse order using slicing;
h. All animals, one by one, in reverse order using a `for` loop.

9. At the zoo

For loop with if... ==... / else...

Can we combine `for` loops and `if/else` constructs? Yes! How? Open Jupyter Notebook 9 and follow along. Read the following example aloud, and try to understand how it works:

- You are at the zoo and you write down a list of some animals you see:

```
[]:
1  animals = ["giraffe", "penguin", "dolphin"]
2  print(animals)
```

```
animals is assigned giraffe, penguin, dolphin
print animals
```

- Then you print the animals one by one:

```
[]:
1  # for each position in the list
2  for i in range(0, len(animals)):
3      print("--- Beginning of loop ---")
4      # print each element and its position
5      print("The element in position " +
       str(i) + " is " + animals[i])
```

```
for each position in the list
for i in range from zero to len animals
print Beginning of loop
print each element and its position
print The element in position concatenated
with str i concatenated with is
concatenated with animals in position i
```

- You really wanted to see a penguin:

```
[]:
1  wanted_to_see = "penguin"
```

```
wanted to see is assigned penguin
```

- Once home, you tell your friend the animals you saw, specifying which one you really wanted to see:

```
[]:
1  # for each position in the list
2  for i in range(0, len(animals)):
3      # if the current animal is
       what you really wanted to see
4      if animals[i] == wanted_to_see:
5          # print that that's the animal
           you really wanted to see
6          print("I saw a " + animals[i] +
           " and I really wanted to see it!")
7      else:
8          # print what you saw
9          print("I saw a " + animals[i])
```

```
for each position in the list
for i in range from zero to len animals
if the current animal is what you really
wanted to see
if animals in position i equals wanted to
see
print that that's the animal you really
wanted to see
print I saw a concatenated with animals in
position i concatenated with and I really
wanted to see it!
else
print what you saw
print I saw a concatenated with animals in
position i
```

What's happening in this code? Get some hints by completing the following exercise.

✎ True or false?

1. We can include a condition in a for loop using an if/else construct. T F
2. The built-in function len() returns the number of elements in a list. T F
3. The hash symbol # starts a new line of code. T F
4. The == symbol checks whether two variables are different. T F

Computational thinking and syntax

Let's start by running the first cell:

[1]:
```
1  animals = ["giraffe", "penguin", "dolphin"]

2  print(animals)
```
```
['giraffe', 'penguin', 'dolphin']
```
```
animals is assigned giraffe, penguin,
dolphin
print animals
```

We consider a list called `animals` containing three strings: `"giraffe"`, `"penguin"`, and `"dolphin"` (line 1), and we print it (line 2).

Let's run the second cell:

[2]:
```
1  # for each position in the list
2  for i in range(0, len(animals)):
3      print("--- Beginning of loop ---")
4      # print each element and its position
5      print("The element in position " +
       str(i) + " is " + animals[i])
```
```
--- Beginning of loop ---
The element in position 0 is giraffe
--- Beginning of loop ---
The element in position 1 is penguin
--- Beginning of loop ---
The element in position 2 is dolphin
```
```
for each position in the list
for i in range from zero to len animals
print Beginning of loop
print each element and its position
print The element in position concatenated
with str i concatenated with is
concatenated with animals in position i
```

We run the `for` loop three times, and each time we print the lines 3 and 5. Let's dig into the code to understand it better! The header of the `for` loop at line 2 contains two changes from the syntax we saw in the previous chapter. First, we use the **abbreviation `i` for the variable `index`**. Shortening names of frequently used variables is common in coding because it reduces the amount of typing required. Some abbreviations become conventions—like in this case—so, from this point on we will use `i`. Second, instead of an integer, we use `len(animals)` as the stop in the built-in function `range()`. If we used an integer, we would write `3`, because the last element—`"dolphin"`—is in position 2, to which we add 1 for the plus one rule. But what if we added another element to the list? We would have to remember to modify the stop from `3` to `4`. As you can imagine, this practice is very prone to error, as it's easy to forget to update the stop or miscount the last element position. Therefore, we do **not** want to **hard-code** the stop—that is, to **explicitly write its value**. We want to make it **dependent on the variable** we are dealing with so that we do not have to take care of possible variations. To do so, we use **`len()`**, which is a built-in function that **returns the length of a variable**—that is, `3` for the list `animals`. We can use this trick because **the length of a list is always one unit more than the index of the last element**; therefore, it coincides with the stop. From this point on, we will not need to count to find the stop—`len()` will do

it for us!

Let's analyze the body of the `for` loop. At line 3, we print a string stating that we are at the beginning of a loop. It is meant to be **visually different** to make the printouts of each iteration **easy to identify**. Beyond `Beginning of loop`, we could use sentences like `New iteration`, `New loop`, etc. To increase the visibility, we can also use symbols before and/or after the text—such as dashes (`---`), like in this example. Alternatives can be arrows (`-->`), tildes (`~~~`), or any other character on the keyboard. At line 5, we print each element and its position in a sentence composed of four parts concatenated to each other. The first and the third parts—`"The element in position "` and `" is "`—are two hard-coded strings. The second element is the index of the current loop. It's an integer, so we use the built-in function `str()` to convert it into a string. Finally, the last element (`animals[i]`) is a string, containing a list element sliced at a different position `i` at each iteration—that is, `"giraffe"`, `"penguin"`, or `"dolphin"`.

Lines 1 and 4 start with the **hash symbol** (#) and are followed by text. These lines are called **comments**. What are they? Let's give a definition:

Comments are code **descriptions** or **explanations**

Comments are a **fundamental component** of coding. They can contain **descriptions** of the code, **explanations** about why we made a certain coding choice, or any other **information** that is relevant to understand the code they refer to. Comments are in light green in Jupyter Notebook, and, in general, they are above and aligned with the line/s they explain. For example, the comment at line 4 refers to the code at line 5, so it is indented and aligned with line 5. You might wonder why we write comments. For at least two reasons. First reason: **to make code readable** for us and others. When reading old code, we rarely remember why we wrote what we wrote—yes, even if we wrote it ourselves! Similarly, when we read somebody else's code, it is often hard to understand what they did and why, if the code is not well commented. Second reason: **to keep track of what we are doing**. When writing code, we sometimes concentrate on small details and lose the big picture. In these cases, we can end up asking ourselves: why am I writing this again? Using comments to outline code can help us keep track of the steps we have to **implement**—that is, to write. Finally, how do we write useful comments? That's simple: we **use precise language**. Writing `# here is a for loop` does not add any information to code because a loop is clearly visible. It is more meaningful to describe what the `for` loop does and why; for example, `# using a for loop to browse a list and print its elements one by one`. Also, don't take any line of code for granted. It's really so easy to forget why we wrote that line of code that way! In general, remember that **comments are written for human beings**, not for Python. As a matter of fact, Python skips comments when it reads our code. Try to add an hash in front of a line of code yourself: Python is not going to execute it!

Let's run the next cell:

```
[3]: 1 wanted_to_see = "penguin"                    wanted to see is assigned penguin
```

We create a variable called `wanted_to_see` to which we assign the string `"penguin"`.

Let's run the last cell:

```
[4]:
1  # for each position in the list
2  for i in range(0, len(animals)):
3      # if the current animal is
       what you really wanted to see
4      if animals[i] == wanted_to_see:
5          # print that that's the animal
           you really wanted to see
6          print("I saw a " + animals[i] +
           " and I really wanted to see it!")
7      else:
8          # print what you saw
9          print("I saw a " + animals[i])
```

```
I saw a giraffe
I saw a penguin and I really wanted to see it!
I saw a dolphin
```

```
for each position in the list
for i in range from zero to len animals
if the current animal is what you really
wanted to see
if animals in position i equals wanted to
see
print that that's the animal you really
wanted to see
print I saw a concatenated with animals
in position i concatenated with and I
really wanted to see it!
else
print what you saw
print I saw a concatenated with animals
in position i
```

Once more, we use the `for` loop to browse the list elements. But this time, **we apply a condition to each element**. Let's analyze line by line. The header of the `for` loop is the same as the one in cell 2. At line 4, we start an `if/else` construct. It is similar to the one we learned in Chapter 3: it's composed of an `if` condition (line 4), a statement (line 6), an `else` (line 7), and another statement (line 9). However, the condition after the keyword `if` is different. In Chapter 3, we checked if an element was in a list by using the membership operator `in`. In this case, **we check if the values assigned to the two variables** `animals[i]` and `wanted_to_see` **are equal**. To do so, we write (1) the keyword `if`, (2) the first variable, that is, `animals[i]`, (3) the comparison operator `==`, and (4) the second variable, that is, `wanted_to_see`. The comparison operator `==` is pronounced *equals* or *is equal to*. Note that **`==` is very different from `=`**. The symbol `==` is a **comparison operator** and is used in conditions to check if the values assigned to two variables are the same. The symbol `=` is the assignment operator, and it is used to assign a value to a variable.

To make sure that what this code does is clear, let's go through the `for` loop iteration by iteration:

- First loop: at line 2, `i` is `0`. At line 4, we check if `animals` in position `i`—where `i` is `0`, so `animals[0]` is `"giraffe"`—is equal to the value assigned to the variable `wanted_to_see`, which is `"penguin"`. Because `"giraffe"` is not equal to `"penguin"`, we skip the statement under the if at line 6, and we jump directly to the statement under the `else`, which is at line 9. There, we print `"I saw a giraffe"`.
- Second loop: at line 2, `i` is `1`. At line 4, we check again if `animals` in position `i`—where `i` is `1`, so `animals[1]` is `"penguin"`—is equal to the value assigned to the variable `wanted_to_see`. In this case, the values of the two variables `animals[i]` and `wanted_to_see` are equal, so we execute the statement under the `if` condition (line 6), where we print `"I saw a penguin and I really wanted to see it!"`.
- Third loop: at line 2, `i` is `2`. At line 4, we check once more if `animals` in position `i`—where `i` is `2`, thus `animals[2]` is `"dolphin"`—is equal to the value assigned to the variable `wanted_to_see`, which is `"penguin"`. Because `"dolphin"` is not equal to `"penguin"`, we skip the statement at line 6, and we jump directly to the statement under the `else`, which is at line 9. There, we print `"I saw a dolphin"`.

✎ Complete the table

In coding there is a lot of jargon—that is, technical words or expressions that are typically used, but whose meaning is not always clear. Have you familiarized yourself with the jargon introduced so far? Complete the table by writing the meaning of the following expressions:

Expression	Meaning
To run a cell (Chapter 1)	
To write readable code (Chapter 3)	
The function takes one argument (Chapter 5)	
The function returns an integer (Chapter 5)	
To reassign to a variable (Chapter 7)	
The element is hard-coded (Chapter 8)	
To comment code (Chapter 9)	
To hard-code (Chapter 9)	
To implement code (Chapter 9)	

Recap

- In a `for` loop, the variable `index` is commonly abbreviated with `i`.
- The built-in function `len()` returns the length of a variable.
- We can use the `if/else` construct in a `for` loop.
- We can use the comparison operator `==` (*equals* or *is equal to*) in an `if` condition.
- Comments start with the hash symbol `#` and contain descriptions or explanations.

Dealing with TypeError

Type error is common when we try to **concatenate variables of different types**. Let's look at this example, modified from cell 2 in this chapter:

[2]:

```
# for each position in the list
for i in range(0, len(animals)):

    print("--- Beginning of loop ---")
    # print each element and its position
    print("The element in position " +
    i + " is " + animals[i])
```

```
for each position in the list
for i in range from zero to len
animals
print Beginning of loop
print each element and its position
print The element in position
concatenated with i concatenated
with is concatenated with animals in
position i
```

```
--- Beginning of loop ---
```

```
---------------------------------------
TypeError       Traceback (most recent call last)
Cell In[2], line 5
       3 print("-- Beginning of loop --")
       4 # print each element and its position
----> 5 print("The element in position " + i + " is " + animals[i])
TypeError: can only concatenate str (not "int") to str
```

The last line of the error message says `TypeError: can only concatenate str (not "int") to str`. It means that somewhere in our code we are trying to concatenate an integer with one or more strings. Where? The green arrow points to line 5, where there are three concatenations. As mentioned previously in the chapter, the components are `"The element in position"` and `" is "`, which are two hard-coded strings; the list element `animals[i]`—that is, `"giraffe"`, `"penguin"`, or `"dolphin"`—which is a string, too; and the variable `i`, which is an integer between `0` and `2`. So `i` is the issue! Solving the error is very easy: we just transform `i` into a string with the built-in function `str()`, like this: `str(i)`.

Let's look at another example, modified from Chapter 7:

[6]:

```
planets = planets + "Jupyter"

print(planets)
```

```
planets is assigned planets
concatenated with Jupyter

print planets
```

```
---------------------------------------
TypeError       Traceback (most recent call last)
Cell In[6], line 1
----> 1 planets = planets + "Jupyter"
       2 print(planets)
TypeError: can only concatenate list (not "str") to list
```

This time, the last line of the error message says: `TypeError: can only concatenate list (not "str") to list`. We are trying to concatenate a string to a list. Where? The green arrow points to line 1. Around the concatenation operator, there are `planets`—which is a list—and `"Jupyter"`—which is a string! Correcting this error is easy: we simply transform `"Jupyter"` into a list by embedding it in between square brackets, like this: `["Jupyter"]`. When getting a type error, remember to **analyze the type of each variable** located in the line of code where the error occurs. Also, remember that we can only concatenate lists with lists, and strings with strings!

Let's code!

Note: Starting from this chapter, write code comments wherever pertinent.

1. For each of the following scenarios, create code similar to that presented in this chapter:
   a. *Sports.* Write a list of sports you like, and print them out one by one. What is your favorite sport? Create a variable for it. Finally, print all sports one by one, specifying if they are your favorite sports.
   b. *An astronaut's next destination.* You are an astronaut and you write down the list of the planets of the solar system: Mercury, Mars, Venus, Earth, Neptune, Uranus, Saturn, Jupiter. Print the planets one by one. Then, create a variable for your next destination. Finally, print all the planets, specifying if they are your next destination.
2. *Months.* Given the following list:

   ```
   months = ["February", "July", "January", "August", "December", "June"]
   ```

   Print the names of winter months using a `for` loop. Then, print the names of summer months using a `for` loop. Choose a month you like and assign it to a variable. Print all the months one by one, specifying if the current month is your favorite. Finally, what alternative way could you use to check if your favorite month is in the list?
3. *Mary K. Keller.* Given the following list:

   ```
   mary_k_keller = ['a nun', 'She was also', 'in Computer Science.', 'to receive a Ph.D.',
   'American woman', 'the first', 'was', 'Mary K. Keller']
   ```

   Print all the elements in reverse order, first using slicing, and then using a `for` loop. Then, consider the following variable: `name = 'Mary K. Keller'`. Check if this variable is in the list in two ways: first, using the `if/else` construct; and then, using the `if/else` construct in a `for` loop. What are the differences between the two methods?

10. Where are my gloves?

For loop for searching

When combined with lists, a `for` loop is typically used for at least three operations: searching elements, changing elements, and creating new lists, as you will learn in the remaining three chapters of this part. In this chapter, we will start with learning how to use the `for` loop to search elements in a list. Ready? Open Jupyter Notebook 10 and follow along. Cover the code after each task with a piece of paper, and try to guess the answer. Then compare and read the explanation. Let's get started!

- Who doesn't have a messy drawer? Here is ours! It contains some accessories:

```
[1]:  1  accessories = ["belt", "hat", "gloves",
          "sunglasses", "ring"]
      2  print(accessories)
['belt', 'hat', 'gloves', 'sunglasses', 'ring']
```

```
accessories is assigned belt, hat,
gloves, sunglasses, ring
print accessories
```

We start with the list `accessories` composed of 5 strings (line 1), and we print it (line 2).

- Print all accessories one by one, as well as their positions in the list. Use a sentence like: `The element x is in position y`:

```
[2]:  1  # for each position in the list
      2  for i in range(len(accessories)):
      3      # print each element and its position
      4      print("The element " + accessories[i] +
             " is in position" + str(i))
The element belt is in position 0
The element hat is in position 1
The element gloves is in position 2
The element sunglasses is in position 3
The element ring is in position 4
```

```
for each position in the list
for i in range len accessories
print each element and its position
print The element concatenated
with accessories in position i
concatenated with is in position
concatenated with str i
```

We warm up by using a `for` loop to print each list element and its position, as we learned in Chapters 8 and 9. The syntax of the `for` loop is the same as we saw previously, with one last simplification in the header: we **omit the start**. When the start is 0—that is, the beginning of the list—we don't need to write it. Can we also omit the stop when it coincides with the end of the list? Not really: the built-in function `range()` would not know where to stop creating consecutive integers (as you might remember, `range()` creates a list of integers—see Chapter 8). Finally, note that we keep commenting each command to increase code readability.

Now it's time to look for items in the drawer. How do we do it? To search list elements, we have to (1) create a **`for` loop to browse all elements** of a list and (2) **use an `if/else` construct** to check if the current element has the characteristics we want, like we did at cell 4 of Chapter 9. In general, we can search for elements based on various conditions. In the previous chapters, we searched if elements are present in a list (Chapter 3) and for elements equal to a given variable (Chapter 9). In this chapter, we will search for elements of a certain length and in a certain list position. To do that, we will use the **comparison operators**. Ready? Let's go!

1. Print the accessory whose name is **composed of** 6 characters and its position in the list. Use a sentence like: `The element x is in position y and it has n characters`:

[3]:
```
1 # for each position in the list
2 for i in range(len(accessories)):
3     # if the length of the element equals 6
4     if len(accessories[i]) == 6:
5         # print the element, its position,
          and its number of characters
6         print("The element " + accessories[i] +
          " is in position" + str(i) +
          " and it has 6 characters")
```

```
for each position in the list
for i in range len accessories
if the length of the element equals
six
if len accessories in position i
equals six
print the element, its position,
and its number of characters
print The element concatenated
with accessories in position
i concatenated with is in
position concatenated with str i
concatenated with and it has six
characters
```

```
The element gloves is in position 2 and it has 6 characters
```

We create a `for` loop to browse all elements in the list (line 2), and we write an `if/else` construct to evaluate if the current element—that is, `accessories[i]`—is composed of 6 characters (lines 4 and 6). How do we know how many characters a string has? The **number of characters coincides with the length of the string**; therefore, we can use the built-in function `len()`. Thus, in the `if` condition, we compare the length of the current element of the list—`len(animals[i])`—to the number of characters we want—that is, `6`. The comparison operator that we use is `==` (*equals* or *is equal to*), which checks if two values are identical, as you learned in Chapter 9 at cell 4. If the current element satisfies the condition, we print the sentence at line 6, like we do for the element `"gloves"`. What about the other elements? We do not want to do anything, so we simply omit the `else` part of the `if/else` construct. Finally, note the presence of comments on lines 1, 3, and 5 to improve code readability.

2. Print the accessories whose names are composed of **less than** 6 characters:

[4]:
```
1 # for each position in the list
2 for i in range(len(accessories)):
3     # if the length of the element is less
      than 6
4     if len(accessories[i]) < 6:
5         # print the element, its position,
          and its number of characters
6         print("The element " + accessories[i] +
          " is in position" + str(i) +
          " and it has less than 6 characters")
```

```
for each position in the list
for i in range len accessories
if the length of the element is
less than six
if len accessories in position i
less than six
print the element, its position,
and its number of characters
print The element concatenated
with accessories in position
i concatenated with is in
position concatenated with str i
concatenated with and it has less
than 6 characters
```

```
The element belt is in position 0 and it has less than 6 characters
The element hat is in position 1 and it has less than 6 characters
The element ring is in position 4 and it has less than 6 characters
```

The structure of the code is the same as that in example 1. What changes is the comparison operator, which is `<` and is pronounced ***less than*** (line 4). By using this operator, we check if the length of the

current element is less than 6. For the elements composed of less than 6 characters, we print the sentence at line 6—that is, for the strings `"belt"`, `"hat"`, and `"ring"`.

3. Print the accessories whose name is composed of **more than** 6 characters. Also, assign 6 to a variable:

[5]:
```
1 # defining the threshold
2 n_of_characters = 6
3 # for each position in the list
4 for i in range(len(accessories)):
5     # if the length of the element is greater
      than the threshold
6     if len(accessories[i]) > n_of_characters:
7         # print the element, its position,
          and its number of characters
8         print("The element " + accessories[i] +
          " is in position" + str(i) +
          " and it has more than " +
          str(n_of_characters) + " characters")
```

```
defining the threshold
n of characters is assigned six
for each position in the list
for i in range len accessories
if the length of the element is
greater than the threshold
if len accessories in position i
greater than n of characters
print the element, its position,
and its number of characters
print The element concatenated
with accessories in position
i concatenated with is in
position concatenated with str
i concatenated with and it has
more than concatenated with str n
of characters concatenated with
characters
```

```
The element sunglasses is in position 3 and it has more than 6 characters
```

In this example, there are two novelties. The first is straightforward: we use the comparison operator `>`, which is pronounced ***greater than*** (line 6). In this case, only one string has more than 6 characters—that is, `"sunglasses"`—so we print line 8 for that element.

The second novelty is the variable `n_of_characters` (line 2). It is assigned 6—that is, the threshold length above which we want to print list elements. Why do we create `n_of_characters` instead of simply using 6? Because we need it in two lines of code—in the condition (line 6) and in the print (line 8)—and this implies the possibility of errors. What if instead of considering 6 characters, we wanted to consider 4? We would have to modify the number both at lines 6 and 8, and we could forget to change one of the two commands. Instead, by using the variable `n_of_characters`, we change the value just in the assignment (line 2). It is **good practice to assign values to variables** instead of hard-coding them throughout the code. Variable assignments are usually placed **at the beginning of a block of code** so that they are easy to find, especially when the code consists of several commands.

4. Print the accessories whose name is composed of a number of characters **different from** 6:

[6]:
```
1 # defining the threshold
2 n_of_characters = 6
3 # for each position in the list
4 for i in range(len(accessories)):
5     # if the length of the element is not equal
      to the threshold
```

```
defining the threshold
n of characters is assigned six
for each position in the list
for i in range len accessories
if the length of the element is not
equal to the threshold
```

```
6     if len(accessories[i]) != n_of_characters:
7         # print the element, its position,
          and its number of characters
8         print("The element " + accessories[i] +
          " is in position" + str(i) +
          " and it has a number of characters
          different from " +
          str(n_of_characters))
```

```
if len accessories in position i
not equal to n of characters
print the element, its position,
and its number of characters
print The element concatenated
with accessories in position
i concatenated with is in
position concatenated with str
i concatenated with and it has
a number of characters different
from concatenated with str n of
characters
```

```
The element belt is in position 0 and it has a number of characters different from 6
The element hat is in position 1 and it has a number of characters different from 6
The element sunglasses is in position 3 and it has a number of characters different
from 6
The element ring is in position 4 and it has a number of characters different from 6
```

The comparison operator for *different from* is **!=** and is pronounced ***not equal to*** (line 6). The structure of the code is the same as that above: we use the variable `n_of_characters` to avoid hard coding (line 2); we create a `for` loop to browse all list elements (line 4); we create an `if` condition to check what strings have lengths not equal to the threshold (line 6); and, finally, we print a sentence for those elements that satisfy the condition (line 8)—that is, `"belt"`, `"hat"`,`"sunglasses"`, and `"ring"`. Before each command, we write a comment to explain what the command does (lines 1, 3, 5, and 7).

5. Print the accessories whose position is **less than or equal to** 2:

[7]:

```
1  # defining the threshold
2  position = 2
3  # for each position in the list
4  for i in range(len(accessories)):
5      # if the position of the element is less
       than of equal to the threshold
6      if i <= position:
7          # print the element, its position,
           and its position characteristic
8          print("The element " + accessories[i] +
           " is in position" + str(i) +
           ", which is less than or equal to " +
           str(position))
```

```
defining the threshold
position is assigned two
for each position in the list
for i in range len accessories
if the position of the element is
less than or equal to the threshold
if i less than or equal to position
print the element, its position,
and its position characteristic
print The element concatenated
with accessories in position
i concatenated with is in
position concatenated with str i
concatenated with , which is less
than or equal to concatenated with
str position
```

```
The element belt is in position 0, which is less than or equal to 2
The element hat is in position 1, which is less than or equal to 2
The element gloves is in position 2, which is less than or equal to 2
```

In this example, we introduce two novelties again. The first novelty is the comparison operator **<=**, which is pronounced ***less than or equal to*** (line 6). What is the difference between the two comparison operators <= (less than or equal to) and < (less than)? When using **<=**, we **include the threshold**—that is, we consider all the elements whose position is equal to 2 or less. When using **<**, we **exclude the threshold**—that is, we consider only the elements whose position is strictly less than 2.

The second novelty is that we want to **search for elements based on their position**. How do we do it? First, we create a variable called `position` to which we assign the threshold—that is, `2` (line 2). Then, we need to write the comparison. How do we know the position of each element? **In a `for` loop, the position of the current list element is `i`**! Remember the following from the previous chapters?

- In the first loop, `i` is `0`, thus `accessories[i]` is `accessories[0]`, which is `"belt"`.
- In the second loop, `i` is `1`, thus `accessories[i]` is `accessories[1]`, which is `"hat"`.
- In the third loop, `i` is `2`, thus...

Therefore, in the `if` condition, we compare the current element position `i` to the threshold position in the variable `position` (line 6). For all those elements whose position `i` is less than or equal to `position`, we print line 8—that is, for `"belt"`, `"hat"`, and `"gloves"`.

6. Print the accessories whose position is **at least** 2:

```
[8]:
1  # defining the threshold
2  position = 2
3  # for each position in the list
4  for i in range(len(accessories)):
5      # if the position of the element is greater
       than of equal to the threshold
6      if i >= position:
7          # print the element, its position,
           and its position characteristic
8          print("The element " + accessories[i] +
           " is in position" + str(i) +
           ", which is at least " + str(position))
```

```
defining the threshold
position is assigned two
for each position in the list
for i in range len accessories
if the position of the element
is greater than or equal to the
threshold
if i greater than or equal to
position
print the element, its position, and
its position characteristic
print The element concatenated
with accessories in position
i concatenated with is in
position concatenated with str i
concatenated with , which is at
least concatenated with str position
```

```
The element gloves is in position 2, which is at least 2
The element sunglasses is in position 3, which is at least 2
The element ring is in position 4, which is at least 2
```

In this last example, the code structure remains the same, but we use the comparison operator **`>=`**, pronounced ***greater than or equal to*** (line 6). Similarly to before, the difference between `>=` (greater than or equal to) and `>` (greater than) is that when using **`>=`**, we **include the threshold**, whereas when using **`>`**, we **exclude the threshold**. In this case, we print the sentence at line 8 for all the elements whose position is at least—that is, greater than or equal to—`position`, which are `"gloves"`, `"sunglasses"`, and `"ring"` (line 8).

Finally, a trick to remember the spelling of comparison operators composed of two symbols: the symbol **`=` is always in the second position**, as you might have noticed for `!=` (cell 4), `<=` (cell 5), and `>=` (cell 6).

✎ Complete the table

In this chapter, you learned the six comparison operators. Sum up their characteristics in your own words in the table below:

Comparison operator	What it does	Pronunciation
==		
!=		
<		
<=		
>		
>=		

✎ Insert into the right column

Up to now, you have learned several coding elements: data types, built-in functions, keywords, and list methods. Do you remember which is which? Insert the following elements into the right column:

string, else, input(), if, .remove(), print(), .index(), len(), str(), del, list, .append(), range(), for, .insert(), integer, .pop()

Data types	Built-in functions	**Keywords**	List methods

Recap

- We can use a `for` loop combined with an `if/else` construct to search for elements in a list.
- It is good practice to create variables instead of hard-coded values in a block of code to reduce the possibility of errors. Variables are usually located at the beginning of a block of code.
- In Python, there are six comparison operators: `==`, `!=`, `<`, `<=`, `>`, `>=`.

Let's use keyboard shortcuts!

While coding, it can be very convenient to use keyboard shortcuts to minimize typing interruptions. Although it might sound like a bit of an exaggeration, using the mouse can really be distracting at times because it can slow down the typing rhythm and interrupt the flow of thought. Shortcuts, on the other hand, allow us to never leave the keyboard! They are **combinations of keys pressed simultaneously** that can perform various operations. Let's have a look at the most common ones. In the following examples, we will use the keys that are colored in Figure 10.1.

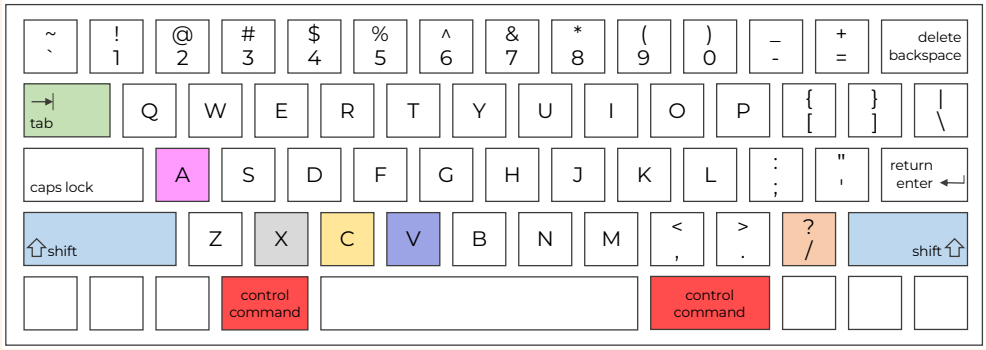


Figure 10.1. Example of a keyboard. The colored keys are commonly used for shortcuts.

In the following shortcut combinations, *Control/Command* means that you will have to press they key *Control* if you are using a Windows operating system, or the key *Command* if you are using a macOS operating system (that is, one of the red keys in Figure 10.1). In addition, the symbol + means that you have to press the listed keys simultaneously. What shortcuts do you know among the following ones?

- *Control/Command* + A (red key + pink key): selects all the lines of code in a cell—the letter A stands for *all*.
- *Control/Command* + X (red key + grey key): cuts selected lines of code.
- *Control/Command* + C (red key + yellow key): copies selected lines of code.
- *Control/Command* + V (red key + purple key): pastes selected lines of code.
- *Control/Command* + / (red key + orange key): adds a # in front of the selected lines of code—that is, it comments them out. If the key combination is re-pressed, the # is removed, and the code is un-commented.
- *Tab* (green key): indents the selected lines of code—that is, it moves the lines four spaces towards right.
- *Shift* + *Tab* (blue key + green key): outdents the selected lines of code—that is, it moves the lines four spaces towards the left.

Note that these shortcuts can be used for several lines of code at once, thus speeding up the writing. Together with learning to type with ten fingers (see the *In more depth* section in Chapter 1), using shortcuts is an efficient way to write code faster and without interruptions!

Let's code!

1. *Seasons.* Given the following list:

   ```
   seasons = ["spring", "summer", "fall", "winter"]
   ```

   Print:

   a. All seasons whose names are composed of at least 5 characters;
   b. All seasons whose names are composed of a number of characters that is equal to or less than 4;
   c. All seasons whose position is less than 2;
   d. All seasons whose position is at least 2.

2. *Word search.* You are working for a magazine and you have just created a new word search game for your readers. Here are the words hidden in the game:

   ```
   words = ["cards", "park", "pets", "football", "golf", "crosswords", "toys", "exercise",
   "hobbies", "riding", "biking", "games", "reading", "movies", "walking", "concerts"]
   ```

   After the grid is completed:

   a. Create a variable called `title` containing the number of words to find, and then print it (e.g., `Word search with 16 words`).
   b. Find words composed of 5 letters. More specifically, print a title, which has to contain the number of letters of this word group, and the words.
   c. Are there words with less than 5 characters? If so, for each word, print a sentence containing the word itself, its position in the list, and its number of characters.
   d. Similarly, are there words with more than 8 characters? If so, for each word, print a sentence containing the word itself, its position in the list, and its number of characters.
   e. What are the words in the second part of the list that have a number of characters different than 7? What's their position? And their number of characters?
   f. Finally, what are the words in the first fourth of the list that are composed of 4 characters? What's their position?

   You can download the real word search game for this exercise from the website!

3. *Spelling competition.* Here are some words of the category musculoskeletal (msk) system that you have to memorize for the next spelling competition:

   ```
   msk_words = ["ankle", "patella", "rib", "femur", "sternocleidomastoid", "tendon", "sternum",
   "abdominal external oblique", "muscle", "scapula", "radius", "bone", "vertebra", "ligament",
   "ulna", "skull", "clavicle"]
   ```

   a. How many words do you have to learn? Compute it and print it.
   b. What is the length of each word? (including spaces if any).
   c. Let's now group words based on their length. Here is a list of short words:
      ```
      short = ["leg"]
      ```
      Add all words with 6 characters or less to the list and print the result. How many words are in the list?
   d. Here is a list of words of intermediate length:
      ```
      intermediate = ["cartilage"]
      ```
      Add all words with 7, 8, and 9 characters. Then print the result. How many words are in the list?
   e. And finally, here is a list of long words:
      ```
      long = ["pectoralis major"]
      ```
      Add all the remaining words and print the result. How many words are in the list?

11. Cleaning the mailing list

For loop to change list elements

Time to learn how to use the `for` loop to change list elements! Open Jupyter Notebook 11 and follow along. Don't forget to pay attention to code pronunciation. Let's go!

- You are responsible for a newsletter, and you have to send an email to the following addresses:

```
[]:
```

```
emails = ["SARAH.BROWN@GMAIL.com",
"Pablo.Hernandez@live.com",
"LI.Min@hotmail.com"]
```

```
emails is assigned
SARAH.BROWN@GMAIL.com,
Pablo.Hernandez@live.com,
LI.Min@hotmail.com
```

- For the sake of consistency, you want all email addresses to be lowercase. So you change them:

```
[]:
```

```
# for each position in the list
for i in range(len(emails)):

    print("-> Loop: " + str(i))

    # print element before the change
    print("Before the change, the element in
    position " + str(i) + " is " + emails[i])


    # change element and reassign
    emails[i] = emails[i].lower()


    # print element after the change
    print("After the change, the element in
    position " + str(i) + " is " + emails[i])


# print the modified list
print("Now the list is: " + str(emails))
```

```
for each position in the list
for i in range len emails

print -> Loop: concatenated with str i

print element before the change
print Before the change, the element
in position concatenated with str i
concatenated with is concatenated with
emails in position i

change element and reassign
emails in position i is assigned
emails in position i dot lower

print element after the change
print After the change, the element
in position concatenated with str i
concatenated with is concatenated with
emails in position i

print the modified list
print Now the list is: concatenated
with str emails
```

What's new in the code above? Get some hints by completing the following exercise.

✎ True or false?

1. To change a list element, we need to reassign after the change. T F
2. The method .lower() is a list method. T F
3. The method .lower() changes a string to uppercase. T F
4. Comments and empty lines make code more readable. T F

Computational thinking and syntax

Let's run the first cell:

[1]:
```
1 emails = ["SARAH.BROWN@GMAIL.com",
  "Pablo.Hernandez@live.com",
  "LI.Min@hotmail.com"]
```

```
emails is assigned
SARAH.BROWN@GMAIL.com,
Pablo.Hernandez@live.com,
LI.Min@hotmail.com
```

We consider a list composed of three strings, each corresponding to an email address (line 1).

Let's run the second cell:

[2]:
```
1  # for each position in the list
2  for i in range(len(emails)):
3
4      print("-> Loop: " + str(i))
5
6      # print element before the change
7      print("Before the change, the element in
       position " + str(i) + " is " + emails[i])
8
9      # change element and reassign
10     emails[i] = emails[i].lower()
11
12     # print element after the change
13     print("After the change, the element in
       position " + str(i) + " is " + emails[i])
14
15 # print the modified list
16 print("Now the list is: " + str(emails))
```

```
for each position in the list
for i in range to len emails

print -> Loop: concatenated with str i

print element before the change
print Before the change, the element
in position concatenated with str i
concatenated with is concatenated with
emails in position i

change element and reassign
emails in position i is assigned
emails in position i dot lower

print element after the change
print After the change, the element
in position concatenated with str i
concatenated with is concatenated with
emails in position i

print the modified list
print Now the list is: concatenated
with str emails
```

```
-> Loop: 0
Before the change, the element in position 0 is: SARAH.BROWN@GMAIL.com
After the change, the element in position 0 is: sarah.brown@gmail.com
-> Loop: 1
Before the change, the element in position 1 is: Pablo.Hernandez@live.com
After the change, the element in position 1 is: pablo.hernandez@live.com
-> Loop: 2
Before the change, the element in position 2 is: LI.Min@hotmail.com
After the change, the element in position 2 is: li.min@hotmail.com
Now the list is: ['sarah.brown@gmail.com', 'pablo.hernandez@live.com',
'li.min@hotmail.com']
```

We use a `for` loop to browse all the elements in the list (line 2). Within the `for` loop, there are four commands. Let's have a look at them one by one.

At line 4, we print a title for each iteration of the `for` loop, as we learned at cell 2 of Chapter 9. The title is composed of a symbol (i.e., `->`) and the number of the current loop—represented by the variable `i`. The symbol makes the title easy to visually identify in the printout, and the loop number favors

checking what happens at each specific iteration.

At line 7, we print the current element (`emails[i]`) before the change, as it is in the list. This will be convenient for comparing the current element before and after the change.

At line 10, we change the current element. How do we do it? We take the current element `emails[i]`, and we change it to lowercase using the **string method** `.lower()`. You might remember that methods are functions for specific data types, they are colored blue in Jupyter Notebook, and their syntax is: (1) variable name, (2) dot, (3) method name, and (4) round brackets, in which there can be an argument (see Chapter 4). How do we know that `.lower()` is a string method and not, for example, a list method? Because `emails[i]` is a string! Python has at least four methods to change character cases:

- **`.lower()`** to change all characters of a string to lowercase;
- **`.upper()`** to change all characters of a string to uppercase;
- **`.capitalize()`** to change the first character of a string to uppercase and all the remaining characters to lowercase;
- **`.title()`** to change the first character of each word in a string to uppercase, and all the remaining characters to lowercase.

Finally, to actually change a list element, we need to **re-assign the modified element**—such as, `emails[i].lower()`—to the element **itself**—that is, `emails[i]`. In other words, we need to **overwrite the current element with its new version**. If we do not do that, the list element will remain unchanged!

At line 13, we print a sentence containing the modified element to check that the change actually occurred. For a double check, we can also compare this sentence with the sentence containing the element before the change, which we printed at line 7.

At line 16, we print the new list. In this case, we need to transform the list `emails` to a string because of the concatenation. Thus, we use the built-in function `str()`, like we do for integers.

Finally, we use two techniques to increase **code readability**. First, we add **comments** before each major command to explain what the code does (lines 1, 6, 9, 12, and 15). Second, we add **empty lines** to visually separate units of thought corresponding to one or more commands, like we would do for paragraphs in a text (lines 3, 5, 8, 11, and 14).

✎ Match the code

Given the following string:

```
greeting = "hElLo, How arE YoU?"
```

Connect each command with the correct output:

1.	`print(greeting.lower())`	a.	`'HELLO, HOW ARE YOU?'`
2.	`print(greeting.upper())`	b.	`'Hello, how are you?'`
3.	`print(greeting.title())`	c.	`'hello, how are you?'`
4.	`print(greeting.capitalize())`	d.	`'Hello, How Are You?'`

Recap

- To change list elements, we always need to reassign the changed element to itself.
- String methods to change cases are: `.lower()`, `.upper()`, `.title()`, and `.capitalize()`.

Changing an element in the correct list using a nested command

Sometimes, we have to change a list element before adding it to an existing list. This can create confusion about where and how to change the list element. Let's consider this example:

- Given the following list:

```
[1]:  1  sports = ["diving", "hiking"]        sports is assigned diving, hiking
```

- Add given this other list:

```
[2]:  1  mountain_sports = ["CLIMBING"]       mountain sports is assigned
                                              CLIMBING
```

- Add the mountain sport in `sports` to the list `mountain_sports`, making sure that the string is uppercase.

We want to take the string `"hiking"` from the list `sports`, transform it into `"HIKING"`, and add it to the list `mountain_sports`. Where do we change the string to uppercase? Let's have a look at these two cases.

Case 1: Changing the element both in the original list and in the new list. Consider the following code:

```
[3]:  1  sports[1] = sports[1].upper()        sports in position one is assigned
                                              sports in position one dot upper
      2  mountain_sports.append(sports[1])    mountain sports dot append sports
                                              in position one
      3  print(sports)                        print sports
      4  print(mountain_sports)               print mountain sports
     ['diving', 'HIKING']
     ['CLIMBING', 'HIKING']
```

In this example, we first change the element in position `1` to uppercase (line 1), and then we append the changed element to the list `mountain_sports` (line 2). When we print the two lists (lines 3 and 4), we see that the element `"HIKING"` is uppercase in both lists. As you can imagine, changing the element in the original list is not the best option because we might need the original list `sports` for further computations. How do we make `"hiking"` uppercase only in `mountain_sports`? Let's have a look at the next example. ↳

Case 2: Changing the element only in the new list. Consider the following code:

```
[3]:
1  current_sport = sports[1].upper()
2  mountain_sports.append(current_sport)
3  print(sports)
4  print(mountain_sports)
['diving', 'hiking']
['CLIMBING', 'HIKING']
```

```
current sport is assigned sports in position one dot upper
mountain sports dot append current sport
print sports
print mountain sports
```

In this example, we assign the transformed element—that is, `'HIKING'`, created with the command `sports[1].upper()`—to a new variable. This new variable is `current_sport` (line 1). Then, we append the variable `current_sport` to the list `mountain_sports` (line 2). When we print both lists (lines 3 and 4), we see that `"HIKING"` is only in the list `mountain_sports`. We can call `current_sport` an **intermediary**, **auxiliary**, or **temporary** variable. Its role is to temporarily store a value that we will use in subsequent code. Although they are very convenient, temporary variables are generally not recommended because they occupy computer memory. Can we avoid using `current_sport`? Yes, let's have a look at this last example:

```
[3]:
1  mountain_sports.append(sports[1].upper())
2  print(sports)
3  print(mountain_sports)
['diving', 'hiking']
['CLIMBING', 'HIKING']
```

```
mountain sports dot append sports in position one dot upper
print sports
print mountain sports
```

In this final example, there is a **nested command**, which is a command containing one or more commands, similar to Russian nesting dolls (line 1). To break down nested commands, we usually start from the inner command and move outwards. In this example, the inner command is `sports[1].upper()`, where we modify the string `'hiking'` to be uppercase. The outer command is `mountain_sports.append()`, where we add the modified element—that is, `'HIKING'`—to the list. As you can see, the inner command is what we assigned to the variable `current_sport` in the previous example. Therefore, we can **avoid an intermediate variable by directly substituting its content in a nested command**. Finally, when we print both lists (lines 2 and 3), we see that we changed `"hiking"` to uppercase only in the list `mountain_sports`.

Nested commands are a convenient way to write **compact code**. How many commands can we nest into each other? Theoretically, as many as we want! In practice, we want to keep nested commands to a minimum for a good balance between code efficiency and code readability.

Let's code!

1. For each of the following scenarios, create code similar to that presented in this chapter:
   a. *Editing an article.* You work at a newspaper, and you have to edit a paper that has plenty of acronyms:
   `acronyms = ["asap", "faq", "fyi", "diy"]`
   All the acronyms are lowercase. Change them to uppercase.

   b. *Name tags.* You are organizing an event, and you have the following list of names:
      `names = ["JOHN", "geetha", "xiao", "LAURA"]`
      You want to print nice name tags, so you capitalize all names.

2. *Colors.* Given the following list:

```
colors = ["yellow", "beige", "green", "red", "ultramarine", "coral", "lavender", "silver",
"cyan", "blue", "black", "magenta", "gold", "pink", "scarlet", "brown"]
```

   a. How many colors are there? Compute it!
   b. Starting from the second element (position 1), change every third word to uppercase.
   c. Starting from the third element (position 2), capitalize every third word.
   d. Add all the colors of the first half of the list `colors` to the following list using a `for` loop, making sure they are lowercase:
      `some_colors = ["white"]`
      How many colors are there in `some_colors` now?
   e. Add all the colors of the second half of the list `colors` to the following list using slicing:
      `more_colors = ["purple"]`
      How many colors are there in `more_colors` now? Change them to uppercase.

3. *Camping.* Given the following list:

```
camping = ["tent", "adventure", "boots", "hiking", "hat", "nature", "path", "lake",
"mountain_sports", "fire", "water bottle", "fishing", "national park", "beach", "compass",
"forest", "trail", "sleeping bag"]
```

   a. How many elements are in there?
   b. Get all the words composed of less than (including) 6 letters and add them to the following list, capitalizing each word:
      `short_camping = ["Trip"]`
   c. Slice every second word of the list `camping` starting from the first word (position 0) and assign them to a new variable called `some_camping_words`.
   d. Capitalize each word of the strings in `some_camping_words` composed of a number of characters other than 4.
   e. In `some_camping_words`, remove the first word (position 0) using a list method.
   f. In `some_camping_words` remove `"path"` using a list method.
   g. Are there more words in `short_camping` or `some_camping_words`? Use an `if/else` construct to print which list has more words, as well as how many words they contain.

12. What a mess at the bookstore!

For loop to create new lists

Let's finally learn how to use a `for` loop to create new lists. Open Jupyter Notebook 12 and follow along. Once more, don't forget to read the code out loud!

- There were many customers in the shop today, and they mixed up the books whose authors' last names start with A and S:

```
[]:  1  authors = ["Alcott", "Saint-Exupéry",
     "Arendt", "Sepulveda", "Shakespeare"]
```

```
authors is assigned Alcott,
Saint-Exupéry, Arendt, Sepulveda,
Shakespeare
```

- So you have to put the books whose authors' last name starts with A on one shelf, and the books whose authors' last name starts with S on another shelf:

```
[]:  1  # initialize the variables as empty lists
     2  shelf_a = []
     3  shelf_s = []
     4
     5  # for each position in the list
     6  for i in range(len(authors)):
     7
     8      # print the current element
     9      print("The current author is: " +
            authors[i])
    10
    11      # get the initial of the current author
    12      author_initial = authors[i][0]
    13      print("The author's initial is: " +
            author_initial)
    14
    15      # if the author's initial is A
    16      if author_initial == "A":
    17          # add the author to the shelf a
    18          shelf_a.append(authors[i])
    19          print("The shelf A now contains: " +
                str(shelf_a) + "\n")
    20
    21      # otherwise (author's initial is not A)
    22      else:
    23          # add the author to the shelf s
    24          shelf_s = shelf_s + [authors[i]]
    25          print("The shelf S now contains: " +
                str(shelf_s) + "\n")
    26
```

```
initialize the variables as empty lists
shelf a is assigned empty list
shelf s is assigned empty list

for each position in the list
for i in range len authors

print the current element
print The current author is: concatenated
with authors in position i

get the initial of the current author
author initial is assigned authors in
position i in position zero
print The author's initial is:
concatenated with author initial

if the author's initial is A
if author initial equals A
add the author to the shelf a
shelf a dot append authors in position i
print The shelf A now contains:
concatenated with str shelf a
concatenated with backslash n

otherwise (author's initial is not A)
else
add the author to the shelf s
shelf s is assigned shelf s concatenated
with authors in position i
print The shelf S now contains:
concatenated with str shelf s
concatenated with backslash n
```

```
27 # print the final shelves
28 print("The authors on the shelf A are: " +
   str(shelf_a))
29 print("The authors on the shelf S are: " +
   str(shelf_s))
```

```
print the final shelves
print The authors on the shelf A are:
concatenated with str shelf a
print The authors on the shelf S are:
concatenated with str shelf s
```

What are the new concepts in this code? Complete the following exercise to get some hints.

✎ True or false?

1. We initialize an empty list by assigning a pair of square brackets. T F
2. We can compose several slicings in one command. T F
3. The method .append() and list concatenation perform two different actions. T F
4. The escape character "\n" creates an empty line after a print. T F

Computational thinking and syntax

Let's run the first cell:

[1]:
```
1 authors = ["Alcott", "Saint-Exupéry",
  "Arendt", "Sepulveda", "Shakespeare"]
```

```
authors is assigned Alcott,
Saint-Exupéry, Arendt, Sepulveda,
Shakespeare
```

The list `authors` is composed of five strings, each of them corresponding to the last name of a book author. The last names start with either A or S.

Let's run the second cell. The code is long, so we break it in pieces. Here are lines 1–3:

[2]:
```
1 # initialize the variables as empty lists
2 shelf_a = []
3 shelf_s = []
```

```
initialize the variables as empty lists
shelf a is assigned empty list
shelf s is assigned empty list
```

We create two new lists, `shelf_a` and `shelf_s`, to which we assign a pair of empty square brackets. Technically, we say that we **initialize two empty lists**—meaning that we create the two lists `shelf_a` and `shelf_s`, but they don't have any content yet. Why do we do that? We will answer this question when we analyze lines 18 and 24. So, let's keep going!

Let's analyze lines 5–9:

```
5 # for each position in the list
6 for i in range(len(authors)):
7
8     # print the current element
9     print("The current author is: " +
      authors[i])
```

```
for each position in the list
for i in range len authors

print the current element
print The current author is: concatenated
with authors in position i
```

We create a `for` loop to browse all the elements in the list `authors` (line 6), and we print a sentence to keep track of the list element sliced at each iteration (line 9).

Let's continue with lines 11–13:

```
11      # get the initial of the current author
12      author_initial = authors[i][0]
13      print("The author's initial is: " +
        author_initial)
```

```
get the initial of the current author
author initial is assigned authors in
position i in position zero
print The author's initial is:
concatenated with author initial
```

At each iteration, we obtain the initial of the current author (line 12), and we print it (line 13). How do we get an author's initial? Let's focus on the right side of the assignment operator—`authors[i][0]`—at line 12. There are two pairs of square brackets, indicating two consecutive slicings. To understand how this works, let's substitute the variables with their corresponding values. In the first loop, `i` is `0`; thus, we get `authors[0][0]`. `authors[0]` is `"Alcott"`, and `"Alcott"[0]` is `"A"`. Similarly, in the second loop, `i` is 1, thus we get `authors[1][0]`. `authors[1]` is `"Saint-Exupéry"`, and `"Saint-Exupéry"[0]` is `"S"`. And so on. With the first pair of square brackets `[i]`, we slice a list and obtain a string, whereas with the second pair of square brackets `[0]`, we slice a string and obtain a single character. In summary, when dealing with **several consecutive slicings, we execute one at the time, starting from the left**. Note that **string slicing works the same way as list slicing**.

Let's have a look at lines 15–25:

```
15      # if the author's initial is A
16      if author_initial == "A":
17          # add the author to the shelf a
18          shelf_a.append(authors[i])
19          print("The shelf A now contains: " +
            str(shelf_a) + "\n")
20
21      # otherwise (author's initial is not A)
22      else:
23          # add the author to the shelf s
24          shelf_s = shelf_s + [authors[i]]
25          print("The shelf S now contains: " +
            str(shelf_s) + "\n")
```

```
if the author's initial is A
if author initial equals A
add the author to the shelf a
shelf a dot append authors in position i
print The shelf A now contains:
concatenated with str shelf a
concatenated with backslash n

otherwise (author's initial is not A)
else
add the author to the shelf s
shelf s is assigned shelf s concatenated
with authors in position i
print The shelf S now contains:
concatenated with str shelf s
concatenated with backslash n
```

We are still in the `for` loop whose header is at line 6, and we find an `if/else` construct. If the author's initial is equal to A (line 16), we append the current author `authors[i]` to the list `shelf_a` (line 18). Then, we print the current status of `shelf_a` (line 19). If the author's initial is not A, then we go to the `else` (line 22), and we concatenate the current author `authors[i]` to the list `shelf_s` (line 24). Note that `authors[i]` is in between square brackets for type compatibility: `authors[i]` is a string, so it must be transformed into a list to be concatenated to the list `shelf_s` (we learned this at cell 6 of Chapter 7). Finally, we print the current status of `shelf_s` (line 25). Let's now look at a few more details.

At lines 18 and 24, we add an element to a list. In the first case, we use the list method `.append()`, whereas in the second case, we use concatenation. The two approaches perform exactly the same operation and can be used interchangeably. However, it is recommended to use one single approach throughout a block of code to improve code readability. Here, we used both approaches to show that they are equally valid.

At the end of the print commands at lines 19 and 25, you'll notice **`"\n"`**. What's that? It's an **escape character** that **creates an empty line after a print**. The backslash `\` tells Python to consider `n` not as a letter of the alphabet, but as a special character meaning *new line*. Printing an empty line is another way to increase code readability in a `for` loop, in addition to printing loop titles (see cell 2 in Chapters 9 and 11). You will see more escape characters in the *In more depth* section of Chapter 27.

Finally, we can answer the question we asked at lines 1–3: why do we need to **initialize** `shelf_a` and `shelf_s` as empty lists? Because it would be **impossible to add new elements to a list that does not exist**!

As a general rule, **when using a `for` loop to create and fill an empty list**, we have to:

1. **Initialize an empty list before the `for` loop**.
2. **Concatenate or append new elements within the `for` loop**.

Let's conclude with lines 27–29:

```
27 # print the final shelves
28 print("The authors on the shelf A are: " +
   str(shelf_a))
29 print("The authors on the shelf S are: " +
   str(shelf_s))
```

```
print the final shelves
print The authors on the shelf A are:
concatenated with str shelf a
print The authors on the shelf S are:
concatenated with str shelf s
```

Above, we print the final versions of the created lists—`shelf_a` (line 28) and `shelf_s` (line 29). In both cases, we transform the list to a string using the built-in function `str()` to concatenate.

Let's look at the printouts:

```
The current author is: Alcott
The author's initial is: A
The shelf A now contains: ['Alcott']

The current author is: Saint-Exupéry
The author's initial is: S
The shelf S now contains: ['Saint-Exupéry']

The current author is: Arendt
The author's initial is: A
The shelf A now contains: ['Alcott', 'Arendt']

The current author is: Sepulveda
The author's initial is: S
The shelf S now contains: ['Saint-Exupéry', 'Sepulveda']

The current author is: Shakespeare
The author's initial is: S
The shelf S now contains: ['Saint-Exupéry', 'Sepulveda', 'Shakespeare']

The authors on the shelf A are: ['Alcott', 'Arendt']
The authors on the shelf S are: ['Saint-Exupéry', 'Sepulveda', 'Shakespeare']
```

Each triplet of lines of code is printed during a `for` loop iteration. Within each triplet, the first line is printed at line 9 (e.g., `The current author is: Alcott`), the second line is printed at line 13 (e.g., `The author's initial is: A`), and the third line is printed at line 19 if the author's initial is A (e.g., `The`

`shelf A now contains: ['Alcott']`), or at line 25 is the author's initial is S (e.g., `The shelf S now contains: ['Saint-Exupéry']`). After each group of 3 lines, there is an empty line because of `"\n"` at the end of the print commands at lines 19 and 25. The last two lines containing the final content of `shelf_a` and `shelf_s` come from the prints at lines 28 and 29. Finally, note that the code contains several comments and empty lines between blocks of code to improve readability.

✎ Match the code

Before starting coding, let's summarize what we learned about for loops in the Chapters 8-12. Given the following list:

```
hot_drinks = ["tea", "coffee", "hot chocolate"]
```

Connect each command with the correct output and the corresponding action:

1.
```
for i in range(len(hot_drinks)):
    print(hot_drinks[i])
```

a.
```
['TEA', 'coffee',
 'hot chocolate']
```

★ create list elements

2.
```
for i in range(len(hot_drinks)):
    if hot_drinks[i][0] == "c":
        print(hot_drinks[i])
```

b.
```
tea
coffee
hot chocolate
```

♣ change list elements

3.
```
for i in range(len(hot_drinks)):
    if len(hot_drinks[i]) == 3:
        hot_drinks[i] = hot_drinks[i].upper()
print(hot_drinks)
```

c.
```
['coffee', 'hot chocolate']
```

■ print list elements one by one

4.
```
long_words = []
for i in range(len(hot_drinks)):
    if len(hot_drinks[i]) >= 6:
        long_words.append(hot_drinks[i])
print(long_words)
```

d.
```
coffee
```

▲ find list elements

Recap

- To create and fill a list in a `for` loop, we have to: (1) initialize an empty list before the `for` loop and (2) fill the list using `.append()` or list concatenation in the `for` loop.
- String slicing works the same way as list slicing.
- In multiple consecutive slicings, we execute one slicing at a time, starting from the left.
- The escape character `"\n"` creates an empty line after a print.

Append or concatenate. Don't assign!

When creating a new list within a `for` loop, a common mistake is to assign a new element to the list instead of appending it or concatenating it. Let's see what this means with the following example. Here is the same list as the one used earlier in this chapter:

[1]:
```
1  authors = ["Alcott", "Saint-Exupéry",
   "Arendt", "Sepulveda", "Shakespeare"]
```
```
authors is assigned Alcott,
Saint-Exupéry, Arendt, Sepulveda,
Shakespeare
```

Let's simplify the code by creating only the list containing author last names starting with `A` and skipping empty lines. To show how an error can occur, at line 10 we assign `authors[i]` to the new list `shelf_a`, instead of appending it (or concatenating it). What happens to `shelf_a` throughout the code?

[2]:
```
1  # initialize the variable
2  shelf_a = []
3  # for each position in the list
4  for i in range(len(authors)):
5      # get the author's initial
6      author_initial = authors[i][0]
7
7      # if the author's initial is A
8      if author_initial == "A":
9          # add the author to the shelf a
10         shelf_a = authors[i]
10
11         print("The shelf A now
           "contains: " + str(shelf_a))
12 # print the final shelf
13 print("The authors on the shelf A are: "
   + str(shelf_a))
```
```
initialize the variable
shelf a is assigned empty list
for each position in the list
for i in range len authors
get the author's initial
author initial is assigned authors
in position i in position zero
if the author's initial is A
if author_initial equals A
add the author to the shelf a
shelf a is assigned authors in
position i
print The shelf A now contains:
concatenated with str shelf a
print the final shelf
print The authors on the shelf A
are: concatenated with str shelf a
```
```
The shelf A now contains: Alcott
The shelf A now contains: Arendt
The authors on the shelf A are: Arendt
```

Let's go through the `for` loop focusing only on the names starting with A:

- When `i` is `0` (line 4), `author_initial` is `"A"`—from `"Alcott"` (line 6); the `if` condition is true (line 8), so we assign `authors[i]`—that is, `"Alcott"`—to `shelf_a` (line 10), and we print: `The shelf A now contains: Alcott` (line 11). With the assignment at line 10, we implicitly transform `shelf_a` from a list—which we initialized at line 2—into a string—because we assign it the string `"Alcott"`.
- When `i` is 2 (line 4), `author_initial` is `"A"`—from `"Arendt"` (line 6); the `if` condition is true (line 8), we assign `authors[i]`—that is, `"Arendt"`—to `shelf_a` (line 10), and we print: `The shelf A now contains: Arendt` (line 11). In this case, in the assignment at line 10, we overwrite the value `"Alcott"`—which we assigned in the previous loop—with the value `"Arendt"`. `shelf_a` remains a string.

At line 13, we print the final version of `shelf_a`, which is a string with value `"Arendt"`.

As this example shows, **assigning a variable to a list** (e.g., `shelf_a = authors[i]`) **changes the type of the list to the variable type** (e.g., `shelf_a` becomes a string). In addition, **the value is**

overwritten at each loop, and the final value is the one assigned in the last loop. Thus, the correct way to add elements to a list is either to append—e.g., `shelf_a.append(authors[i])`—or concatenate—e.g., `shelf_a = shelf_a + [authors[i]]`—as we saw in this chapter.

Let's code!

1. For each of the following scenarios, create code similar to that presented in this chapter.
   a. *Selling electric cars.* You work at a famous car company, and you have to ship new electric cars that have just arrived. Your colleagues plated the cars destined to Spain and to Portugal, but they mixed them up:
   `e_cars = ["PT-754J", "ES-096L", "PT-536G", "ES-543H", "PT-653H"]`
   Separate the two groups of cars according to their destinations.
   b. *Teaching English verbs.* You are an English teacher for foreign students. Some of them have difficulties understanding when a present verb is conjugated in the third person singular (he/she/it), or in other persons (I/you/we/they). So you provide a list of verbs:
   `english_verbs = ["eat", "drink", "eats", "sleep", "drinks", "sleeps"]`
   Help your students separate the verbs between third person and other persons.
2. *Desserts.* Given the following list:
   `desserts = ["meringue", "apple pie", "eclair", "rice pudding", "chocolate", "english pudding", "cake", "icing"]`
   Get all the initials, change them to uppercase, and concatenate them in a new list. Then invert the list. What dessert do you get?
3. *Guess the jobs.* Given the following list:
   `jobs = ["photog", "bal", "mu", "inve", "ambas", "si", "ler", "stig", "rapher", "ci", "ator", "ina", "an", "sador"]`
   Group strings composed of 2, 3, 4, 5, and 6 letters in new lists. What jobs do you get? Make sure that the first letter of each job is uppercase.
4. *Art.* Given the following list:
   `art = ["apor", "refsscu", "atwat", "fetes", "erta", "jtylpt", "aprco", "srap", "ruolo", "texture", "gitp", "puors"]`
   Create new lists for each of the following:
   - If the string length is 4, then get two letters starting from the second (positions 1 and 2).
   - If the string length is 5, then get the third and fourth letters (positions 2 and 3).
   - If the string length is at least 6, then get the last three letters.

   What art words do you get? Make sure all strings are uppercase!

PART 4

NUMBERS AND ALGORITHMS

In this part, you will learn how to perform arithmetic operations, play with random numbers, and implement your first algorithms. Ready? Let’s start!

13. Implementing a calculator

Integers, floats, and arithmetic operations

In the previous chapters, you have developed quite a bit of computational thinking, so now you are ready for numbers, some easy math, and algorithms! There is a general misconception that in order to be good at coding one has to be very good at math. That's not necessarily true, as you will see in the coming chapters!

In this chapter, you will start becoming familiar with numbers in coding by implementing a calculator. To do that, you first need to learn arithmetic operators in Python and how to ask a user for a number. As in previous chapters, try first to solve the task by yourself and then compare your answer with the code below. You will find the code also in Jupyter Notebook 13. Let's start!

1. What are the arithmetic operations in Python?

In Python, there are **7 arithmetic operations**. Let's quickly explore them one by one. Which ones do you already know, and which ones are new?

1. **Addition**:

```
[1]: 1 4 + 3                    four plus three
     7
```

To sum two numbers, we use the arithmetic operator **+**, pronounced *plus*. As you know, the same symbol + is used as a concatenation operator when merging strings or lists; in that case, it is pronounced *concatenated with*.

2. **Subtraction**:

```
[2]: 1 6 - 2                    six minus two
     4
```

To subtract one number from another, we use the arithmetic operator **-**, pronounced *minus*.

3. **Multiplication**:

```
[3]: 1 6 * 5                    six times five
     30
```

To multiply two numbers, we use the multiplication operator *****, which is pronounced *times*. Note that in Python (and in other programming languages), the multiplication symbol differs from the symbol used in paper-and-pencil computations, which can be the cross symbol x or the midline dot ·.

4. **Exponentiation**:

```
[4]: 1 2 ** 3                   two to the power of three
     8
```

To calculate the power of a number, we use the exponentiation operator ******, which is pronounced *to the power of*. The operation `2**3` corresponds to 2^3 in paper-and-pencil.

5. **Division**:

```
[5]: 1 10 / 5                    ten divided by five
     2.0
```

To divide a number by another number, we use a forward slash `/`, and we pronounce it *divided by*. Note that the result of a division is **always a decimal number**.

6. **Floor division**:

```
[6]: 1 7 // 4                    seven floor-divided by four
     1
```

To execute a floor division, we use the operator `//`, composed of two forward slashes and pronounced *floor-divided by*. A floor division is a division where the result is rounded to the **closest lower integer**. In this example, the result of the corresponding division `/` would be `1.75`. Thus, the result of the floor division is the closest lower integer to `1.75`, which is `1`. The word *floor* indicates that we round the result down, that is—using a metaphor–to the floor of a house.

7. **Modulo**:

```
[7]: 1 7 % 4                     seven mod four
     3
```

To calculate a reminder, we use the operator `%`, which is pronounced *mod*. A **reminder** is the number needed to go back to the dividend after a floor division. For example, from cell 6 we know that the result of the floor division `7//4` is `1`. If we multiply `1` (the result) times `4` (the divisor), we get `4` (`4x1=4`). To get to `7` (the dividend), we need `3`, which is the reminder (`4+3=7`). Modulo operations are used quite often in coding, as you will see in the next chapter.

To summarize, Python provides seven arithmetic operators:
- 1 for addition (`+`);
- 1 for subtraction (`-`);
- 2 for the "multiplication family", that is, for multiplication (`*`) and exponentiation (`**`);
- 3 for the "division family", that is, for division (`/`), floor division (`//`), and modulo (`%`).

Note that the division operators can provide whole numbers or decimal numbers as results, independently of the characteristics of dividend and divisor. Discover more nuance by solving the following exercise. Test your answers in Python!

✎ True or false?

1. The result of a division is always a whole number (e.g., without decimals). For example, the result of `11/5` is the whole number `2`. T F
2. The result of `7//2` is `3`, but the result of `-7//2` is `-4`. This is because the floor division rounds to the closest lower integer. T F
3. The result of `7.5 % 3` is `1.5`. Therefore, the result of a modulo operation can be a decimal number. T F

2. How do we ask a user to input a number?

When asking a user to input a number, it's important to be careful about variable types. Let's see what this means!

- Ask a user to input a number, assign it to a variable, and print the variable:

[8]:
```
1 number = input("Insert a number:")
2 print(number)
```
```
Insert a number: 9
9
```
```
number is assigned input Insert a number:
print number
```

We use the built-in function `input()` to ask the user to type a number, and we save the answer in the variable `number` (line 1). Then, we print the variable value (line 2). What type do you expect the variable `number` to be? Let's find out!

- Check the type of the variable `number`:

[9]:
```
1 type(number)
```
```
str
```
```
type number
```

To know the type of a variable, we use the built-in function **`type()`**, which **takes a variable as an input and returns its type**. In the printout, we see that the type of `number` is **`str`**, which is an abbreviation for string. But shouldn't `9` be an integer? Yes! However, `number` is a string because the built-in function **`input()` returns strings**, regardless of what a user types on a keyboard (characters, numbers, or symbols). To transform the value of `number` into an actual number that we can use in calculations, we have to transform its type from string to integer.

- Transform `number` into an integer, print it, and check its type:

[10]:
```
1 number = int(number)
2 print(number)
3 type(number)
```
```
9
int
```
```
number is assigned int number
print number
type number
```

The built-in function **`int()` takes a non-integer variable as an input and returns it as an integer**. Note that to actually transform a variable type, we need to **reassign** the output of the built-in function `int()` to the variable itself (line 1). At line 2, we print `number`, which is still `9`. However, this time `number` is of type **`int`**, as we can see from `type(number)` at line 3. What if we want a decimal number? In that case, we have to transform the variable type into float!

- Transform `number` into a float, print it, and check its type:

[11]:
```
1 number = float(number)
2 print(number)
3 type(number)
```
```
9.0
float
```
```
number is assigned float number
print number
type number
```

The built-in function **`float()` takes a non-decimal variable as an input and returns it as a decimal**. Also in this case, we need to reassign the output of `float()` to the variable itself to actually change the data type (line 1). From the print at line 2, we see that the variable `number` is now `9.0`, that is, a decimal number. And from the command at line 3, we see that `number` is now of type **`float`**. Let's close the circle, and go back to the variable `number` being a string! How would you do that?

- Transform `number` back into a string, print it, and check its type:

```
[12]:
1  number = str(number)
2  print(number)
3  type(number)
```

```
9.0
str
```

```
number is assigned str number
print number
type number
```

To transform a variable into a string, we use the built-in function `str()`, which we learned in Chapter 8. Note that because we transform `number` into a string from a float (and not an integer), the value is now `9.0`—that is, it contains the decimal component. To summarize:

Numerical variables can be of three types:

- **Integers** (whole numbers), used in computations
- **Floats** (decimal numbers), used in computations
- **Strings**, when we need numbers as text—for example, when concatenating to strings

We finally know arithmetic operations in Python and how to ask a number to a user. So we are ready to create a calculator! Where do we start? From the user inputs! Let's define the inputs in the following exercise.

✎ Complete the sentences

Complete the following sentences with the inputs you need from a user to implement a calculator. If you are not sure, think about what you enter when using a calculator:

1. The first input is _____ .
2. The second input is _____ .
3. The third input is _____ .

3. Let's create the calculator!

In the following solution, the first input is the first number in the operation, the second input is the operator, and the third input is the second number in the operation. Let's implement the calculator!

- Ask the user for the first input, which is the first number. What type should it be?

```
[13]:
1  first_number = input("Insert the first
   number:")
2  first_number = float(first_number)
3  type(first_number)
```

```
Insert the first number: 4
float
```

```
first number is assigned input Insert the
first number:
first number is assigned float first number
type first number
```

We ask the user to input the first number using the built-in function `input()`, and we assign the user's choice to the variable `first_number` (line 1). Then, we need to transform the type of `first_number` from a string into a numerical type to perform calculations. Which type do we choose: integer or float? If the user enters a whole number, we need to transform `first_number` into an integer. But what

if the user enters a decimal number? Then, we need to transform `first_number` into a float! So we go for an inclusive solution, that is, we transform `first_number` into a float to comprehend both whole numbers and decimal numbers. Thus, we use the built-in function `float()`, and we reassign to the variable `first_number` (line 2). Finally, we print `first_number`'s type to check that it's correct (line 3).

- Ask the user for the second input, which is the arithmetic operator:

```
[14]: 1  operator = input("Insert an arithmetic
          operator:")
      2  type(operator)
      Insert the arithmetic operator: +
      str
```

```
operator is assigned input Insert an
arithmetic operator:
type operator
```

We ask the user for an arithmetic operator and we save the value in the variable `operator` (line 1). We keep it as a string—you will understand why in a bit—and we print its type to check for correctness (line 2).

- Finally, ask the user for the third and final input, which is the second number. What type should it be?

```
[15]: 1  second_number = float(input("Insert the
          second number:"))
      2  type(second_number)
      Insert the second number: 3
      float
```

```
second number is assigned float input
Insert the second number:
type second number
```

As we did for `first_number`, we ask the user for the second number using the built-in function `input()`. Then, we need to transform the user's choice from string to float using the built-in function `float()`. Instead of using two separate commands like we did at cell 13 (lines 1 and 2), we **nest the two built-in functions one into the other**—we previously saw some nested commands in the *In more depth* section of Chapter 11. We transform the user's choice into a float before assigning it to the variable `second_number` (line 1). Then, we print the `second_number`'s type to make sure that it's a float (line 2).

- Let's write the core of the calculator! How would you do it? Try out some ideas before looking at the implementation below:

```
[16]: 1  if operator == "+":
      2      result = first_number + second_number
      3  elif operator == "-":
      4      result = first_number - second_number
      5  elif operator == "*":
      6      result = first_number * second_number
      7  elif operator == "**":
      8      result = first_number ** second_number
      9  elif operator == "/":
      10     result = first_number / second_number
```

```
if operator equals plus
result is assigned first number plus
second number
elif operator equals minus
result is assigned first number minus
second number
elif operator equals times
result is assigned first number times
second number
elif operator equals to the power of
result is assigned first number to the
power of second number
elif operator equals divided by
result is assigned first number divided by
second number
```

```
11 elif operator == "//":                    elif operator equals floor-divided by
12     result = first_number // second_number result is assigned first number
                                             floor-divided by second number
13 elif operator == "%":                     elif operator equals mod
14     result = first_number % second_number result is assigned first number mod second
                                             number
15 else:                                     else
16     print("You didn't enter an            print You didn't enter an arithmetic
       arithmetic operator")                 operator

17 print(result)                             print result
7.0
```

The operation that our code will execute depends on the arithmetic operator entered by the user; thus, we need to take into account all possibilities. To do that, we create a long list of conditions for the arithmetic operator, with the corresponding calculations in the statements. We start by considering addition (lines 1 and 2). In the `if` condition, we check if the variable `operator` entered in cell 14 is equal to the symbol `+`. Because `operator` is a string, we need to consider the addition operator as a string as well, so we embed it in between quotes (i.e., `"+"`) (line 1). In the subsequent statement, we calculate the sum between the two numerical variables (`first_number` and `second_number`) entered by the user, and we assign the result to the variable `result` (line 2). Then, we consider subtraction (lines 3 and 4). We structure the code as we did above: first, we write a condition where we check that the variable `operator` is equal to the string `"-"` (line 3); then, we execute the diis fference between the two numbers entered by the user, and finally we assign the result to the variable `result` (line 4). As you might have noticed, the condition at line 3 starts with the **keyword `elif`**, which is an abbreviation for `else if`. We use `elif` when we check **several conditions on one single variable**—`operator` in our case. We continue the code with a similar structure for the remaining arithmetic operations (lines 5–14). We conclude the list of conditions with an `else` (line 15), which prints a warning in case the user did not enter a valid arithmetic operator (line 16). Finally, we print the variable `result` to check that our code is correct (line 17). Note that we print `result` at the end of the `if/elif/else` construct instead of after each statement (lines 2, 4, 6, 8, 10, 12, 14) to avoid redundancy. An important remark: when using an `if/elif/else` construct, it is important to **always test code under all conditions**. To do that in our example, re-enter different values for the variables `first_number`, `operator`, and `second_number` to execute the code under each condition and make sure that `result` is what you expected.

- Finally, let's print the result:

```
[17]: 1 print(str(first_number) + " " + operator    print str first number concatenated
        + " " + str(second_number) + " = " +       with space concatenated with operator
        str(result))                               concatenated with space concatenated with
                                                   str second_number concatenated with equals
                                                   concatenated with str result
4.0 + 3.0 = 7.0
```

We print the result, concatenating `first_number`, `operator`, `second_number`, and `result`. Note that we convert the numerical variables into strings for the concatenation.

Finally, let's put together our code to create a real calculator by merging all the code from cells 13–17 into a single cell. This will allow us to run only one cell (instead of multiple cells) when executing the code:

[18]:

```
1  # first input
2  first_number = float(input("Insert the
   first number:"))
3
4  # operator
5  operator = input("Insert an arithmetic
   operator:")
6
7  # second input
8  second_number = float(input("Insert the
   second number:"))
9
10 # computations
11 if operator == "+":
12     result = first_number + second_number
13 elif operator == "-":
14     result = first_number - second_number
15 elif operator == "*":
16     result = first_number * second_number
17 elif operator == "**":
18     result = first_number ** second_number
19 elif operator == "/":
20     result = first_number / second_number
21 elif operator == "//":
22     result = first_number // second_number
23 elif operator == "%":
24     result = first_number % second_number
25 else:
26     print("You didn't enter an
       arithmetic operator")
27
28 # print the result
29 print(str(first_number) + " " + operator
   + " " + str(second_number) + " = " +
   str(result))
```

```
first input
first number is assigned float input
Insert the first number:

operator
operator is assigned input Insert an
arithmetic operator:

second input
second number is assigned float input
Insert the second number:

computations
if operator equals plus
result is assigned first number plus
second number
elif operator equals minus
result is assigned first number minus
second number
elif operator equals times
result is assigned first number times
second number
elif operator equals to the power of
result is assigned first number to the
power of second number
elif operator equals divided by
result is assigned first number divided by
second number
elif operator equals floor-divided by
result is assigned first number
floor-divided by second number
elif operator equals mod
result is assigned first number mod second
number
else
print You didn't enter an arithmetic
operator

print the result
print str first number concatenated
with space concatenated with operator
concatenated with space concatenated with
str second number concatenated with equals
concatenated with str result
```

When we merge code in one cell at the end of an implementation, we usually **edit and clean it** for better **readability**. In this example, we directly transform `first_number` in a float by nesting the built-in function `input()` into the built-in function `float()` (line 2); we delete all the intermediate prints (i.e., we remove line 3 from cell 13, line 2 from cells 14 and 15, and line 17 from cell 16); and we add comments (lines 1, 4, 7, 10, and 28) and lines spaces (lines 3, 6, 9, 27).

✎ Complete the table

In this chapter, you learned the seven arithmetic operators. Sum up their characteristics in your own words in the table below:

Arithmetic operator	Operation	Pronunciation
+		
-		
*		
**		
/		
//		
%		

Recap

- There are seven arithmetic operators in Python: `+`, `-`, `*`, `**`, `/`, `//`, `%`.
- Numbers can be represented by three data types: integers for whole numbers, floats for decimal numbers, and strings when we need to treat them as text.
- To transform a variable into an integer, we use the built-in function `int()`; to transform a variable into a float, we use the built-in function `float()`.
- To check the type of a variable, we use the built-in function `type()`.
- We use the keyword `elif` to check multiple conditions on the same variable. It's important to always execute the code under each condition to check for correctness.

Solving arithmetic expressions

Arithmetic expressions are combinations of arithmetic operations. As we do in paper-and-pencil expressions, we execute operations in a specific order, which is summarized by the acronym BEDMAS. First, we perform operations between brackets, then we compute exponentiation, division, multiplication, addition, and subtraction. Here is an example:

```
[1]:  1  1 + 2 * 3                                  one plus two times three
      7
```

First we execute the multiplication, followed by the addition. Thus, we first calculate `2 * 3`, which is `6`, and then `1 + 6`, which is `7`.

Here is another example:

```
[2]:  1  (1 + 2) * 3                                one plus two times three
      9
```

↳

First, we execute the operation between round brackets `(1 + 2)`, which is 3, and then the multiplication `3 * 3`, which is 9.

In coding, brackets can only be round in expressions. Let's look at this example:

```
[3]: 1 ((1 + 2) + 3 * 4) * 5
     75
```

one plus two plus three times four times five

We solve the expression starting from the inner pair of round brackets. Thus, we compute `(1 + 2)` to obtain 3. Then, we solve the expression in the outer pair of round brackets: `(3 + 3 * 4)`. There, first we compute the multiplication `3 * 4`, which is 12, and then the addition 3+12, which is 15. Finally, we multiply the result by 5 to obtain 75. In paper-and-pencil, this expression would be written as $[(1 + 2) + 3 \times 4] \times 5$.

Let's code!

1. *Math competition.* You are holding a math competition where participants have to choose among three envelopes and solve the arithmetic expressions contained in the chosen envelope:
   - If the participant chooses envelope 1, she will have to solve: $(3 \times 5^2 \div 15) - (5 - 2^2)$.
   - If the participant chooses envelope 2, she will have to solve: $-1 \times [(3 - 4 \times 7) \div 5] - 2^3 \times 24 \div 6$.
   - If the participant chooses envelop 3, she will have to solve: $\frac{(36-3)\times 4}{(15-9)\div 3}$.

   Compute the solutions.
2. *Geometry tutoring.* You are helping your neighbor's kid with some geometry exercises. He has to calculate the area and volume of a cylinder, and you want to test result correctness using Python. Ask the kid for cylinder radius and height. Then calculate area and volume of a cylinder using these formulas: $area = 2\pi r^2 + 2\pi rh$ and $volume = \pi r^2 h$. *Hint*: What is the value of π? Assign it to a variable!
   He also has to calculate surface and area of a cube of edge length $a = 4$. He does not have the right formulas, so you look for them on the internet. Write code to test whether his calculations are correct.
3. *What's the temperature out there?* You are traveling between Europe and North America, and you need to pack the right clothes. Write a temperature converter, knowing that the relation between Celsius and Fahrenheit degrees is $C = 5 \div 9 \times (F - 32)$. Answer these two questions:
   a. The temperature in Miami is 75°F. What is the temperature in Celsius?
   b. The temperature in Lisbon is 17°C. What is the temperature in Fahrenheit?

14. Playing with numbers

Common operations with lists of numbers

Lists of numbers are one of the most used data structures in coding. They follow the same rules as lists of strings—that is, we can use slicing and methods (e.g., `.append()`, `.remove()`, etc.) to manipulate them. In this chapter, we will explore some typical tasks performed with lists of numbers. Open Jupyter Notebook 14 and follow along. As we've done previously, try first to solve the task by yourself: start by defining the expected solution, outline the steps to reach it, and then write the code to solve it. When you are done, compare your implementation with the one proposed here.

1. Changing numbers based on conditions

One of the most common tasks in coding is changing numbers in a list based on some conditions. Let's have a look at this example!

- Given the following list of numbers:

```
[1]:
1  numbers = [12, 3, 15, 7, 18]
```

```
numbers is assigned twelve, three, fifteen, seven,
eighteen
```

We start with a list containing five integers.

- Subtract 1 from the numbers greater than or equal to 10, and add 2 to the numbers that are less than 10:

```
[2]:
1  # for each position in the list
2  for i in range(len(numbers)):
3
4      # if current number >= 10
5      if numbers[i] >= 10:
6          # subtract 1
7          numbers[i] = numbers[i] - 1
8
9      # otherwise
10     else:
11         # add 2
12         numbers[i] = numbers[i] + 2
13
14 # print the final result
15 print(numbers)
```

```
[11, 5, 14, 9, 17]
```

```
for each position in the list
for i in range len numbers

if current number is greater than or equal to ten
if numbers in position i greater than or equal to
ten
subtract one
numbers in position i is assigned numbers in
position i minus one

otherwise
else
add two
numbers in position i is assigned numbers in
position i plus two

print the final result
print numbers
```

We implement a `for` loop to browse all the elements of the list `numbers` (line 2). Then, we use an `if/else` construct to define a condition and compute accordingly. If the current number—that is, `numbers[i]`— is greater than `10` (line 5), we subtract `1`, and we reassign the result to `numbers[i]` (line 7), similarly to what we saw in Chapter 11 (cell 2, line 10). If the current number is not greater than or

equal to `10`, we jump to the `else` (line 10). Then, we add `2` to the current number, and we reassign (line 12). Let's see how this works step by step:

- In the first loop, `i` is `0` (line 2), `numbers` in position `0` is `12`, which is greater than `10` (line 5), so we subtract `1`, obtaining `11`, and we replace `12` with `11` by reassigning (line 7).
- In the second loop, `i` is `1` (line 2), `numbers` in position `1` is `3`, which is not greater than or equal to `10` (line 5), so we jump to the `else` (line 10). We add `2` to `3`, obtaining `5`, and we replace `3` with `5` by reassigning (line 12).
- Etc.

Finally, we print the obtained list to check its correctness (line 15).

2. Separating numbers based on conditions

Another very common task with lists of numbers is to separate numbers into new lists based on given conditions. Let's see an example here!

- Given the following list of numbers:

```
[3]:
1  numbers = [2, 10, 7, 5, 0, 9]
```

```
numbers is assigned two, ten, seven, five, zero,
nine
```

We start with a list containing six integers.

- Separate the numbers into two different lists—one for odd numbers, and one for even numbers:

```
[4]:
1  # initialize the empty lists
2  even = []
3  odd = []
4
5  # for each position in the list
6  for i in range(len(numbers)):
7
8      # if the current number is even
9      if numbers[i] % 2 == 0:
10         # add it to the list even
11         even.append(numbers[i])
12     # otherwise
13     else:
14         # add it to the list odd
15         odd.append(numbers[i])
16
17 # check the final results
18 print(even)
19 print(odd)
```

```
[2,10,0]
[7,5,9]
```

```
initialize empty lists
even is assigned empty list
odd is assigned empty list

for each position in the list
for i in range len numbers

if the current number is even
if numbers in position i mod two equals zero
add it to the list even
even dot append numbers in position i
otherwise
else
add it to the list odd
odd dot append numbers in position i

check the final results
print even
print odd
```

We create two empty lists, one that will contain the even numbers (line 2) and one that will contain the odd numbers (line 3). To fill them up, we need a `for` loop together with the list method `.append()` (or with concatenation), as we learned in Chapter 12. Thus, we create a `for` loop that browses all the list numbers one by one (line 6). Then, we use an `if/else` construct to determine whether each element of the list `numbers` will go to `even` or `odd` (lines 8–15). How do we **decide if a number is even or odd**?

We know that even numbers are divisible by 2—that is, when divided by 2, the remainder is 0—whereas odd numbers are not divisible by 2—the remainder is 1. Thus, checking the value of the reminder is the key to determining whether a number is even or odd! To assess the reminder, we use the **modulo** operation, one of the seven arithmetic operations we learned in the previous chapter. If the remainder of the current list number (e.g., `numbers[i]`) divided by 2 is 0 (line 9), we append `numbers[i]` to the list `even` (line 11). Otherwise (line 13), we append `numbers[i]` to the list `odd` (line 15). Finally, we print the two lists to check the results (lines 18 and 19).

3. Finding the maximum of a list of numbers

A third very common task when dealing with lists of numbers is to find the maximum (or minimum) number in a list. Try to find the maximum of the list below by yourself, drafting and experimenting with code, before looking into the solution.

- Given the following list of numbers:

```
[5]:
1  numbers = [2, -5, 34, 70, 22]
```

```
numbers is assigned two, minus five,
thirty-four, seventy, twenty-two
```

- Find the maximum number in the list:

```
[6]:
1  # initialize the maximum with the
   first element of the list
2  maximum = numbers[0]
3
4  # for each position in the list
   starting from the second
5  for i in range(1, len(numbers)):
6
7  # if the current number is greater
   than the current maximum
8      if numbers[i] > maximum:
9          # assign the number to maximum
10         maximum = numbers[i]
11
12 # print the maximum of the list
13 print(maximum)
70
```

```
initialize the maximum with the first element
of the list
maximum is assigned numbers in position 0

for each position in the list starting from the
second
for i in range one len numbers

if the current number is greater than the
current maximum
if numbers in position i greater than maximum
assign the number to maximum
maximum is assigned numbers in position i

print the maximum of the list
print maximum
```

We create a variable called `maximum` that will contain the maximum number in the list, and we initialize it with the first number in the list, which is `numbers[0]` (line 2). Then, we employ a `for` loop starting from the second position to the last position of the elements in the list (line 5)—we do not start from 0 because it is not very meaningful to compare `numbers[0]` from the loop to `maximum`, which is already set to `numbers[0]`. Then, we check if the current number is greater than the maximum (line 8). If so, we assign the number to the maximum (line 10). If not, we do not need to perform any action; therefore, we do not need an `else` and its statement. Finally, we print the maximum (line 13). In other words, we assign the first number of the list—that is, `2`—to a variable that we call `maximum` (line 1). Then, we compare all the subsequent numbers of the list to the value of `maximum`, and if the list number is greater than `maximum`, we assign the list number to `maximum` (lines 5–10). When we look into each iteration, this is what happens:

- When `i` is `1`, `numbers[1]` is `-5`, which is not greater than `2`, so we don't do anything.
- When `i` is `2`, `numbers[2]` is `34`, which is greater than `2`. Thus, `34` is the new maximum and we assign it to the variable `maximum`.
- When `i` is `3`, `numbers[3]` is `70`, which is greater than `34`. Thus, `70` is the new maximum and we assign it to the variable `maximum`.
- When `i` is `4`, `numbers[4]` is `22`, which is not greater than `70`, so we don't do anything. Since the `for` loop is over, the value of `maximum` is `70`, as we found in the previous iteration.

Finally, why do we initialize the variable `maximum` with the first element of the list and not with a very small number? Suppose that we initialize the variable `maximum` with a small number like `0.000005`. If we try to find the maximum value in the list `[0.000001, 0.000002, 0.000003]` by comparing each element to `maximum`, we won't succeed because all elements in the list are smaller than `0.000005`. As a result, we will incorrectly conclude that the maximum is `0.000005`, while the correct maximum should be `0.000003`. When we look for a maximum, picking a specific number as the initial `maximum` does not allow us to generalize our code. We want to **compare the numbers within the list**.

✎ True or false?

1. To change a number in a list, we need to reassign the new value to the same list position. T F
2. To calculate whether a number is divisible or multiple of another number, we use the arithmetic operation floor division. T F
3. To calculate the maximum of a number in a list, we compare the list numbers with each other. T F

Recap

- When dealing with lists of numbers, some of the basic tasks are:
  - Changing numbers in a list depending on conditions;
  - Separating numbers into new lists based on conditions;
  - Finding the maximum (or minimum) number in a list.

Don't name variables with reserved words!

When naming variables, it's important not to use reserved words, that is, names of built-in functions or keywords. How do we know if a name is a reserved word? And what happens if we used it as a variable name? Consider the following example:

```
[1]:  1  len = 10                    len is assigned ten
      2  print(len)                   print len
     10
```

We create a variable called `len` to which we assign the number ten (line 1). As you can see, the **variable name** is colored **green**, which means it is a **reserved word**—we know that `len()` is a Python built-in function, and that variable names are colored black. When we print the variable (line 2), we do not encounter any issue. However, if we want to calculate the length of a list in

subsequent code, we get an error:

```
[2]:  1  numbers = [1, 2, 3]                    numbers is assigned one, two, three
      2  len(numbers)                            len numbers
```

```
---------------------------------------
TypeError       Traceback (most recent call last)
Cell In[2], line 2
       1 numbers = [1, 2, 3]
----> 2 len(numbers)
TypeError: 'int' object is not callable
```

The error message says: `'int' object is not callable`, which means that we want to use `len` as a function; however, now `len` is an integer! In other words, by naming the variable `len` (cell 1, line 1), we overwrote the function `len` with an integer, and we cannot use it as a function anymore. To solve this issue, we have to restart the kernel, that is, we need to erase all variables and start from scratch (see the *In more depth* section in Chapter 7).

Let's code!

1. *Finding the minimum in a list of numbers.* Given the following list of numbers:

   `numbers = [78, -900, 356, -103, 0, -78]`

   find the minimum number in the list.
2. *Grouping numbers by position.* Given the following list of numbers:

   `numbers = [4, 25, 7, -8, 59, 63, -10, 74]`

   separate the numbers in odd positions from the numbers in even positions using a `for` loop.
3. *Number multiples.* Given the following list of numbers:

   `numbers = [20, 24, 69, 15, 100, 16, 40, 80, 33, 57, 2, 200]`

   create a list for the numbers that are multiples of 10, a list for the numbers that are multiples of 3, and a list for the remaining numbers. Finally, delete the list `numbers`.
4. *Longest and shortest string.* Given the following list of strings:

   `dogs = ["labrador", "chihuahua", "basset hound", "bernese shepherd", "poodle", "cocker spaniel"]`

   find the longest and the shortest strings. Print the two strings and their lengths.
5. *Summing numbers in a list.* Given the following list of numbers:

   `numbers = [3, 5, 2]`

   calculate the sum.
6. *Fibonacci sequence.* The Fibonacci sequence is a sequence of numbers where the current number is the sum of the two previous numbers. Write code that asks the user for a number `n` and prints the Fibonacci sequence of `n`.

   Hint: Start the sequence as `[1,1]`

   Example:

   - User input: `10`
   - Output: `[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]`.

15. Fortune cookies

The Python module `random`

Let's continue our discovery of numbers in Python by learning how to generate random numbers. Randomness is quite useful in coding, for example to create games! Read the following example and try to understand it. You can find the code in Notebook 15. Let's start!

- You are at a Chinese restaurant, and at the end of the meal, you get a fortune cookie. There are only three fortune cookies left. Each of them contains a message:

```
[]:
1  fortune_cookies = ["The man on the top of the
   mountain did not fall there", "If winter comes,
   can spring be far behind?", "Land is always on
   the mind of a flying bird"]
```

```
fortune cookies is assigned The
man on the top of the mountain did
not fall there, If winter comes,
can spring be far behind?, Land
is always on the mind of a flying
bird
```

- Which fortune cookie will you get? Let the computer decide! To do so, the computer needs a Python module called `random`:

```
[]:
1  import random
```

```
import random
```

- Here is your message when the computer picks an index:

```
[]:
1  # pick a message index
2  message_index =
          random.randint(len(fortune_cookies)-1)
3  print(message_index)
4
5  # get the message
6  message = fortune_cookies[message_index]
7  print(message)
```

```
pick a message index
message index is assigned random
dot randint len fortune cookies
minus one
print message index

get the message
message is assigned fortune
cookies at message index
print message
```

- And here is your message when the computer directly picks an element:

```
[]:
1  # pick a message
2  message = random.choice(fortune_cookies)
3  print(message)
```

```
pick a message
message is assigned random dot
choice fortune cookies
print message
```

✎ True or false?

1. `import` is a function. T F
2. `random` is a Python module. T F
3. `.randint()` and `.choice()` are functions of the package `random`. T F
4. The arguments of the functions `.randint()` and `.choice()` are of type string. T F

Computational thinking and syntax

Let's begin by running the first cell:

```
[1]:
1  fortune_cookies = ["The man on the top of the
   mountain did not fall there", "If winter comes,
   can spring be far behind?", "Land is always on
   the mind of a flying bird"]
```

```
fortune cookies is assigned The
man on the top of the mountain did
not fall there, If winter comes,
can spring be far behind?, Land
is always on the mind of a flying
bird
```

The variable `fortune_cookies` is a list containing 3 strings (line 1).

Let's continue with the second cell:

```
[2]:
1  import random
```

```
import random
```

We use the keyword **`import`** to import the module **`random`** (line 2). What does this mean? As you know, Python contains basic built-in functions, such as `print()`, `input()`, `len()`, `range()`, etc. However, when coding, we also need tools for recurring tasks that are more complex, such as generating random numbers, browsing directories, or performing computations. For these tasks, Python contains additional units called **modules**. We will talk about modules in greater detail in Chapter 32. For now, let's keep in mind this definition:

A **module** is a **unit** containing **functions** for a **specific task**

Because in Python there are plenty of modules—which could slow down our computer if imported all at once—we usually import only the module (or modules) that we are planning to use. To import a module, we simply use the **keyword `import`** followed by the **module name**.

Let's now run cell number 3:

```
[3]:
1  # pick a message index
2  message_index =
          random.randint(len(fortune_cookies)-1)

3  print(message_index)
4
5  # get the message
6  message = fortune_cookies[message_index]

7  print(message)
```

```
pick a message index
message index is assigned random
dot randint len fortune cookies
minus one
print message index

get the message
message is assigned fortune
cookies at message index
print message
```

```
2
Land is always on the mind of a flying bird
```

The module `random` contains several functions, and in this cell we use `.randint()` (line 2). As you can see, the **syntax to call a module function** is as follows: (1) **module name**, (2) **dot**, (3) **function name**, and (4) **function inputs in between round brackets**. The function **`.randint()`** takes two integers as inputs—which we can we call `a` and `b` (`.randint(a,b)`)—and returns a random integer between `a` and `b` included—that is, `a` and `b` can be possible values. In our example, we want to pick a random number representing the index (or position) of an element in the list `fortune_cookies`. Thus, we could write `.randint(0,2)`. But what if we added or removed some strings to or from the list? We would have to manually change the endpoint `b`, and this could be prone to error! Similarly to what

we do for the `stop` in a `for` loop, we **parametrize** `b`, that is, we write `b` as a function of the length of a list. Thus, we type `len(fortune_cookies)`, from which we subtract 1 because list indexes start from zero (i.e., `len(fortune_cookies)` is 3, but the index of the last element is 2). After creating the random number, we assign it to `message_index` (line 2), and we print it (line 3). Finally, we slice the list `fortune_cookies` in position `message_index` to extract a string, assign it to the variable `message` (line 6), and print to the screen (line 7). One last note: try to run the cell several times. What happens? Every time `.randint()` returns a different number (0, 1, or 2), and thus we get a different fortune cookie message!

Let's have a look at the last cell:

```
[4]:
1  # pick a message
2  message = random.choice(fortune_cookies)

3  print(message)
```

```
The man on the top of the mountain did not fall there
```

```
pick a message
message is assigned random dot
choice fortune cookies
print message
```

In this case, we use another function from the module `random` called **`.choice()`**, which takes a list as an input and returns a randomly selected element of the list (line 2). Finally, we print the message (line 3).

What is the **difference between `.choice()` and `.randint()`**? When using `.choice()`, we do not know the **position** of the element the computer randomly selects, whereas when using `.randint()`, we know where the element is in the list.

Match the sentence halves

1. In `range(start, stop, step)`
2. In `.randint(a,b)`
3. The function `.randint(a,b)`
4. The function `.choice(list)`
5. The syntax to use a string or list method is
6. The syntax to use a function from a module is

a. module name, dot, function name().
b. returns a random element from a list.
c. stop is excluded.
d. variable name, dot, method name().
e. the endpoint b is included.
f. returns a random integer between a and b (included).

Recap

- A module is a unit containing functions for a specific task.
- To import a module, we use the keyword `import`. Imports are usually written at the beginning of code, and only once.
- When calling a module function, we use the following syntax: `module_name.function_name()`.
- `random` is a module to generate random numbers. It contains several functions, including:
  - `.randint(a,b)`: returns a random integer between the endpoints `a` and `b` (included);
  - `.choice(list_name)`: returns an element of a list.

What if I don't use the index in a for loop?

As we know from the previous chapters, in a `for` loop, the variable `i` changes its value from the start to the stop (minus 1!) of the interval created by the function `range()`. Within the loop, we use `i` to either print the current loop number (e.g., `print("This is loop number " + str(i))`) or to automatically slice list elements (e.g., `friends[i]`). However, in some cases, we do not need `i`. Let's look at an example:

[1]:
```
1  import random
2
3  # repeat the commands 3 times (index not
   needed)
4  for _ in range(3):
5      # create a random number between 10
       and 20
6      random_number = random.randint(10,20)
7
8      # print the number
9      print("The random number is" +
       random_number)
```

```
The random numbers is: 14
The random numbers is: 17
The random numbers is: 12
```

```
import random

repeat the commands 3 times
(index not needed)
for underscore in range three
create a random number between
ten and twenty
random number is assigned random
dot randint ten twenty
print the number
print The random number is
concatenated with random number
```

After importing the module `random` (line 1), we use a `for` loop to generate and print three random numbers (lines 3–8). As you can see, we use the `for` loop to repeat commands that do not contain `i`. In this case, it is a Python style convention to substitute `i` with an underscore (i.e., _) in the header of the `for` loop (line 4) to signal that we do not need an index in the loop. Using `i` in the loop header would not be an error, but it would decrease code readability.

Let's code!

1. For each of the following scenarios, create code similar to that presented in this chapter:
   a. *Tossing a coin.* What are the two possibilities when tossing a coin? Write them in a list. Then, toss the coin, once using `.randint()` and once using `.choice()`. What do you get?
   b. *Rolling a die.* What are the possibilities when rolling a die? Write them in a list. Then, roll the die, once using `.randint()` and once using `.choice()`. What numbers do you get? Finally, choose one method and roll the die three times. What numbers do you get?
2. *Ten random numbers.* Create a list of 10 random numbers between 0 and 100 using a `for` loop.
3. *Unique random numbers multiple of a number.* Create a list of 100 random numbers between 5 and 60. Divide them into two lists depending on whether they are a multiple of 4 or not. Then, create another list called `unique`, where you add unique multiples of 4 from the previous list. This means that, for example, if 42 is present more than once, it will appear only once in `unique`. If the number is already present in `unique`, print a sentence like: `The number x is already in unique`. How many unique multiples of 4 could you generate randomly?
4. *Playing with prime numbers.* Create a list of 150 random numbers between 50 and 100, and divide them into lists depending on whether they are multiple of the prime numbers 2, 3, 5, or 7 (a number can be added to more than one lists if it is multiple of several prime numbers). Then, sum up all the elements for each list separately (do not use built-in functions you might find online!). Is each sum a multiple of the original prime number? That is, is the sum of all the multiples of 3 a multiple of 3 itself?

16. Rock paper scissors

Introduction to algorithms

Everybody knows the game rock paper scissors! Kids in every corner of the world play this game, which originated at least 2,000 years ago in China[1]. In this chapter, we will learn how to implement rock paper scissors in Python. How would you do it? Write your ideas in the next exercise and try to write your own implementation. Then, have a look at the computational solution below, implemented also in Notebook 16.

✎ Complete the sentences

Think about three steps you need to implement rock paper scissors and write them below. Consider that you will play against the computer: it will pick either paper, rock, or scissors, and you will do the same. Who wins?

1. _____ .
2. _____ .
3. _____ .

In the following solution, the three steps are: (1) the computer picks randomly among paper, rock, and scissors; (2) the player chooses among paper, rock, and scissors; and (3) the computer's pick and the player's choice are compared to determine who wins. Let's implement the game!

1. Computer pick

In the first step, the computer picks among paper, rock, and scissors. How? Let's have a look at the code below.

- Make the computer pick `paper`, `rock`, or `scissors`:

[1]:

```
1 import random
2
3 # list of game possibilities
4 possibilities = ["rock", "paper", "scissors"]
5
6 # computer random pick
7 computer_pick = random.choice(possibilities)
8
9 print(computer_pick)
```

```
rock
```

```
import random

list of game possibilities
possibilities is assigned rock, paper,
scissors

computer random pick
computer pick is assigned random dot
choice possibilities
print computer pick
```

We import the package `random`, which we learned in the previous chapter (line 1). Then, we create a list containing the possible choices—that is, the three strings `"rock"`, `"paper"`, and `"scissors"` (line 4). We use the function `.choice()` from the package `random` to randomly pick an element from the list

[1] https://en.wikipedia.org/wiki/Rock_paper_scissors

possibilities. Finally, we save the pick in the variable computer_pick (line 7) and we print it (line 8). In this case, computer_pick is rock.

2. Player choice

In the second step, it's the player's turn to choose among rock, paper, and scissors. Let's have a look below.

- Make the player choose among paper, rock, or scissors:

[2]:
```
1 # asking the player to make their choice
2 player_choice = input("Rock, paper, or
  scissors?")
3 print(player_choice)
```
```
Rock, paper, or scissors? rock
rock
```
```
asking the player to make their choice
player choice is assigned input Rock,
paper, or scissors?
print player choice
```

We use the built-in function input() to ask the player to choose among rock, paper, or scissors, and we save the choice in the variable player_choice (line 2). Then, we print it as a check (line 3). In our example, the player chooses rock.

3. Determine who wins

It's time to determine who wins! How do we do it? The computer has three possible picks, and so does the player. Thus, there are nine possible scenarios. How do we code them without forgetting any? One option is to define three situations where the computer's pick is fixed and the player's choice varies. Let's see the implementation!

- If the computer picks rock:

[3]:
```
1 if computer_pick == "rock":
2
3     # compare to the player's choice
4     if player_choice == "rock":
5         print("Tie!")
6     elif player_choice == "paper":
7         print("You win!")
8     else:
9         print("The computer wins!")
```
```
Tie!
```
```
if computer pick equals rock

compare to the player's choice
if player choice equals rock
print Tie!
elif player choice equals paper
print You win!
else
print The computer wins!
```

We start with an if condition to check if the computer pick equals "rock" (line 1). Then we evaluate the player's choice. If the player's choice equals "rock" (line 4), then we print that it's a tie (line 5). If the player's choice equals "paper" (line 6), then we print that the player wins (line 7). Finally, in the remaining case represented by else—the player's choice is "scissors"—(line 8), we print that the computer wins (line 9). The code is very simple: an if condition containing an if/elif/else construct with prints in the statements. As you can see, we **print a message directly to the player**, not to the coder. You might remember that when we code, we alternate two hats: the coder hat and the player hat (see Chapter 1). If we print "The player wins" (line 7), we tell the coder that the code works. But if we print You win!, we talk to the player, who is the person we are coding for! Think about when you

play a computer game: what kind of messages do you get?

In an `if/elif/else` construct, it is important to **test all conditions**. We want to make sure that all statements execute correctly, as we mentioned when we implemented a calculator (Chapter 13). What does testing mean exactly?

Testing means to **evaluating and verifying** that the **code does what it is supposed** to do

How do we test the code in this example? We rerun cell 2—where we ask the player to choose among `rock`, `paper`, and `scissors`—two times: once entering `paper` and once entering `scissors`. After each run, we rerun cell 3 to check that the corresponding printout is correct. It is important to enter the strings in the same order as they appear in the conditions: first `rock`, then `paper`, and finally `scissors`. **Keeping the same order helps us make sure that we test all conditions**, without skipping any.

Sometimes testing is confused with **debugging**, but they are two very different concepts. You might have heard the word **debugging** many times. What is its exact meaning?

Debugging means **identifying** and **removing errors** from code

Debugging is a bit of a detective job. When we get error messages, or we do not obtain the result that we expect, we need to understand **where** the error is so that we can fix it. A very common way to debug is to **print** variables after every line of code, to check the value they are assigned. When the variable value is not the expected one, that's where the error happens! Once we have found the error, we can fix it, and then we can keep coding. To understand further why we use the word debugging, read the *In more depth* section at the end of this chapter.

Let's continue implementing rock paper scissors, looking at the second computer pick possibility.

- If the computer picks `paper`:

[4]:
```
1  if computer_pick == "paper":
2
3      # compare to the player's choice
4      if player_choice == "paper":
5          print("Tie!")
6      elif player_choice == "scissors":
7          print("You win!")
8      else:
9          print("The computer wins!")
```

```
if computer pick equals paper

compare to the player's choice
if player choice equals paper
print Tie!
elif player choice equals scissors
print You win!
else
print The computer wins!
```

The structure of the code is the same as in the previous cell: an `if` condition (line 1) containing an `if/elif/else` construct (lines 4–9). What changes are the terms of comparison—that is, the strings—in the conditions: we check if the computer picks `"paper"`, and we change the conditions for the player according to the printed messages.

When we write code with a repetitive structure—like in our example—it is crucial to use parallelism. Do you know what it is?

Parallelism means maintaining a **corresponding structure**
for subsequent lines or blocks of code

In our example, we can either keep the **conditions in the same order**—e.g., the first term of comparison is always `"rock"`, the second is always `"paper"`, and the third is always `"scissors"`—or we can keep the **statements in the same order**—that is, the first message is always `"Tie!"` (line 5 in both cells 3 and 4), the second is always `"You win!"` (line 7 in both cells), and the third is always `"The computer wins!"` (line 9 in both cells). Parallelism helps us remember to **list all conditions** in every construct, and it improves code **readability**.

Once more, let's not forget to **test all conditions**. We first have to make sure that the computer pick is `paper`. Since we have only three options, a simple way is to rerun cell 1 until we get what we need—that is, `"paper"`. Then, we re-run cells 2 and 4 three times, each time entering the player choice and testing the corresponding print, **in the same order as** in the `if/elif/else` construct. In other words, first we enter `"paper"` at cell 2, and run cell 4 to test lines 4–5. Then, we enter `"scissors"` at cell 2, and run cell 4 testing lines 6–7. And finally, we enter `"scissors"` at cell 2, and run cell 4 to test lines 8–9.

Let's finally look into the third scenario.

- If the computer picks `scissors`:

```
[5]:
1  if computer_pick == "scissors":                  if computer pick equals scissors
2
3      # compare to the player's choice             compare to the player's choice
4      if player_choice == "scissors":             if player choice equals scissors
5          print("Tie!")                           print Tie!
6      elif player_choice == "rock":               elif player choice equals rock
7          print("You win!")                       print You win!
8      else:                                       else
9          print("The computer wins!")             print The computer wins!
```

Also in this last case, the code structure is similar: an `if` condition (line 1) nesting an `if/elif/else` construct (lines 4–9). We check if the computer picked `"scissors"` and if the player chose `"scissors"` (line 4), `"rock"` (line 6), or `"paper"` (the `else` in line 8). As in cell 4, we construct the conditions so that the print statements are **parallel** to those in cell 3. Finally, once more, we want to make sure we **test the code**. Thus, first we re-run cell 1, making sure that `computer_pick` is `"scissors"`. Then, we re-run cells 2 and 5, subsequently entering and testing for `"scissors"`, `"rock"`, and `"paper"`.

Note that we considered a **well-behaved player**, that is, a player that enters `rock`, `paper`, or `scissors` correctly, without any misspelling. We will assume that we are dealing with well-behaved players in all coming chapters to focus on computational thinking and coding syntax. We will learn to check for input correctness in Chapter 30.

At this point, the code is completed! As coders, we have taken care of the various parts of the code, writing and testing them. Now it's time to put all the code together for the player!

Merging the code

- Let's merge the code:

[6]:

```
import random

# list of game possibilities
possibilities = ["rock", "paper", "scissors"]

# computer random pick
computer_pick = random.choice(possibilities)


# asking the player to make their choice
player_choice = input("Rock, paper, or scissors?")

# determine who wins
# if the computer picks rock
if computer_pick == "rock":
    # compare to the player's choice
    if player_choice == "rock":
        print("Tie!")
    elif player_choice == "paper":
        print("You win!")
    else:
        print("The computer wins!")

# if the computer picks paper
if computer_pick == "paper":
    # compare to the player's choice
    if player_choice == "paper":
        print("Tie!")
    elif player_choice == "scissors":
        print("You win!")
    else:
        print("The computer wins!")

# if the computer picks scissors
if computer_pick == "scissors":
    # compare to the player's choice
    if player_choice == "scissors":
        print("Tie!")
    elif player_choice == "rock":
        print("You win!")
    else:
        print("The computer wins!")
```

```
import random

list of game possibilities
possibilities is assigned rock, paper,
scissors
computer random pick
computer pick is assigned random dot
choice possibilities

asking the player to make their choice
player choice is assigned input Rock,
paper, or scissors?

determine who wins
if the computer picks rock
if computer pick equals rock
compare to the player's choice
if player choice equals rock
print Tie!
elif player choice equals paper
print You win!
else
print The computer wins!

if the computer picks paper
if computer pick equals paper
compare to the player's choice
if player choice equals paper
print Tie!
elif player choice equals scissors
print You win!
else
print The computer wins!

if the computer picks scissors
if computer pick equals scissors
compare to the player's choice
if player choice equals scissors
print Tie!
elif player choice equals rock
print You win!
else
print The computer wins!
```

```
Rock, paper, or scissors? rock
You win!
```

When merging code, we usually do some **editing to improve code use and readability**. In this case, we erased the print of `computer_pick` (which was in cell 1, line 8) because we do not want the player to

know the computer choice in advance. Similarly, we delete the print of `player_choice` (which was in cell 2, line 3), as the player already sees their choice from the entry at line 9. Other editing might include improving comments, making variable names more meaningful, restructuring parts of the code, etc.

Let's now zoom out and observe the procedure we use to implement the game. We first defined three steps—see the exercise *Complete the sentences*. Then, we implemented each step separately—see Sections 1. *Computer pick*, 2. *Player choice*, and 3. *Determine who wins*. Finally, we merged all the code together and edited it, as shown in the current section. This way of approaching a task is called divide and conquer.

Divide and conquer means **dividing** a project into **sub-projects**, **solving** the sub-projects, and **combining the solutions** of the sub-projects to obtain the solution of the whole project

In other words, there are three steps to solve a computational task:

1. Break the project into sub-projects
2. Solve the sub-projects separately
3. Merge the solutions of the sub-projects to obtain the solution of the whole project

Last but not least, let's talk about algorithms! You have surely heard this word many times. What is an algorithm?

An **algorithm** is a **sequence of rigorous steps** to **execute and complete a task**

Algorithms are just procedures to solve tasks, problems, or assignments. They do not have to be complicated. They can actually be pretty simple. There are plenty of algorithms in everyday life! Think about the sequence of steps you make to brush your teeth: taking the toothpaste tube, opening and squeezing it, placing toothpaste on the toothbrush, etc. This is an algorithm! Or think about cooking recipes, especially printed recipes. At the top, there is a list of ingredients (e.g., 2 carrots, 3 onions), which are the variables (e.g., `carrots = 2`, `onions = 3`). Then, there is the execution of the recipe, that is, the steps to process the ingredients into the final dish. In programming, many algorithms have been developed in the past few decades. The most famous algorithms were designed to sort lists, find prime numbers, find elements in a list, etc. We will not look into them in this book, but you can find plenty of examples and explanations in more advanced books and on the internet.

✎ Complete the table

In this chapter, you learned several more important concepts in coding. Write their definitions in your own words:

Concept	Definition
Testing	
Debugging	
Parallelism	
Divide and conquer	
Algorithm	

Recap

- An algorithms is a sequence of steps to execute a task.
- When writing an algorithm (and code in general), we largely use testing, debugging, parallelism, and divide and conquer.

Why do we say Debugging, Divide and conquer, and Algorithm?

Do you know why we say debugging, divide and conquer, and algorithm? The term **debugging** originated in 1947, when a moth was found in a relay of the Mark II computer at Harvard University, causing the computer to malfunction. The moth was then taped to a log sheet, with the annotation *Relay 70 Panel F (moth) in relay. First actual case of a bug being found* (see Figure 16.1). Although the word debugging is not mentioned in the annotation, it became popular thanks to Grace Hopper, who worked on the same computer[a,b]. **Divide and conquer** is attributed to Philip II of Macedon, and it was reused by the Roman ruler Julius Caesar, the French emperor Napoleon, and many more[c]. It refers to a military strategy where the invaders divide the enemy forces to defeat them more easily and conquer them as a whole. Finally, the term **algorithm** derives from al-Khwarizmi, the last name of Muhammad ibn Musa al-Khwarizmi, a 9th-century Persian mathematician and astronomer whose books were widely read in Europe in the late Middle Ages. He wrote a book on the Hindu–Arabic numeral system, which was translated into Latin in the 12th century. The latin manuscript starts with the phrase Dixit Algorizmi ("Thus spoke Al-Khwarizmi"), where "Algorizmi" was the translator's Latinization of Al-Khwarizmi's last name[d].

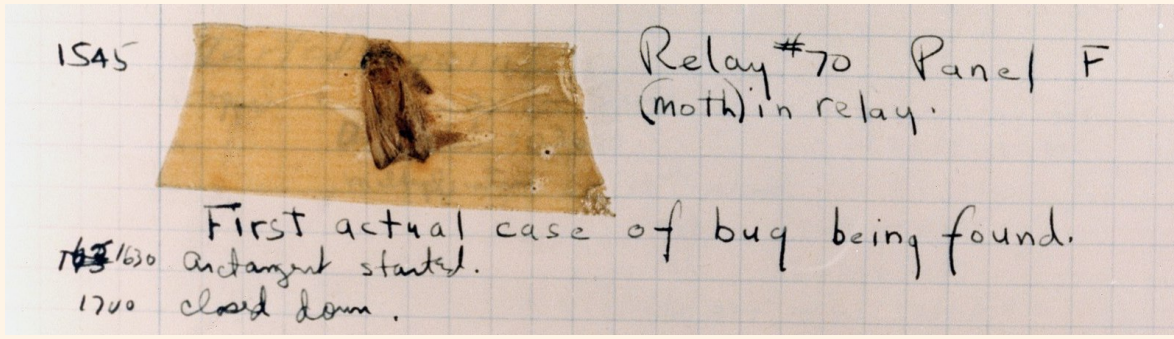


Figure 16.1. The page of the log from the Mark II with the moth taped on it. It dates 9 September 1947. The time is 15:45 as visible on the top left. The log book is at the Smithsonian Institution's National Museum of American History in Washington, D.C., United States.

[a] https://en.wikipedia.org/wiki/Debugging
[b] https://en.wikipedia.org/wiki/Grace_Hopper
[c] https://en.wikipedia.org/wiki/Divide_and_rule
[d] https://en.wikipedia.org/wiki/Algorithm

Let's code!

1. *Trivia night!* Trivia is a quiz game where players have to answer questions about various subjects. For this implementation of Trivia, prepare 3 questions and their corresponding answers for 3 different topics. Ask the player to pick a topic, and then ask a randomly picked question about that topic. Finally, tell the player whether

the answer is correct. If not, print the correct answer. Here are some hints:

- How do you organize your questions and answers? What Python data types do you use?
- What is the sequence of actions you need to perform? Write them down before coding. You can always update them while implementing.
- How do you test that your code is correct?
- Remember to divide and conquer!

PART 5

THE WHILE LOOP AND CONDITIONS

In this part, you will learn the last construct in coding: the `while` loop. You will also learn various types of conditions that you can use in both `while` loops and `if/elif/else` statements. Let's go!

17. Do you want more candies?

The while loop

In coding, there are three constructs: `if/elif/else`, `for` loops, and `while` loops. You have now mastered the first two, and in this chapter, you will finally learn the `while` loop! Read the code below, and try to understand what it does. Follow along with Notebook 17!

[]:

```
# initialize variable
number_of_candies = 0

# print the initial number of candies
print("You have " + str(number_of_candies) +
" candies")


# ask if one wants a candy
answer = input("Do you want a candy?
(yes/no)")

# as long as the answer is yes
while answer == "yes":

    # add a candy
    number_of_candies += 1

    # print the current number of candies
    print("You have " + str(number_of_candies)
    + " candies")


    # ask again if they want more candies
    answer = input("Do you want more candies?
    (yes/no)")

# print the final number of candies
print("You have a total of " +
str(number_of_candies) + " candies")
```

```
initialize variable
number of candies is assigned zero

print the initial number of candies
print You have  concatenated with str
number of candies concatenated with
candies

ask if one wants a candy
answer is assigned input Do you want a
candy? (yes/no)

as long as the answer is yes
while answer equals yes

add a candy
number of candies increased by one

print the current number of candies
print You have  concatenated with str
number of candies concatenated with
candies

ask again if they want more candies
answer is assigned input Do you want
more candies? (yes/no)

print the final number of candies
print You have a total of concatenated
with str number of candies concatenated
with candies
```

Complete the following exercise to start getting to know the functionality and syntax of the `while` loop!

✎ True or false?

1. while is a variable. T F
2. The while loop header contains a condition. T F
3. The variable answer appears 2 times in the code. T F
4. The variable number_of_candies increases by one unit at each loop. T F
5. The while loop continues as long as the player inputs yes and stops when the player inputs no. T F

Computational thinking and syntax

Let's run the cell, and let's analyze the code in two separate blocks. We'll start with the first block:

[1]:

```
1  # initialize variable
2  number_of_candies = 0
3
4  # print the initial number of candies
5  print("You have " + str(number_of_candies) +
   " candies")
```

```
initialize variable
number of candies is assigned zero

print the initial number of candies
print You have  concatenated with str
number of candies concatenated with
candies
```

We create a variable called number_of_candies and initialize it to 0 (line 2). This variable will keep count of the number of candies we want. It is a very important variable, and we will talk about it again when analyzing the second block of code. At line 5, we print the initial number of candies we have, which is zero.

Let's look into the next block, which is the core of the whole code:

```
7  # ask if one wants a candy
8  answer = input("Do you want a candy?
   (yes/no)")
9
10 # as long as the answer is yes
11 while answer == "yes":
12
13     # add a candy
14     number_of_candies += 1
15
16     # print the current number of candies
17     print("You have " + str(number_of_candies)
       + " candies")
18
19     # ask again if they want more candies
20     answer = input("Do you want more candies?
       (yes/no)")
21                                          ↳
```

```
ask if one wants a candy
answer is assigned input Do you want a
candy? (yes/no)

as long as the answer is yes
while answer equals yes

add a candy
number of candies increased by one

print the current number of candies
print You have  concatenated with str
number of candies concatenated with
candies

ask again if they want more candies
answer is assigned input Do you want
more candies? (yes/no)
```

```
22 # print the final number of candies
23 print("You have a total of " +
   str(number_of_candies) + " candies")
```

```
print the final number of candies
print You have a total of concatenated
with str number of candies concatenated
with candies
```

```
You have 0 candies
Do you want a candy? (yes/no) yes
You have 1 candies
Do you want more candies? (yes/no) yes
You have 2 candies
Do you want more candies? (yes/no) no
You have a total of 2 candies
```

Let's see how the `while` loop works. We ask the player whether they want a candy, and we save the reply in the variable `answer` (line 8). Then, we continue with the `while` loop header, which says something like: as long as the variable `answer` is equal to `yes`, do the following (line 11): add a unit to the variable `number_of_candies` (line 14); print the current number of candies (line 17), and ask again the player if they want more candies (line 20). Then, we go back to the `while` loop header (line 11). If the answer at line 20 was `yes`, we'll do the same as above, that is: add a unit to the variable `number_of_candies` (line 14); print the current number of candies (line 17), and ask again the player if they want more candies (line 20). Then, we will go back to the `while` loop header again (line 11). If the answer at line 20 was `yes` again, we will do the same as above once more, that is: add a unit to the variable `number_of_candies` (line 14), ... We'll keep doing this **as long as** the variable `answer` is equal to `yes`. What if the player answers `no` at line 20? When we go back to the `while` loop header (line 11), the condition is not valid anymore, because `answer` is not equal to `yes`! So the loop stops, and we go directly to the first line after the `while` loop body (line 23). There, we print the total number of candies.

Let's now look into the syntax. The `while` loop starts with a **header** (line 11), which is composed of three parts: (1) the keyword `while`, (2) a condition, and (3) colon `:` (every construct header ends with a colon!). In this example, we check whether the value assigned to the variable `answer` equals the string `"yes"`. We will see other kinds of conditions in the next chapter. After the header, there is the **body** of the `while` loop (lines 13–20). The body is **indented**, similarly to the `for` loop body and `if/elif/else` statements. Let's now focus our attention on two variables: `answer` and `number_of_candies`.

How many times do you see the variable `answer` and where? `answer` is in **three** different places: (1) **before** the `while` loop, (2) **in the condition** of the `while` loop, and (3) **in the body** of the `while` loop. Why do we need it three times? Before a `while` loop, we always have to **initialize the variable** contained in the condition of the `while` loop header, otherwise, we cannot evaluate the condition itself when the loop starts. In our example, we initialize `answer` with the first player's answer (line 8). Then, we have to **check the condition** involving the variable `answer`. In this case, we check if `answer` is equal to `yes` (line 11). Finally, we have to **allow the variable to change** (line 20), so that the loop can terminate; otherwise, the loop will keep going indefinitely. Sooner or later, we all forget this last part, and we get into an **infinite loop**! If that happens to you, just stop the cell (if it takes too long, restart the kernel!)

Let's finally look into the variable `number_of_candies`. How many times do you see it and where? `number_of_candies` is in **two** places: (1) **before** the `while` loop, where it is **initialized** (line 2) and (2) **in** the `while` loop, where it is **incremented** by one unit at every loop (line 14). The variable `number_of_candies` is generally called **counter** because **it keeps count of the number of loops**. The symbol **`+=`** is an **assignment operator**, and we can pronounce it as *increased by*. It is a compact way of writing `number_of_candies = number_of_candies + 1`. For any arithmetic operator, there is the

associated assignment operator, that is, `-=` (*decreased by*), `*=` (*multiply by and reassign*), `/=` (*divide by and reassign*), etc. Note that in assignment operators, the symbol `=` is always in the second position, after an arithmetic operator.

What is the difference between a `for` loop and a `while` loop? In Chapter 8, we defined the `for` loop as follows:

> A **`for` loop** is the **repetition** of a group of commands
> for a **determined** number of times

In a **`for` loop, we know exactly how many times** we are going to run the commands in the loop body. Conversely, **in a `while` loop we do not know how many times** we are going to run the commands in the loop body because the duration of a `while` loop depends on the validity of the condition in the header. Let's define the `while` loop and summarize its characteristics:

> A **`while` loop** is the repetition of a group of commands
> **as long as a condition** holds

A `while` loop stops when the condition in the header is not true anymore. As we mentioned earlier, we always have to give the variable in the condition the possibility to change so that the condition in the header can be false and the loop can stop. If the variable in the condition (`answer` in our example) cannot change in the `while` loop body, then we will get an infinite loop. Finally, to know how many times we run the loop, we can use a counter (`number_of_candies` in our example) to keep track of the number of iterations. The presence of a counter is not compulsory.

✎ Insert into the right column

So far, you have learned several operators: arithmetic, assignment, and comparison operators. Insert each symbol in the right column:

```
+, ==, *=, <, /, *, <=, =, //=, /=, //, !=, -=, -, +=, >=, %=, **, %, **=, >
```

Arithmetic operators	Assignment operators	Comparison operators

Recap

- A `while` loop is the repetition of a group of commands as long as a condition holds.
- The variable in the condition must be initialized before the condition. It also has to change somewhere in the loop body so that the loop can stop when the condition does not hold anymore.
- A `while` loop can have a counter. Counters keep track of the number of loops and must be initialized before the loop header.
- When updating a variable with an arithmetic operation, we can use the corresponding assignment operator, that is, `+=`, `-=`, etc.

Writing code is like writing an email!

What steps do we do when writing an email? We start with recipient's address and email subject, then we continue with the salutation, the body of the email, greetings, and we finish with signature (an algorithm, isn't it?). Once we are done, we read the email again for a check. We correct some misspellings, and we quickly edit a few things here and there. Often, we go deeper: we reformulate some sentences or we completely rearrange some paragraphs. Without realizing it, we have gone through the email a couple of times! Now, think about the steps we make when writing code. First, we write the imports, the variables, and the implementation of an algorithm. Then we test it to we check whether it works, and if not, we correct it. Once it finally works, we remove unused variables, compact some code lines, improve variable names, and clean comments. Like we do for emails, we look at our code circularly, that is, from top to bottom a couple of times, exactly like when we re-read an email. But for some reason, when we code, we often want the first draft to be the final implementation, and we get frustrated if this doesn't happen. When writing code, **consider the time you spend testing, debugging, and improving the code as part of the process**, not as some extra time that prevents you from doing something else! It's all part of the process!

Let's code!

1. For each of the following scenarios, create code similar to that presented in this chapter:
   a. *Do you want more cookies?*
   b. *Do you want less exercises?*
2. *At the cheese shop.* You own a cheese shop, and you sell slices of cheese at 50c each. A new customer comes in, and you ask if they want cheese. The customer is uncertain of how much cheese to buy, so after every slice, you ask again if they want another slice of cheese. As long as the customer says yes, you add a slice of cheese, update the final price, and tell them the amount of slices of cheese and the price so far. How many slices of cheese did you sell? And what was the final price?
3. *Playing with numbers.* Given the following list: `numbers = [0]`, ask the player if you should add another number to the list. As long as the player says yes, add to the list the sum of the last number you added and the counter of the current loop. *Example*: If you run the `while` loop 7 times, you will get the following list: `[0, 1, 3, 6, 10, 15, 21, 28]`.

4. *Generating even numbers.* Given an empty list, ask the player if you should add another number to the list. As long as the player says yes, create a random number between 0 and 100, and if the number is even, then add it to the list. How many numbers did you generate? How many even? How many odd? What is the ratio between the amount of even and odd numbers you generated?

18. Animals, unique numbers, and sum

Various kinds of conditions

In the previous chapter, we saw only one kind of condition in a `while` loop—that is, that a variable is equal to `yes`. Let's now take a look at three examples with other kinds of conditions. First, try to solve each task by yourself: read the requirements carefully, list the steps to execute, implement them one by one, and merge the code into the solution (divide and conquer!). This time, also try to take it one step further: keep an eye on the processes that your mind goes through while solving the tasks. **You will often find recurring thinking patterns when coding**. Knowing and recognizing them will give you awareness and thus speed up your work. Compare your though processes with the ones proposed here. Maybe they will be similar, or maybe they will be different. In any case, you will get an idea of possible thinking patterns to solve computational tasks. Enough talk—let's start coding! You can play with the proposed solutions in Notebook 18.

1. Guess the animal!

- Given the following list:

```
1  animals = ["giraffe", "dolphin", "penguin"]
```

```
animals is assigned giraffe, dolphin,
penguin
```

- Create a game in which the computer randomly picks one of the three animals and the player has to guess the animal picked by the computer. Make sure that the player keeps playing until she guesses the animal picked by the computer. At the end of the game, tell the player how many attempts it took to guess the animal.

Let's divide and conquer! The game has four requirements: (1) the computer randomly picks one of the three animals, (2) the player has to guess the animal picked by the computer, (3) the player keeps playing until they guess the animal picked by the computer, and (4) at the end of the game, tell the player how many attempts it took to guess the animal. Let's see how to implement each requirement!

1. *The computer randomly picks one of the three animals.* This is pretty straightforward:

```
1  import random
2
3  # computer pick
4  computer_pick = random.choice(animals)
5  print(computer_pick)
```

```
dolphin
```

```
import random

computer pick
computer pick is assigned random dot
choice animals
print computer pick
```

We import the package `random` (line 1), and we use its function `.choice()` to make the computer pick a random element from the list `animals` (line 4). Then, we print `computer_pick` as a check (line 5).

2. *The player has to guess the animal picked by the computer.* This task is also easy:

```
1  # player guess
2  player_guess = input("Guess the animal! Choices: giraffe, dolphin, penguin:")
```

```
player guess
player guess is assigned input Guess the animal! Choices: giraffe, dolphin, penguin:
```

```
Guess the animal! Choices: giraffe, dolphin, penguin: giraffe
```

We use the built-in function `input()` to ask the player to input their guess (line 2). We assume that the player's input is `giraffe`.

3. *The player keeps playing until they guess the animal picked by the computer.* The phrase "*until* they guess the animal" is equivalent to "*as long as* they guess the animal", which immediately suggests to us that we should use a `while` loop. What condition do we write in the header? Let's see:

```
1  # as long as the player's guess and the computer's pick are different
2  while player_guess != computer_pick:
3
4      # tell the player that the animal is not right
5      print("That's not the right animal!")
6
7      # ask the player to guess again
8      player_guess = input("Try again! Guess the animal! Choices: giraffe, dolphin, penguin:")
9
10 # tell the player that they guessed the right animal
11 print("Well done! You guessed " + computer_pick)
```

```
as long as the player's guess and the computer's pick are different
while player guess not equal to computer pick

    tell the player that the animal is not right
    print That's not the right animal!

    ask the player to guess again
    player guess is assigned input Try again! Guess the animal! Choices: giraffe, dolphin, penguin:

tell the player that they guessed the right animal
print Well done! You guessed concatenated with computer pick
```

```
That's not the right animal!
Try again! Guess the animal! Choices: giraffe, dolphin, penguin: dolphin
Well done! You guessed dolphin
```

The loop must stop when the player guesses the animal, that is, when `player_guess` and `computer_pick` are the same. In general, **when a requirement defines the condition that stops a `while` loop**, we have to think the opposite way: **we need to find the condition that allows the `while` loop to keep going**. In our example, the loop must keep going as long as `player_guess` is not equal to `computer_pick` (line 2). In the loop body, we provide a feedback to the player saying that the animal they picked in not right (line 5), and we ask the player to guess the animal again (line 8) so that the `while` loop can continue. Finally, after the loop, we print a message confirming that the player guessed the right animal (line 11).

4. *At the end of the game, tell the player how many attempts it took to guess the animal.* We definitely need a counter!

```
1  # initializing the counter
2  n_of_attempts = 1
3
4  # as long as the player's guess and the
   computer's pick are different
5  while player_guess != computer_pick:


6
7      # tell the player that the animal is
       not right
8      print("That's not the right animal!")
9
10     # print the numbers of attempts so far
11     print("Number of attempts so far: " +
       str(n_of_attempts)
12
13     # increase the number of attempts
14     n_of_attempts += 1
15
16     # ask the player to guess again
17     player_guess = input("Try again! Guess
       the animal! Choices: giraffe, dolphin,
       penguin:")
18
19 # tell the player that they guessed the
   right animal
20 print("Well done! You guessed " +
   computer_pick + " at attempt number " +
   str(n_of_attempts))
```

```
initializing the counter
n of attempts is assigned one

as long as the player's guess and the
computer's pick are different
while player guess not equal to
computer pick


    tell the player that the animal is not
    right
    print That's not the right animal!

    print the numbers of attempts so far
    print Number of attempts so far:
    concatenated with str n of attempts

    increase the number of attempts
    n of attempts increased by one

    ask the player to guess again
    player guess is assigned input Try
    again! Guess the animal! Choices:
    giraffe, dolphin, penguin:

tell the player that they guessed the
right animal
print Well done! You guessed
concatenated with computer pick
concatenated with at attempt number
concatenated with str n of attempts
```

```
That's not the right animal!
Number of attempts so far: 1
Try again! Guess the animal! Choices: giraffe, dolphin, penguin: dolphin
Well done! You guessed dolphin at attempt number 2
```

We create the counter `n_of_attempts` (line 2), and we initialize it to `1`. Why `1` and not to `0`? Because the player enters the first input before the `while` loop (see requirement 2. *The player has to guess the animal picked by the computer*), and that is the first attempt! Then, we tell the player the current number of attempts (line 11) and increase `n_of_attempts` by one unit at every loop (line 14). Finally, we include the total number of attempts to the last print (line 20).

After solving the four tasks, we can merge the code together! Here is the complete solution:

[1]:
```
1  import random
2
3  # computer pick
4  computer_pick = random.choice(animals)
5  # print(computer_pick)
6
7  # player guess
8  player_guess = input("Guess the animal!
   Choices: giraffe, dolphin, penguin:")
9
10 # initializing the counter
11 n_of_attempts = 1
12
13 # as long as the player's guess and the
   computer's pick are different
14 while player_guess != computer_pick:
15
16     # tell the player that the animal is
       not right
17     print("That's not the right animal!")
18
19     # print the numbers of attempts so far
20     print("Number of attempts so far: " +
       str(n_of_attempts)
21
22     # increase the number of attempts
23     n_of_attempts += 1
24
25     # ask the player to guess again
26     player_guess = input("Try again! Guess
       the animal! Choices: giraffe, dolphin,
       penguin:")
27
28 # tell the player that they guessed the
   right animal
29 print("Well done! You guessed " +
   computer_pick + " at attempt number " +
   str(n_of_attempts))
```

```
import random

computer pick
computer pick is assigned random dot
choice animals
print computer pick

computer pick
player guess is assigned input Guess
the animal! Choices: giraffe, dolphin,
penguin:

initializing the counter
n of attempts is assigned one

as long as the player's guess and the
computer's pick are different
while player guess not equal to
computer pick

tell the player that the animal is not
right
print That's not the right animal!

print the numbers of attempts so far
print Number of attempts so far:
concatenated with str n of attempts

increase the number of attempts
n of attempts increased by one

ask the player to guess again
player guess is assigned input Try
again! Guess the animal! Choices:
giraffe, dolphin, penguin:

tell the player that they guessed the
right animal
print Well done! You guessed
concatenated with computer pick
concatenated with at attempt number
concatenated with str n of attempts
```

```
Guess the animal! Choices: giraffe, dolphin, penguin: giraffe
That's not the right animal!
Number of attempts so far: 1
Try again! Guess the animal! Choices: giraffe, dolphin, penguin: dolphin
Well done! You guessed dolphin at attempt number 2
```

Note that we commented out the print of `computer_pick` (line 5), as the final code is for a player and not for a coder!

2. Create a list of 8 unique random numbers

Here is our next task:

- Create a list of 8 random numbers between 0 and 10. Make sure they are unique, meaning each number is present only once in the list. If the number is already in the list, print: `The number x is already in the list`. How many numbers did you generate to find 8 unique numbers?

Let's divide and conquer once more! The task has four requirements: (1) create a list of 8 random numbers between 0 and 10, (2) make sure they are unique, that is, each number is present only once in the list, (3) if the number is already in the list, print: `The number x is already in the list`, and (4) how many numbers did you generate to find 8 unique numbers? Let's go through the requirements one by one!

1. *Create a list of 8 random numbers between 0 and 10.* According to this requirement only, we can create a list of 8 numbers using a `for` loop and the function `.randint()` from the module `random`:

```
1  import random
2
3  # initialize the number list
4  unique_random_numbers = []
5
6  # for 8 times
7  for _ in range(8):
8
9      # create a random number between 0 and 10
10     unique_random_numbers.append
                   (random.randint(0,10))
11
12 # print the list
13 print(unique_random_numbers)
```

```
[7, 9, 3, 2, 3, 0, 9, 6]
```

```
import random

initialize the number list
unique random numbers is assigned
empty list

for eight times
for underscore in range eight

create a random number between zero
and ten
unique random numbers dot append
random dot randint zero ten

print the list
print unique random numbers
```

We import the package `random` (line 1), and we initialize `unique_random_numbers`—which will contain the created numbers—to an empty list (line 4). Then, we create a `for` loop, where we generate eight random numbers between 0 and 10, and we append them to `unique_random_numbers` (lines 6–10). Note that we use an underscore instead of the variable `i` in the loop header because we do not need `i` in the loop body (see the *In more depth* section *What if I don't use the index in a `for` loop?* in Chapter 15). Finally, we print `unique_random_numbers` to check that it actually contains eight random numbers (line 13). Let's go to the next requirement!

2. *Make sure they are unique, which means each number is present only once in the list.* In the list we printed above, the numbers are not unique: both 3 and 9 are present twice. Thus, we need to modify our code. How? We do not know how many random numbers we need to generate before obtaining 8 unique numbers, that is, we do not know how many times we need to run the command `unique_random_numbers.append(random.randint(0,10))` (line 9 in the cell above). For this reason, we cannot use a `for` loop—which we use when we know the exact number of iterations—but we need to use a `while` loop, which we use when the number of iterations is determined by a condition. **Making changes in code during the drafting process is normal**, as we mentioned in the *In more depth* section *Writing code is like writing an email!* in the previous chapter. What condition do we use in this

while loop? The list must be composed of 8 elements, thus its length has to be 8. Let's see how we can transform the previous code:

```
1  import random
2
3  # initialize the number list
4  unique_random_numbers = []
5
6  # as long as the length of the list is not 8
7  while len(unique_random_numbers) != 8:
8
9      # create a random number between 0 and 10
10     number = random.randint(0,10)
11
12     # if the number is already in the list
13     if number in unique_random_numbers:
14         # place holder
15         a = 0
16     # otherwise
17     else:
18         # add the new number to the list
19         unique_random_numbers.append(number)
20
21 # print the list
22 print(unique_random_numbers)
```

```
[1, 8, 10, 7, 3, 0, 5, 9]
```

```
import random

initialize the number list
unique random numbers is assigned
empty list

as long as the length of the list is
not eight
while len unique random numbers not
equal to eight

create a random number between zero
and ten
number is assigned random dot randint
zero ten

if the number is already in the list
if number in unique random numbers
place holder
a is assigned zero
otherwise
else
add the new number to the list
unique random numbers dot append
number

print the list
print unique random numbers
```

At line 7, we substitute the header of the `for` loop with the header of a `while` loop, with the condition that the loop **keeps going** as long as the length of the list is not equal to 8. Then, we generate a random number (line 10). We need to make sure that the random number is a new one (or unique!) before adding it to the list. Thus, we create an `if ... in / else` construct (lines 12–19), which we learned in Chapter 3. If the number is already in the list (line 13), then we do not want to add it to the list. The next requirement will tell us what to do, so for now we just write `a = 0` (line 15) as a placeholder—that is, a temporary command that we plan to replace later. Using placeholders is not very good coding practice, but sometimes we can make an exception in the very early drafting phase. Finally, if the number is not in the list (`else` at line 17), then we append it to the list (line 19).

3. *If the number is already in the list, print*: `The number x is already in the list`. We substitute the placeholder `a = 0` with the print command (line 15):

```
1  import random
2
3  # initialize the number list
4  unique_random_numbers = []
5
6  # as long as the length of the list is not 8
7  while len(unique_random_numbers) != 8:
8
9      # create a random number between 0 and 10
10     number = random.randint(0,10)
11
12     # if the number is already in the list
13     if number in unique_random_numbers:
14         # print that the number is in the list
15         print("The number " + str(number) +
           " is already in the list")
16     # otherwise
17     else:
18         # add the new number to the list
19         unique_random_numbers.append(number)
20
21 # print the list
22 print(unique_random_numbers)
```

```
import random

initialize the number list
unique random numbers is assigned
empty list

as long as the length of the list is
not eight
while len unique random numbers not
equal to eight

create a random number between zero
and ten
number is assigned random dot randint
zero ten

if the number is already in the list
if number in unique random numbers
print that the number is in the list
print The number concatenated with
str number concatenated with is
already in the list
otherwise
else
add the new number to the list
unique random numbers dot append
number

print the list
print unique random numbers
```

```
The number 1 is already in the list
The number 10 is already in the list
The number 7 is already in the list
The number 5 is already in the list
[1, 8, 10, 7, 3, 0, 5, 9]
```

As we can see in the printouts, the numbers 1, 10, 7, and 5 were generated twice, but they are in the list only once!

4. How many numbers did you generate to find 8 unique numbers?
To satisfy this last requirement, we need a counter. It will keep track of the amount of numbers we generated, which coincides with the number of iterations of the `while` loop!

[2]:

```
1  import random
2
3  # initialize the number list
4  unique_random_numbers = []
5
6  # initialize the counter
7  counter = 0
8
9  # as long as the length of the list is not 8
10 while len(unique_random_numbers) != 8:
11
12     # create a random number between 0 and 10
13     number = random.randint(0,10)
14
15     # increase the counter by 1
16     counter += 1
17
18     # if the number is already in the list
19     if number in unique_random_numbers:
20         # print that the number is in the list
21         print("The number " + str(number) +
           " is already in the list")
22     # otherwise
23     else:
24         # add the new number to the list
25         unique_random_numbers.append(number)
26
27 # print the final list and the total amount
   of generated numbers
28 print(unique_random_numbers)
29 print("The total amount of generated numbers
   is: + str(counter))
```

```
import random

initialize the number list
unique random numbers is assigned
empty list

initialize the counter
counter is assigned zero

as long as the length of the list is
not eight
while len unique random numbers not
equal to eight

create a random number between zero
and ten
number is assigned random dot randint
zero ten

increase the counter by one
counter increased by one

if the number is already in the list
if number in unique random numbers
print that the number is in the list
print The number concatenated with
str number concatenated with is
already in the list
otherwise
else
add the new number to the list
unique random numbers dot append
number

print the final list and the total
amount of generated numbers
print unique random numbers
print The total amount of generated
numbers is: concatenated with str
counter
```

```
The number 1 is already in the list
The number 10 is already in the list
The number 7 is already in the list
The number 5 is already in the list
[1, 8, 10, 7, 3, 0, 5, 9]
The total amount of generated numbers is: 12
```

We initialize the counter (line 7), increment it by one unit at each iteration (line 16), and print it (line 29).

3. Sum the multiples of 3

- Write code that continues asking a player to enter an integer until he enters a negative number. At the end, print the sum of all entered integers that are multiples of 3.

In this final task there are two requests: (1) keep asking a player to enter an integer until he enters a negative number and (2) at the end, print the sum of all entered integers that are multiples of 3. Let's see how to implement them!

1. *Keep asking a player to enter an integer until he enters a negative number*. The requirement is straightforward: we use the `input` function to ask the player to enter numbers and a `while` loop to keep asking. Which condition do we use in the header? Let's have a look:

```
1  # ask the user for an integer                    ask the user for an integer
2  number = int(input("Enter an integer:"))         number is assigned int input Enter
                                                    an integer:
3
4  # as long as the number is positive              as long as the number is positive
5  while number >= 0:                               while number greater than or equal
                                                    to zero
6      # ask for the next new integer               ask for the next new integer
7      number = int(input("Enter another            number is assigned int input Enter
       integer: "))                                 another integer:
```

```
Enter an integer: 3
Enter another integer: 6
Enter another integer: 4
Enter another integer: -1
```

The loop must continue as long as the player enters a negative number, that is, as long as `number` is positive—greater than or equal to zero (line 5). As we learned in the previous chapter, the variable in the condition has to be in three places: before the loop, in the loop header, and within the loop. Thus, first we initialize the variable `number` with the integer entered by the player (line 2). Then, we use `number` in the condition in the `while` loop header, as we mentioned above (line 5). And finally, to avoid an infinite loop, we ask the player to enter a new number (line 7). Let's implement the second requirement!

2. *At the end, print the sum of all entered integers that are multiples of* 3. We need to check whether the numbers the user enters are multiples of 3, and, if they are, we have to sum them. Ideas on how to do it? Let's start drafting the code:

```
1  # list containing the numbers to sum
2  numbers = []
3
4  # ask the user for an integer
5  number = int(input("Enter an integer:"))
6
7  # as long as the number is positive
8  while numbers >= 0:
9
10     # if the number is multiple of 3
11     if numbers % 3 == 0:
12         # add the number to the list
13         numbers.append(number)
14
15     # ask for the next integer
16     number = int(input("Enter another
       integer: "))
17
18 # print the list of multiples of 3
19 print(numbers)
20
21 # initialize the sum to 0
22 sum_of_numbers = 0
23
24 # calculate the sum of numbers
25 for i in range(len(numbers)):
26     sum_of_numbers = numbers[i] +
       sum_of_numbers
27
28 # print the final sum
29 print("The sum of the multiples of 3 is:  +
   str(sum_of_numbers))
```

```
list containing the numbers to sum
numbers is assigned empty list

ask the user for an integer
number is assigned int input Enter
an integer:

as long as the number is positive
while number greater than or equal
to zero

if the number is multiple of three
if number mod three equals zero
add the number to the list
numbers dot append number

ask for the next integer
number is assigned int input Enter
another integer:

print the list of multiples of three
print numbers

initialize the sum to zero
sum of numbers is assigned zero

calculate the sum of numbers
for i in range len numbers
sum of numbers is assigned numbers
in position i plus sum of numbers

print the final sum
print The sum of the multiples of
3 is:  concatenated with str sum of
numbers
```

```
Enter an integer: 3
Enter another integer: 6
Enter another integer: 4
Enter another integer: -1
[3, 6]
The sum of the entered multiples of 3 is: 9
```

We create an empty list called `numbers` that will contain the multiples of 3 (line 2). Then, within the `while` loop, we add an `if` construct, in which we check whether the current number is a multiple of 3 by using the modulo operation (line 11). If the condition is met, then we append the number to the list `numbers` (line 13). At the end of the `while` loop (i.e., after the player has entered a negative number), we sum the numbers in the list, using an approach similar to the one you might have used to solve coding exercise 5 in Chapter 14. First, we create the variable `sum_of_numbers`, which will contain the final sum, and we initialize it to zero (line 22). Then, we use a `for` loop through the list `numbers`—containing the multiples of 3 (line 25)—to add the current list element (`numbers[i]`) to the amount in `sum_of_numbers` (line 26). Finally, we print the sum at line 29.

The task is solved! But can we improve our code? Let's read the following requirement again: at the end, print the sum of all entered integers that are multiples of 3. We are not asked to save the multiples of 3 in a list—just to print their sum. Thus, do we need to create the list? Not really! So, how do we do it? Let's see this alternative solution:

[3]:
```
1  # initialize the sum to 0
2  sum_of_numbers = 0
3
4  # ask the user for an integer
5  number = int(input("Enter an integer: "))
6
7  # as long as the number is positive
8  while numbers >= 0:
9
10     # if the number is a multiple of 3
11     if numbers % 3 == 0:
12         # add the number to the sum
13         sum_of_numbers += number
14
15     # ask for the next integer
16     number = int(input("Enter another
       integer: "))
17
18 # print the final sum
19 print("The sum of the multiples of 3 is:  +
   str(sum_of_numbers))
```

```
initialize the sum to zero
sum of numbers is assigned zero

ask the user for an integer
number is assigned int input Enter
an integer:

as long as the number is positive
while number greater than or equal
to zero

if the number is a multiple of three
if number mod three equals zero
add the number to the sum
sum of numbers increased by number

ask for the next integer
number is assigned int input Enter
another integer:

print the final sum
print The sum of the multiples of
3 is:  concatenated with str sum of
numbers
```

```
Enter an integer: 3
Enter another integer: 6
Enter another integer: 4
Enter another integer: -1
The sum of the entered multiples of 3 is: 9
```

We remove all the code related to the list `numbers`. We initialize `sum_of_numbers` to zero before the `while` loop (line 2). Then, within the loop, we sum the current multiple of 3 (i.e., `number`) to the total sum (line 13)—without saving it to a list. With this trick, we improve our code in two ways: (1) we do not create a list, which occupies space in computer memory and (2) we avoid a `for` loop that occupies memory and time during the execution. The code thus becomes shorter, faster, and more elegant.

Match the sentence halves

1. An `if/else` construct checks whether a condition is true or false	a. for a determined number of times.
2. A `for` loop is the repetition of a group of commands	b. as long as a condition holds.
3. A `while` loop is the repetition of a group of commands	c. and executes code accordingly.

Recap

- In a `while` loop header, we can write various kinds of conditions. The correct condition is the one that keeps the loop going (not stopping!)
- When solving a task, it is common to decompose and analyze the requirements, solve the subtasks, and merge the code to the solution (divide and conquer!)
- When coding, we often write a first draft, and then we improve the draft to make the code faster and robust (writing code is like writing an email!)

Don't confuse the `while` loop with `if/else`!

When learning coding, it can be easy to confuse the `while` loop with the `if/else` construct. If this happened to you while learning the past two chapters, read the following paragraph. If you feel like you mastered the difference between `while` loops and `if/else` constructs, feel free to skip this *In more depth*!

Consider the following example, similar to the first one in this chapter.

- Given the following list:

```
[1]: 1 fruits = ["mango", "orange", "banana"]
```

```
fruits is assigned mango, orange, banana
```

- Create a game where the computer randomly picks a fruit and the player has to guess the fruit picked by the computer. Make sure that the player keeps playing until they guess the fruit picked by the computer.

We have to solve 3 tasks: (1) the computer randomly picks a fruit, (2) the player has to guess the fruit picked by the computer, and (3) we must make sure that the player keeps playing until they guess the fruit picked by the computer. The first two requirements are straightforward, and we will solve them quickly. We will focus on the third requirement. ↳

1. *The computer randomly picks a fruit.*

```
[2]:
1  import random
2
3  # computer pick
4  computer_pick = random.choice(fruits)
5
```

```
import random

computer pick
computer pick is assigned random
dot choice fruits
```

We import the package random (line 1) and we use the method `.choice()` to make the computer randomly pick an element of the list `fruits`.

2. *The player the has to guess the fruit picked by the computer.*

```
6  # player guess
7  player_guess = input("Guess the fruit!
   Choices: mango, orange, banana:")
8
```

```
player guess
player guess is assigned input
Guess the fruit! Choices: mango,
orange, banana:
```

We use the built-in function `input()` to ask the player to enter a fruit (line 7).

3. *Make sure that the player keeps playing until they guess the animal picked by the computer.* The first instinct could be to do the following:

```
9  # check the player guess
10 if player_guess == computer_pick:

11     print("That's right! The fruit is " +
       computer_pick)
12 else:
13     print("Nope! Try again!")
```

```
check the player guess
if player guess equals computer
pick
print That's right! The fruit is
concatenated with computer pick
else
print Nope! Try again!
```

We check if `player_guess` is equal to `computer_pick` with an `if/else` construct, and we print messages accordingly (lines 9–13). If the player did not guess the right fruit, we have to ask them to guess again (like at line 7). Then, we have to check once more if the guess is correct (like at lines 9–13), and so on. In other words, we would have to repeat the same code several times, without knowing how many times. Every time we need to repeat code, we can use a loop! And if we do not know the number of repetitions, but we have a condition that stops the repetitions, then we use a `while` loop. So, here is the correct solution:

```
9  # as long as the player's guess and the
   computer's pick are different
10 while player_guess != computer_pick:

11     player_guess = input("Nope! Try again!
       Guess the fruit! Choices: mango, orange
       banana:")
```

```
as long as the player's guess and
the computer's pick are different
while player guess not equal to
computer pick
player guess is assigned input
Nope! Try again! Guess the fruit!
Choices: mango, orange, banana:
```

As long as `player_guess` is not equal to `computer_pick` (line 10), we ask the player to make a guess (line 11), which, in the following iteration, we check again in the condition of the `while` loop header (line 10), and the loop keeps going as long as the condition holds.

Let's code!

1. *Guess the number!* Create a game where the computer picks a number between 0 and 10, and the player has to guess it. If the player guesses a number that is too high or too low, then the computer tells the player. The game stops when the player guesses the number. At the end, tell the player how many attempts it took to guess the number.
2. *12 even random numbers.* Create a list of 12 even random numbers between 0 and 30. How many odd numbers did you exclude?
3. *Spelling game for kids.* Create a game that helps kids learn spelling. The game has the following requirements: (1) Create a list of words to be spelled. Among these words, choose a word randomly, and tell the kid the chosen word (e.g., `Spell the word: hello`). (2) The kid has to enter one letter at the time. If the kid enters the correct letter, then provide positive reinforcement (e.g., `Well done!`), and ask for the next letter. If the kid does not enter the correct letter, then tell them that the letter is not correct, and ask for a letter again.

   Challenge 1: Instead of creating only one list of words, create three lists, one per topic, so that the kid can choose a topic before spelling a word.

   Challenge 2: The game continues as long as the kid wants to spell a new word.

19. And, or, not, not in

Combining and reversing conditions

Up to now, we have considered only one condition in `if/else` constructs and `while` loops. What if we need more than one condition? And what if we need to reverse a condition? In this chapter, we will learn how to combine or reverse conditions using the logical operators `and`, `or`, `not`, and the membership operator `not in`. As usual, try to solve the tasks yourself before looking at the solutions, which you can also find in Notebook 19. Let's start!

1. and

- Given the following list of integers:

[1]:
```
1  numbers = [1, 5, 7, 2, 8, 19]
```
```
numbers is assigned one, five, seven,
two, eight, nineteen
```

- Print the numbers that are between `5` **and** `10`:

[2]:
```
1  # for each position in the list
2  for i in range(len(numbers)):
3
4      # if the current number is between 5
       and 10
5      if numbers[i] >= 5 and numbers[i] <= 10:
6
7          # print the current number
8          print("The number " + str(numbers[i])
           + " is between 5 and 10")
```
```
for each position in the list
for i in range len numbers

# if the current number is between
five and ten
if numbers in position i greater
than or equal to five and numbers in
position i less than or equal to ten

print the current number
print The number concatenated with
str numbers in position i concatenated
with is between five and ten
```
```
The number 5 is between 5 and 10
The number 7 is between 5 and 10
The number 8 is between 5 and 10
```

We use a `for` loop to browse all the elements in the list (line 2). Then, we check if each number is between 5 and 10 (line 5). To be in between two numbers, a number must be greater than or equal to the smaller number and smaller than or equal to the greater number. The two conditions—greater than or equal to and smaller than or equal to—must be valid at the same time. To check if two—or more—conditions are valid simultaneously, we join them using the **logical operator and**.

> We use the logical operator **and** when we want to check
> whether **all conditions** are **valid**

Let's look at the syntax. For each condition both before and after the logical operator `and`, we have to write: (1) a variable (e.g., `numbers[i]`), (2) a comparison operator (e.g., `>=`), and (3) a term of comparison (e.g., `5`). At the end of the code, we print the numbers that satisfy both conditions (line 8).

2. or

- Given the following string:

```
[3]:
1 message = "Have a nice day!!!"
```

```
message is assigned Have a nice day!!!
```

- And given all punctuation:

```
[4]:
1 punctuation = "\"\/'()[]{}<>.,;:?!^@~#$%&*_-"
```

```
punctuation is assigned
\"\/'()[]{}<>.,;:?!^@~#$%&*_-
```

The string `punctuation` contains all punctuation on a Latin alphabet keyboard. Compare the symbols with the ones on your keyboard and note whether there are additional ones! If so, add them to `punctuation` in Jupyter Notebook 19. The symbols at the beginning of the string `punctuation` `"\"\/` might be a bit confusing, so let's disentangle them. The first quote `"\"\/` is the symbol that introduces the string. The following two symbols `"\"\/` are escape characters—you might remember the escape character `"\n"`, which creates a new line (Chapter 12). The backslash `\` tells Python that the following quote `"` is an actual quote and not the symbol that we use to open or close a string. The last backslash `"\"\/` is an actual backslash because the following forward slash `/` is not part of an escape character.

- Print and count the number of characters that are punctuation **or** vowels:

```
[5]:
1  # string of vowels
2  vowels = "aeiou"
3
4  # initialize counter
5  counter = 0
6
7  # for each position in the message
8  for i in range(len(message)):
9
10     # if the current element is punctuation
       # or vowel
11     if message[i] in punctuation or
       message[i] in vowels:
12
13         # print a message
14         print(message [i] + " is a vowel
           or a punctuation")
15
16         # increase the counter
17         counter += 1
18
19 # print the final amount
20 print("The total amount of punctuation or
   vowels is " + counter)
```

```
string of vowels
vowels is assigned aeiou

initialize counter
counter is assigned zero

for each position in the message
for i in range len message

# if the current element is
punctuation or vowel
if message in position i in
punctuation or message in position
i in vowels

print a message
print message in position i
concatenated with is a vowel or a
punctuation

increase the counter
counter increased by one

print the final amount
print The total amount of punctuation
or vowels is concatenated with counter
```

```
a is a vowel or a punctuation
e is a vowel or a punctuation
a is a vowel or a punctuation ↳
```

```
i is a vowel or a punctuation
e is a vowel or a punctuation
a is a vowel or a punctuation
! is a vowel or a punctuation
! is a vowel or a punctuation
! is a vowel or a punctuation
The total amount of punctuation or vowels is 9
```

Similarly to what we did for `punctuation`, we create a string containing vowels (line 2). We also create a counter, which we will use to calculate the number of characters that are punctuation or vowels, and we initialize it to zero (line 5). Then, we get to the core of the solution! We use a `for` loop to browse all the characters in the string `message` (line 8). **`for` loops for strings work exactly the same way as `for` loops for lists**. In the loop body, we check if each character is a punctuation or a vowel by using the membership operator `in` (line 11), which we learned in Chapter 3. More specifically, we check if `message[i]` is `in` the string `punctuation` or `in` the string `vowels`. Note that as for the `for` loop, **the membership operator `in` works for strings the same way as it works for lists**. Since only one of the conditions can be valid—a character cannot be both a punctuation and a vowel at the same time!—we merge the two conditions—that is, `message[i] in punctuation` or `message[i] in vowels`—using the **logical operator `or`**.

We use the logical operator **`or`** when we want to check
whether **at least one condition** is **valid**

The syntax is the same as for the logical operator `and`, that is, we need to write: (1) a variable, (2) a comparison operator, and (3) a term of comparison both before and after `or`. To conclude the loop body, we print a message for the characters that satisfy at least one condition (line 14), and we increment the counter by one unit (line 17). At the end of the loop, we print the final number of characters that are vowels or punctuation (line 20).

3. not

- Given the following list of integers:

[6]:
```
1 numbers = [4, 6, 7, 9]
```

```
numbers is assigned four, six, seven, nine
```

- Print the numbers that are **not** divisible by 2:

[7]:
```
1 # for each position in the list
2 for i in range(len(numbers)):
3
4     # if the current number is not even
5     if not numbers[i] % 2 == 0:
6
7         # print the current number
8         print(numbers[i])
```

```
7
9
```

```
for each position in the list
for i in range len numbers

# if the current number is not even
if not numbers in position i mod two equals zero

print the current number
print numbers in position i
```

For each position in the list (line 2), we have to check whether the number is not even. For a moment, let's think about the opposite: what condition would we write if we had to check whether the number is even? `if numbers[i] % 2 == 0`. To negate a condition, we just add the **logical operator `not` before the condition**—more specifically, before the variable at the beginning of the condition (line 5).

> We use the logical operator **`not`** when we want to check
> whether **the opposite of a condition** is **valid**

If the condition is satisfied, then we print the number (line 8).

Is this the only way to solve this task? Maybe the first idea you had in mind was more similar to this solution:

[8]:
```
1  # for each position in the list
2  for i in range(len(numbers)):
3
4      # if the current number is odd
5      if numbers[i] % 2 != 0:
6
7          # print the current number
8          print(numbers[i])
```
```
7
9
```
```
for each position in the list
for i in range len numbers

# if the current number is odd
if numbers in position i mod two not
equal to zero

print the current number
print numbers in position i
```

For each position in the list (line 2), we check whether the remainder of `numbers[i]` divided by `2` is not equal to `0` (line 5). If so, then we print the number (line 8).

What solution is better? It's a matter of preference! If you are undecided, pick the solution that looks like the simplest to you, both in term of syntax and reasoning. In coding, there are often various ways of solving a task. Keeping the solution **simple** favors **readability** and **understanding**.

Last note about conditions: when combining conditions, we need to follow a precise order, similarly to what we do with arithmetic operators (see the *In more depth* section *Solving arithmetic expressions* in Chapter 13). The order from highest to lowest precedence is: `not`, `and`, `or` (easy-to-memorize acronym: **NAO**). Whenever you are uncertain, write the condition to prioritize within round brackets `()`.

4. not in

- Generate 5 random integers between 0 and 10. If the random integers are **not** already **in** the following list, then add them:

[9]:
```
1  numbers = [1, 4, 7]
```
```
numbers is assigned one, four, seven
```

[10]:
```
1  import random
2
```
```
import random
```

```
3  # for five times
4  for _ in range(5):
5
6      # generate a random number between 0 and 10
7      number = random.randint(0, 10)
8
9      # print the number as a check
10     print(number)
11
12     # if the new number is not in numbers
13     if number not in numbers:
14         # add the number to numbers
15         numbers.append(number)
16
17 # print the final list
   print(numbers)
```

```
for five times
for underscore in range five

generate a random number between zero
and ten
number is assigned random dot randint
zero ten
print the number as a check
print number

if the new number is not in numbers
if number not in numbers
add the number to numbers
numbers dot append number

print the final list
print numbers
```

```
6
6
10
7
9
[1, 4, 7, 6, 10, 9]
```

We start by importing the package `random` (line 1). Then, we create a `for` loop that runs for five times (line 4)—note the underscore instead of the variable `i` because we will not need any index in the `for` loop body (see the *In more depth* section *What if I don't use the index in a* `for` *loop?* in Chapter 15). Then, we create a random variable (line 7) and print it as a check (line 9). To **evaluate if the variable** `number` **is not already in the list** `numbers` (line 12), we use the **membership operator `not in`**, which is the opposite of the membership operator `in` (Chapter 3). If the condition is met, then we append the randomly generated number to the list of numbers (line 14). Finally, we print the completed list (line 17).

✎ Insert into the right column

You now know all membership, comparison, and logical operators. Insert each symbol in the right column:

<, or, in, !=, not, >, ==, not in, >=, and, <=

Membership operators	Comparison operators	Logical operators

Recap

- The logical operators are `and`, `or`, and `not`.
- When combining conditions, the order of execution is `not`, `and`, `or` (NAO).
- The membership operators are `in` and `not in`.

What is GitHub?

You might have heard about GitHub, or you might have browsed some code on its site `www.github.com`. Surely, you have checked the solutions of the exercises of this book on GitHub! But what is GitHub exactly? In a simplified manner, we can think of GitHub as a cloud service or a huge server for code. Instead of using Dropbox, Google Drive, etc., coders prefer to synchronize their code with GitHub. GitHub has its own language: folders are called **repositories**, sending files to the server is called a **push**, and getting files from the server is called a **pull**. Each repository contains files—they can store any files, either containing code or not—and elements that are specific to coding, such as **issues**, where anybody can indicate bugs to be solved or suggest new features. Why do coders use GitHub instead of other cloud services? Because GitHub supports **version control**, that is, it **keeps track of code changes over time**. Every time we push a code update, we can compare it with previous versions, and if the new code does not work, then we can go back to an earlier version. Furthermore, GitHub is useful for **collaborative projects**: programmers can work on different sections of a task individually and then integrate the code without accidentally influencing each other's code, all while keeping track of each programmer's contribution. These tasks are actually executed by **Git**, which is a distributed version control system, that is, a software that manages changes to code. Other platforms that employ Git include GitLab (`www.gitlab.com`) and Bitbucket (`www.bitbucket.org`), with GitHub being the most popular.

Let's code!

1. *The Zen of Python.* Solve the following 4 steps, and you will discover the Zen of Python!
   a. Given the following list of strings:
   ```
   strings_to_slice = ["reisk", "kpan", "xfsimpleg", "bosolutionb", "pobetterx", "weorb",
   "ofworsep", "aathanx", "hoau", "hfcomplexx", "poors", "opcomplicatedx", "rwsolutions",
   "re?o"]
   ```
   Create a new list called `sliced_strings` containing the same strings but without the first two letters and the last letter (example: `"gfhio"` will become `"hi"`).
   b. Given the following list of strings:
   ```
   strings_to_invert= ["emos", "elpoep", "kniht", "taht", "xelpmoc", "ro", "detacilpmoc",
   "si", "retteb", "naht", "elpmis"]
   ```
   Create an new list called `inverted_strings` containing the same strings but inverted (example: `"ih"` would become `"hi"`).
   c. Given the following list of strings:
   ```
   strings_to_pick = ["this", "sounds", "simple", "but", "is", "it?", "some", "things",
   "look", "better", "than", "when", "complex", "but", "complex", "again", "is", "worse",
   "better", "than", "complicated", "I'm", "confused"]
   ```

Find the words that are present both in `sliced_strings` and `inverted_strings`, change them to uppercase, and add them to a new list. What sentence do you get?

d. Where does the obtained sentence come from? Run the following Python command: `import this`.

2. *Playing with numbers.* Given the following list of numbers:

```
numbers = [7, 9, 15, 19, 24, 30, 37, 45, 50]
```

   a. Print the numbers that are divisible by 3 and 5.
   b. Print the numbers that are divisible by 3 or 5.
   c. Print the numbers that are divisible by 3 but not 5. Perform this task in two different ways, once using `not`, and once without using `not`.

3. *Upgrading Rock paper scissors.* In Chapter 16, we implemented rock paper scissors. In that version, there were many repetitions. In coding, we usually do not want repetitions because they can lead to errors. How can we make the code less repetitive? By combining conditions! What conditions can you combine in this game? Rewrite rock paper scissors in a more succinct way using logical operators. After you have optimized the code, make it a real game by adding a `while` loop that allows players to play as long as they want. *Hint*: Instead of thinking in terms of computer and player choices, think in terms of outcomes, i.e., tie and the player's (or the computer's) win.

20. Behind the scenes of comparisons and conditions

Booleans

It's finally time to unveil what's behind comparisons and conditions! What does Python "see" when we write a comparison or a condition? Let's find it out with the code below! Follow along with Notebook 20.

1. Single comparison or condition

- Given the following assignment:

```
[1]: 1 number = 5                    number is assigned five
```

- What is the outcome of the following comparison operation?

```
[2]: 1 print(number > 3)              print number greater than three
     True
```

The printed value is `True`. In fact, it is true that 5 is greater than 3! But what is `True`? A string? A variable? Let's figure it out in the next cell!

- Assign the above operation to a variable and print it. What type is it?

```
[3]: 1 result = number > 3           result is assigned number greater than three
     2 print(result)                 print result
     3 type(result)                  type result
     True
     bool
```

We assign the result of the comparison operation `number > 3` to the variable `result` (line 1). Then, we print `result` (line 2) and we get `True`—like in cell 2. Finally, we print the outcome of `type(result)` to determine the type of the variable `result`, which is `bool` (line 3)—we mentioned the built-in function `type()` in Chapter 13. We say that the variable `result` is of **type Boolean** and its value is `True`. Booleans are a data type exactly like strings, lists, integers, etc.

Let's continue our exploration of what lies behind comparisons and conditions. Let's look at this example:

- What is the outcome of the following comparison operation?

```
[4]: 1 print(number < 3)              print number less than three
     False
```

This time, the print is `False`. Obviously, 3 is not smaller than 5. Let's continue, similarly to what we did in cell 3.

- Assign the above operation to a variable and print it. What type is it?

[5]:
```
1 result = number < 3
2 print(result)
3 type(result)
```
```
result is assigned number less than three
print result
type result
```
```
False
bool
```

We assign the output of the comparison operation `number < 3` to the variable `result` (line 1), and we print it (line 2), obtaining `False`, like in cell 4. Then, we print the type of the variable `result` (line 3) and we get `bool`, like it happened for `True`. To summarize:

> Booleans are a variable **type**. They can have only two values: **`True`** or **`False`**

When we write conditions in an `if/else` construct or in a `while` loop header, Python "reads" the result behind the conditions: that is, **`True`** or **`False`**. For example, when we write:

```
1 if numbers > 3:
2     print("Correct!")
```
```
if number greater than three
print Correct!
```

Python "sees":

```
1 if True:
2     print("Correct!")
```
```
if True
print Correct!
```

2. Combining comparisons or conditions

Let's take a step further and see what happens when we combine conditions.

- What is the outcome of the following comparison operations?

[6]:
```
1 number = 3
2 print(number > 1)
3 print(number < 5)
4 print(number > 1 and number < 5)
```
```
number is assigned 3
print number greater than one
print number less than five
print number greater than one and number less
than five
```
```
True
True
True
```

We assign `3` to the variable `number` (line 1). Then, we print the outcome of three comparison operations. For all operations—`number > 1` (line 2), `number < 5` (line 3), and `number > 1 and number < 5` (line 4)—the outcome is `True`. The outcomes at lines 2 and 3 follow the same principles as in cell 2. Let's focus on line 4, where we combine two comparison operations with the logical operator `and`. For these combined operations, Python "sees":

```
4 print(True and True):
```
```
print True and True
```
```
True
```

As we can see, the output of two `True` conditions combined by the logical operator `and` is `True`.

- What happens if we change the first condition to be `False`?

[7]:
```
1 number = 3
2 print(number > 4)
3 print(number < 5)
4 print(number > 4 and number < 5)
```
```
False
True
False
```
```
number is assigned three
print number greater than four
print number less than five
print number greater than four and number less
than five
```

The first condition is now `False` because `3` is not larger than `4` (line 2), whereas the second condition is still `True` (line 3). The combination of the `False` condition from line 2 with the `True` condition from line 3 returns `False` (line 4). In this last case, Python "sees":

```
4 print(False and True):
```
```
False
```
```
print False and True
```

Thus, the output of one `True` and one `False` conditions merged by the logical operator `and` is `False`. Let's continue analyzing the remaining combinations!

- What happens if we change the second condition to be `False`?

[8]:
```
1 number = 3
2 print(number > 1)
3 print(number < 2)
4 print(number > 1 and number < 2 )
```
```
True
False
False
```
```
number is assigned three
print number greater than one
print number less than two
print number greater than one and number less
than two
```

The first condition is `True` (line 2)—like it was in cell 6—whereas the second condition is now `False` because `3` is not smaller than `2` (line 3). Similarly to cell 7, the combination of one `True` condition and one `False` condition (line 4) returns `False`. In this case, Python "reads":

```
4 print(True and False):
```
```
False
```
```
print True and False
```

We can deduce that the output of one `False` and one `True` conditions merged by the logical operator `and` is always `False`, regardless of the order of the conditions.

- Finally, what happens if we change both conditions to be `False`?

[9]:
```
1 number = 3
2 print(number > 4)
3 print(number < 2)
4 print(number > 4 and number < 2)
```
```
False
False
False
```
```
number is assigned three
print number greater than four
print number less than two
print number greater than four and number less
than two
```

Both conditions are `False` because `4` is neither larger than `4` (line 2) nor smaller than `2` (line 3). The combination of the two conditions is `False` too (line 4). This is what Python "sees":

```
4 print(False and False):
```

```
print False and False
```

```
False
```

We can summarize the outcome of combinations of conditions using the logical operators `and` in a **truth table**:

	First condition	Second condition	First condition **and** Second condition
(1)	`True`	`True`	`True`
(2)	`False`	`True`	`False`
(3)	`True`	`False`	`False`
(4)	`False`	`False`	`False`

Row 1 corresponds to the example we saw in cell 6, where both conditions were `True`, and their combination was also `True`. We can pronounce the first row as *True and True gives True*. Row 2, where *True and False gives False*, corresponds to the example in cell 7. Row 3—*False and True gives False*—corresponds to the example at cell 8. Finally, row 4 corresponds to the example in cell 9, where *False and False gives False*. When you write code that combines conditions using `and`, you can use this table as a reference to determine the outcome!

What happens when we combine conditions using the logical operator `or`? Here is the **truth table for `or`**:

	First condition	Second condition	First condition **or** Second condition
(1)	`True`	`True`	`True`
(2)	`False`	`True`	`True`
(3)	`True`	`False`	`True`
(4)	`False`	`False`	`False`

For the logical operator `or`, *True and True gives True* (row 1), *False and True gives True* (row 2), *True and False gives True* (row 3), and *False and False gives False* (row 4).

What are the similarities and differences between the `and` and `or` truth tables? The columns for the first and second conditions are the same for both tables, but the results change. For **`and`**, the result is **`True` only when both conditions are `True`**, and it is `False` in all other cases. Conversely, for **`or`**, the result is **`False` only when both conditions are `False`**, and it is `True` for all other cases. A side note: In other textbooks or on the Internet, you might find that the columns of the first and second condition are inverted. But the results remain the same!

Let's conclude with the **truth table for the logical operator `not`**. Here it is:

	Condition	**`not`** condition
(1)	`True`	`False`
(2)	`False`	`True`

`not` inverts conditions. When we write `not` in front of a `True` condition, it becomes `False` (row 1). Conversely, when we write `not` in front of a `False` condition, it becomes `True` (row 2).

✎ Create your examples

In a notebook, write an example for each row of the `or` truth table and of the `not` truth table, similar to what we did above for `and`.

3. Where else do we use Booleans?

Booleans are often used as **flags** in `while` loops. What does this mean?

- Look at this modified version of the example *Do you want more candies?* from Chapter 17:

[10]:

```
# initialize variable
number_of_candies = 0

# use a Boolean as a flag
flag = True

# print the initial number of candies
print("You have " + str(number_of_candies) + " candies")


# as long as the flag is True
while flag == True:

    # ask if they want a candy
    answer = input("Do you want a candy? (yes/no)")

    # if the answer is yes
    if answer == "yes":

        # add a candy
        number_of_candies += 1

        # print the total number of candies
        print("You have " + str(number_of_candies) + " candies")


    # if the answer is not yes
    else:

        # print the final number of candies
        print("You have a total of " + str(number_of_candies) + " candies")


        # stop the loop by assigning False to the flag
        flag = False
```

```
initialize variable
number of candies is assigned zero

use a Boolean as a flag
flag is assigned True

print the initial number of candies
print You have concatenated with str number of candies concatenated with candies

as long as the flag is True
while flag equals True

ask if they want a candy
answer is assigned input Do you want a candy? (yes/no)

if the answer is yes
if answer equals yes

add a candy
number of candies increased by one

print the total number of candies
print You have concatenated with str number of candies concatenated with candies

if the answer is not yes
else

print the final number of candies
print You have a total of concatenated with str number of candies concatenated with candies

stop the loop by assigning False to the flag
flag is assigned False
```

✎ Find the differences

Can you identify some differences between the while loop in the example above and the one in Chapter 17?

Recap

- When we write a comparison or a condition, the outcome is a Boolean variable.
- Booleans are a Python type, like lists, strings, integers, etc.
- There are only 2 Boolean values: `True` and `False`.
- Combinations of conditions using `and`, `or`, and `not` follow the truth tables.
- Booleans can be used as flags in `while` loops (they act like traffic lights).

What is the difference between GeeksforGeeks and Stack Overflow?

There are several online resources for coding. What are the differences among them? How do we choose which to use? In a simple manner, we can categorize websites into two groups: tutorial websites and question and answer (Q&A) websites. In **tutorial websites**, each page contains clear and extensive explanations about a specific topic. Common website tutorials are GeeksforGeeks (www.geeksforgeeks.org), W3Schools (www.w3schools.com), or learnpython.org (www.learnpython.org). The last two also offer the possibility of typing code directly in their webpages so that you can immediately test what you learn. On the other hand, in **Q&A website**, each page starts with a question by a user, followed by answers by other users. Usually, questions are about solving bugs or looking for better code implementations. Examples include Stack Overflow (www.stackoverflow.com) or Reddit (www.reddit.com). Q&A websites are extremely useful for coders. We all encounter issues that we don't know how to solve. The great news is that there is always somebody else who had the same issue before us and shared the solution!

Let's code!

1. *Do you want less exercises?* Rewrite the `while` loop from coding exercise 1.b *Do you want less exercises?* in Chapter 17 using a Boolean as a flag in the header.
2. *Flipping coins!* When flipping a coin, we have two outcomes: heads and tails. In this exercise, we will use `True` for heads and `False` for tails. Flip a coin 8 times and save the outcomes in a list whose elements are of type Boolean. How many outcomes of heads and tails did you get? What is the ratio between the number of heads and tails? Now flip a coin 1000 times. What is the new ratio? How do the two ratios differ?
3. *Comparator*. A comparator is an algorithm that compares two numbers. It is similar to a calculator, but instead of using arithmetic operators, it uses comparison operators. Create a comparator that asks a user for two integers and prints all the possible comparisons between the two integers.
   Example: If the user enters 3 and 5, then print:
   3 > 5 is False
   3 < 5 is True
   etc.
   Make sure to: (1) use all the comparison operators, (2) use Booleans wherever possible, and (3) allow the user to use the comparator for as long as they want. Which numbers did you use to test that the comparator works correctly? When do you get `True` as an output?

PART 6

OVERVIEW OF LISTS AND THE FOR LOOP

In this part, you will integrate your existing knowledge of lists and `for` loops with new concepts and properties. At the end of part 6, you will have fully mastered lists and loops!

21. Overview of lists

Operations, methods, and tricks

We are halfway through our journey of learning computational thinking and coding in Python! Thus, this is a good moment to take a break and summarize everything we have learned about lists so far. In this chapter, we will re-organize some of the concepts, operations, and methods that you have already learned in a clearer, more structured way, and we will introduce some new ones. Consider this chapter as a comprehensive guide to lists—the one that you can return to whenever you have questions or doubts. Let's start! Follow along with Notebook 21.

1. Arithmetic operations on list elements

In lists, arithmetic operations are performed **element-wise**—that is, **on each element individually**—using **`for` loops**. Element-wise operations can be done between two or more lists of the same length or between a list and a number. Let's look at an example for each case using addition—the same rules apply for any of the seven arithmetic operations.

1.1 Element-wise arithmetic operations between two or more lists of the same length

- Sum two lists element-wise:

[1]:

```
1 odd_numbers = [1, 3, 5]                    odd numbers is assigned one, three, five
2 even_numbers = [2, 4, 6]                   even numbers is assigned two, four, six
3 summed = []                                summed is assigned empty list
4
5 for i in range(len(odd_numbers)):          for i in range len odd numbers
6     summed.append(odd_numbers[i] +         summed dot append odd numbers in position i
                    even_numbers[i])         plus even numbers in position i
7
8 print(summed)                              print summed
```

```
[3, 7, 11]
```

We start with `odd_numbers` and `even_numbers`, two lists containing 3 integers each (lines 1 and 2), and `summed`, which is initialize as an empty list and will contain the result (line 3). Then, we create a `for` loop that spans the indices of one of the two lists (line 5)—either list can be used, as they have the same length. In the loop, we append the sum of the current element in `odd_numbers` to the element in the same position in `even_numbers` to `summed` (line 6). Finally, we print the result for a check (line 8). In this case, we save the result in a third list (`summed`), but we could also overwritten one of the existing lists (e.g., `odd_numbers[i] = odd_numbers[i] + even_numbers[i]`), although in this case the name `odd_numbers` would no longer accurately describe the list content.

1.2 Element-wise arithmetic operations between a list and a number

- Sum a number to each element of a list:

[2]:

```
1 numbers = [1, 2, 3]                        numbers is assigned one, two, three
2 number = 3                                 number is assigned three
3                                                                          ↳
```

```
4  for i in range(len(numbers)):          for i in range len numbers
5      numbers[i] += number               numbers in position i increased by number
6
7  print(numbers)                         print numbers
[4, 5, 6]
```

We create the list `numbers` containing three integers (line 1) and the variable `number` to which we assign the number `3` (line 2). Then, we use a `for` loop to browse all the positions of the list elements (line 4), and we increase each element by the value of `number` (line 5). Finally, we print the result (line 7). Similar to the previous example, we can either overwrite the existing list (as we do in this example) or we can create an empty list before the `for` loop (e.g., `summed = []`) and fill it in the loop (e.g., `summed.append(numbers[i] + number)`).

2. "Arithmetic" operations between lists

The operations between lists are not actually arithmetic, even though they use arithmetic symbols with different meanings. The two possible operations between lists are concatenation and replication.

2.1 List concatenation

As you know, in concatenation, we use the symbol `+`, pronounced *concatenated with*, to merge two (or more) lists together. Here is a quick review example.

- Concatenate two lists:

```
[3]: 1  odd_numbers = [1, 3, 5]          odd numbers is assigned one, three, five
     2  even_numbers = [2, 4, 6]         even numbers is assigned two, four, six
     3  concatenated = odd_numbers + even_numbers   concatenated is assigned odd numbers
                                         concatenated with even numbers
     4  print(concatenated)              print concatenated
     [1, 3, 5, 2, 4, 6]
```

We create two lists, `odd_numbers` (line 1) and `even_numbers` (line 2), containing odd and even numbers, respectively. Then, we concatenate them using the concatenation operator `+`, and we store the result in a new list called `concatenated` (line 3). When concatenating lists, we can also overwrite one of the two original lists instead of creating a new variable, as in `odd_numbers = odd_numbers + even_numbers`—although in this case, the name `odd_numbers` would no longer be appropriate for the resulting list. Finally, we print the result (line 4), which is a list containing the elements of `odd_numbers` and `even_numbers` merged together.

2.2 List replication

In replication, we use the symbol `*`, pronounced *replicated by*, to replicate a list a specified number of times. Let's look at the following example.

- Replicate a list 3 times:

```
[4]: 1  numbers = [1, 2, 3]              numbers is assigned one,two,three
     2  number = 3                       number is assigned three
     3  replicated = numbers * number    replicated is assigned numbers replicated
                                         by number
     4  print(replicated)                print replicated
     [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

We create a list called `numbers` (line 1) and an integer variable called `number` (line 2). Then, we replicate the list `numbers` by the number of times indicated by the variable `number` using the symbol `*`, and we save the result in a new list called `replicated` (line 3). Once more, instead of creating a new variable, we could overwrite the existing list: `numbers = numbers * number`. Finally, we print `replicated` (line 4). As you can see in the print, `replicated` contains the list `numbers` repeated three times. You might wonder: When is replication useful? For example, when we have to **initialize a long list** of zeros! Look at the following example:

```
[5]:
1  short_list = [0]                          short list is assigned zero
2  number = 50                               number is assigned fifty
3  long_list = short_list * number           long list is assigned short list
                                             replicated by number
4  print(long_list)                          print long list
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

We initialize `short_list` as a list containing one zero (line 1) and the variable `number` containing the integer `50` (line 2). Then, we replicate `short_list` by the number of times indicated by `number`, and we store the result in the variable `long_list` (line 3). Finally, we print `long_list` (line 4). As you can see, we obtained a list containing 50 zeros. If we had created `long_list` manually, it would have been very tedious, and we could have easily miscounted the number of zeros in the list!

3. List assignment

We can **assign a list to another list** but we have to be very careful! Let's see why.

- Given a list containing a few integers:

```
[6]:
1  given_list = [1, 2, 3]                   given list is assigned one, two, three
2  print(given_list)                         print given list
[1, 2, 3]
```

We create a list called `given_list` containing some integers (line 1) and we print it (line 2).

- Assign `given_list` to `new_list`:

```
[7]:
1  new_list = given_list                     new list is assigned given list
2  print(new_list)                           print new list
[1, 2, 3]
```

We assign `given_list` to another list called `new_list` (line 1), and we print it (line 2). As we can see, `new_list` contains the same elements as `given_list`, as expected. Let's go one step further!

- Change the first list element of `new_list`:

```
[8]:
1  new_list[0] = 40                          new list in position zero is assigned
                                             forty
2  print(new_list)                           print new list
[40, 2, 3]
```

We change the first element of `new_list` to `40` (line 1) and we print `new_list` after the change (line 2). As expected, the first element is now `40`. What about `given_list`?

- Print `given_list`:

```
[9]:  1  print(given_list)                    print given list
      [40, 2, 3]
```

The first element of `given_list` is also `40`! This happens because when we assign a list to another, we give two names to the same list. It is a bit like when a person has two names: for example, my brother's name is Flavio Alberto. Whether I call him Flavio or Alberto, he is always the same person!

- How can we create an independent copy of a list?

```
[10]: 1  given_list = [1, 2, 3]              given list is assigned one, two, three
      2  new_list = given_list.copy()        new list is assigned given list dot copy
      3  new_list[0] = 40                    new list in position zero is assigned
                                             forty
      4  print(given_list)                   print given list
      5  print(new_list)                     print new list
      [1, 2, 3]
      [40, 2, 3]
```

As we did in cell 6, we create the list `given_list` that contains a few numbers (line 1). Then, instead of assigning `given_list` to `new_list` (as we did in cell 7), we use the method **`.copy()`**, which **creates an independent copy of a list** (line 2). Continuing the brother analogy, it is as if we created a twin who is similar but independent. Since we have two separate lists, any change made to one list will not affect the other. In our code, we change only the first element of `new_list` to `40` (line 3)—like we did in cell 8—and when we print both lists (lines 4 and 5), we see that only `new_list` was affected.

4. Adding an element or a list to a list

In lists, we can either add elements to a list or we can add a list to another list.

As you already know, we can add an element to a list in two ways: either at the end using the method `.append()` (see Chapter 4), or at a specific position using the method `.insert()` (see Chapter 5). Let's quickly review these two cases for completeness.

4.1 Adding an element at the end of a list using the method `.append()`

- Add one element at the end of a list:

```
[11]: 1  numbers = [1, 2, 3]                 numbers is assigned one, two, three
      2  numbers.append(4)                   numbers dot append four
      3  print(numbers)                      print numbers
      [1, 2, 3, 4]
```

We create the list `numbers` containing three integers (line 1), and we add the number `4` using the method **`.append()`** (line 2). Then, we print `number` to check the result (line 3).

4.2 Adding an element at a specific position in a list using the method `.insert()`

- Insert the number 2 in position 1:

```
[12]: 1  numbers = [1, 3, 4]                 numbers is assigned one, three, four
      2  numbers.insert(1, 2)                numbers dot insert at position one, two
      3  print(numbers)                      print numbers
      [1, 2, 3, 4]
```

We initialize a list containing the integers 1, 3, and 4 (line 1). At position 1, we insert the number 2 using the method **`.insert()`**, which takes as arguments first the position and then the value of the new element (line 2). Finally, we print `numbers` (line 3).

We can add **a list at the end of another list** using two ways: concatenation (as we saw in cell 3 and refresh in the example below) and the method `.extend()`.

4.3 Adding a list to another list using concatenation

- Concatenate two lists:

[13]:
```
1  first_list = [1, 2, 3]
2  second_list = [4, 5, 6]
3  third_list = first_list + second_list

4  print(third_list)
```
```
[1, 2, 3, 4, 5, 6]
```
```
first list is assigned one, two, three
second list is assigned four, five, six
third list is assigned first list
concatenated with second list
print third list
```

We create two lists, called `first_list` and `second_list`, to which we assign some integers (lines 1 and 2). Then, we concatenate the two lists to obtain `third_list` (line 3). Finally, we print `third_list` (line 4).

4.4 Adding a list to another list using the method `.extend()`

- Add one list at the end of another list:

[14]:
```
1  first_list = [1, 2, 3]
2  second_list = [4, 5, 6]
3  first_list.extend(second_list)
4  print(first_list)
```
```
[1, 2, 3, 4, 5, 6]
```
```
first list is assigned one, two, three
second list is assigned four, five, six
first list dot extend second list
print first list
```

We create again the two lists `first_list` and `second_list` (lines 1 and 2), but this time we use the method **`.extend()`** to merge them. The syntax for `.extend()` is (1) the list to which we want to add another list, (2) dot, and (3) the added list in between round brackets (line 3). Finally, we print the merged list (line 4).

What are the **differences** between concatenation and `.extend()`? When using **concatenation**, we can either create a **new list** (e.g., `third_list = first_list + second_list`) or **add a list to an existing list** (e.g., `first_list = first_list + second_list`). In addition, we can add a list at any position: at the beginning, (e.g. `first_list = second_list + first_list`), at the end (e.g. `first_list = first_list + second_list`), or—when concatenation is combined with slicing— in the middle of another list (e.g. `first_list = first_list[:2] + second_list + first_list[2:]`). In contrast, when using **`.extend()`**, we can only **modify the list** to which the method is applied (i.e., `first_list` in cell 14) and we can add a list **only at the end** of another list.

5. Removing elements from a list

As you know, we can remove list elements either based on their value, using `.remove()` (see Chapter 4) or on their position, using `.pop()` (see Chapter 5). We can also remove all elements using `.clear()`. Let's see some example to refresh these methods and learn some new tricks.

5.1 Removing an element from a list based on its value using the method `.remove()`

- From the following list, remove all the elements `"ciao"`:

```
[15]: 1 greetings = ["ciao","ciao","hello"]
      2 greetings.remove("ciao")
      3 print(greetings)
      ['ciao', 'hello']
```

```
greetings is assigned ciao, ciao, hello
greetings dot remove ciao
print greetings
```

We start with a list containing three strings, where the element `"ciao"` is present twice (line 1). Then, we use the method `.remove()`, to eliminate `"ciao"` (line 2). Finally, we print `greetings` (line 3). Only one `"ciao"` (the first one) was removed! This is because in lists containing multiple similar elements, the method **`.remove()` deletes only the first element**. How do we remove both `"ciao"` from the list `greetings`? The first instinctive idea might be to use a `for` loop that goes through all element positions and removes the unwanted elements based on a certain condition (in this case, remove the element if it is equal to `"ciao"`). However, this solution does not work for the reasons explained in the *In more depth* section at the end of this chapter. What we need here is a **`while` loop**:

```
[16]: 1 greetings = ["ciao", "ciao", "hello"]
      2 while "ciao" in greetings:
      3     greetings.remove("ciao")
      4 print(greetings)
      ['hello']
```

```
greetings is assigned ciao, ciao, hello
while ciao in greetings
greetings dot remove ciao
print greetings
```

We start with the list `greetings` (line 1), then we create a `while` loop where as long as the string `"ciao"` is in `greetings` (line 2), we remove it using the method `.remove()` (line 3). Finally, we print the result (line 4).

5.2 Removing an element from a list based on its position using the method `.pop()`

- Remove the string `"hello"` based on its position:

```
[17]: 1 greetings = ["ciao", "ciao", "hello"]
      2 greetings.pop(2)
      3 print(greetings)
      ['ciao', 'ciao']
```

```
greetings is assigned ciao, ciao, hello
greetings dot pop two
print greetings
```

We rewrite the list `greetings` (line 1) and we use the method **`.pop()`**, whose argument is the position of the element to delete. Finally, we print to check the result (line 3).

Another way to remove one or more elements in a list is by using **list comprehension**. We will see it in the next chapter.

5.3 Removing all elements from a list using the method `.clear()`

- Remove all elements in a list:

```
[18]: 1 greetings = ["ciao", "ciao", "hello"]
      2 greetings.clear()
      3 print(greetings)
      []
```

```
greetings is assigned ciao, ciao, hello
greetings dot clear
print greetings
```

Once more, we rewrite the list `greetings` (line 1). Then, we use the method **`.clear()`** (line 2) to remove all elements. When we print (line 3), we see that `greetings` is now an **empty list**.

6. Sorting a list

Sorting lists is a very common task in coding. For example, we might want to sort a list of prices increasingly (or decreasingly) or a list of names in alphabetical order (see the exercise A *further step*! below). In the three examples below (cells 19, 20, and 21), we will create a list of integers called `numbers` (line 1), use three different ways to sort it (line 2), and print the outcome (line 3).

6.1 Sorting list elements in ascending order using the method `.sort()`

- Sort the following list of integers:

[19]:
```
1 numbers = [5, 7, 6]
2 numbers.sort()
3 print(numbers)
```
```
[5, 6, 7]
```
```
numbers is assigned five, seven, six
numbers dot sort
print numbers
```

To sort the list `number`, we use the method **`.sort()`** (line 2). As you can see from the print, the numbers are sorted in an increasing (or ascending) way, that is from the smallest to the greatest. What if we want to sort the numbers in a decreasing (or descending) way? The answer is in the next example.

6.2 Sorting list elements in descending order using the method `.sort()`

- Sort the following list of integers in a descending way:

[20]:
```
1 numbers = [5, 7, 6]
2 numbers.sort(reverse = True)
3 print(numbers)
```
```
[7, 6, 5]
```
```
numbers is assigned five, seven, six
numbers dot sort reverse is assigned True
print numbers
```

We use `.sort()` as we did in the example above, but we add the **argument `reverse`**, to which we assign the Boolean `True`—you will learn more about method (or function) arguments starting from Chapter 28. As you can see from the print, the list is now sorted in a descending way: that is, from the greatest to the smallest number.

6.3 Reversing the elements in a list using the method `.reverse()`

- Reverse the following list of integers:

[21]:
```
1 numbers = [5, 7, 6]
2 numbers.reverse()
3 print(numbers)
```
```
[6, 7, 5]
```
```
numbers is assigned five, seven, six
numbers dot reverse
print numbers
```

We use the method **`.reverse()`** to **invert the order of the elements in the list**. Thus, the last will become the first, the second to last element will become the second, etc.

Note that **`.sort()`** sorts a list elements based on their **value**, whereas **`.reverse()`** sorts a list elements based on their **position**.

7. Searching and counting elements

Let's conclude our long journey through list operations and methods by learning how to search and count elements in a list.

7.1 Searching an element in a list using the method `.index()`

- Create a list and search for a specific element:

```
[22]: 1 letters = ["a", "g", "c", "g"]
      2 position = letters.index("g")
      3 print(position)
      1
```

```
letters is assigned a, g, c, g
position is assigned letters dot index g
print position
```

We create the list `letters` containing strings (line 1), and we **look for the position of the element** `"g"` by using the method **`.index()`** (line 2). Then, we print the results (line 3). As you can see, `.index()` gives us only the **position of the first element**, which is `1`, because element positions start from `0`.

- How do we find all positions?

```
[23]: 1 letters = ["a", "g", "c", "g"]
      2 positions = []
      3 for i in range(len(letters)):
      4     if letters[i] == "g":
      5         positions.append(i)
      6 print(positions)
      [1, 3]
```

```
letters is assigned a, g, c, g
positions is assigned empty list
for i in range len letters
if letters in position i equals g
positions dot append i
print positions
```

To find **all positions of an element** in a list, we can use the **`for` loop**! We create the list `letters` (line 1) and the empty list `positions` that will contain the indices corresponding to the letter `"g"` (line 2). Then, we create a `for` loop that browses all the positions of the letters (line 3), and if the current letter is equal to `"g"` (line 4), then we append its position (that is, `i`) to the list `positions` (line 5). Finally, we print the result (line 6).

7.2 Counting how many times an element appears in a list using the method `.count()`

- Count how many times an element is present in a list:

```
[24]: 1 letters = ["a", "g", "c", "g"]
      2 n = letters.count("g")
      3 print(n)
      2
```

```
letters is assigned a, g, c, g
n is assigned letters dot count g
print n
```

We start with the same list `letters` as in the example above (line 1). Then, we use the method **`.count()`**, which takes the **element we want to count**—that is, `"g"`—as an **argument** and returns how often it appears (line 2). Finally, we print the result (line 3).

In this chapter, you have refreshed and learned how to execute all the typical operations that we perform on lists by using list methods and various operators. At this point, you can consider yourself an expert in lists! Congratulations!

✎ A further step!

Answer the following questions to discover more tricks about lists!

1. How can we efficiently remove the elements of a list in even positions?

2. What is the difference between the method `.clear()` and the keyword **`del`**?

3. What is the output of the method `.sort()` for a list of strings? E.g.: `sweets = ["chocolate", "icecream", "candy", "cake"]`

4. What is the output of the method `.sort()` for a list of strings and numbers? E.g.: `sweets_numbers = ["chocolate", 43, "icecream", "candy", "cake", 18]`

✎ Complete the table

In this chapter, you refreshed or learned the 11 list methods. Fill out the table below with method definitions and alternative ways to implement the same operations. Some alternatives are presented in this chapter or in previous chapters, but for others, you will have to come up with new ideas (feel free to consult the internet!).

Method	What it does	Alternative
`.append()`		
`.clear()`		
`.copy()`		
`.count()`		
`.extend()`		
`.index()`		
`.insert()`		
`.pop()`		
`.remove()`		
`.reverse()`		
`.sort()`		

Recap

- We can perform element-wise operations in lists using the arithmetic operators `+`,`-`, `*`, `/`, `**`, `//`, `%`.
- We can perform "arithmetic" operations on lists using concatenation `+` and replication `*`.
- The 11 list methods are: `.append()`, `.clear()`, `.copy()`, `.count()`, `.extend()`, `.index()`, `.insert()`, `.pop()`, `.remove()`, `.reverse()`, `.sort()`.
- Of the 11 methods, the 3 methods that return a value are `.copy()`, `.count()`, and `.index()`. The other 8 methods modify the lists themselves.

Why not use a for loop to remove list elements?

A `for` loop is not the right way to remove elements in a list for at least two reasons. Let's discover them with the help of the following example:

[1]:

```
1  greetings = ["ciao", "ciao", "hello"]
2  for i in range(len(greetings)):
3      print("---------------")
4      print("i == " + str(i))
5      print("before the if:")
6      print(greetings)
7      if greetings[i] == "ciao":
8          del greetings[i]
9      print("after the if:")
10     print(greetings)
```

```
greetings is assigned ciao, ciao, hello
for i in range len greetings
print dashes
print i equals concatenated with str i
print before the if:
print greetings
if greetings in position i equals ciao
del greetings in position i
print after the if:
print greetings
```

```
---------------
i == 0
before the if:
['ciao', 'ciao', 'hello']
after the if:
['ciao', 'hello']
---------------
i == 1
before the if:
['ciao', 'hello']
after the if:
['ciao', 'hello']
---------------
i == 2
before the if:
['ciao', 'hello']
```

(a) (b) (c) (d) (e) (f) (g) (h) (i) (j) (k) (l) (m) (n) (o)

```
---------------------------------------
IndexError                Traceback (most recent call last)
Cell In[16], line 6
       5 print("before the if:")
       6 print("greetings")
----> 7 if greetings[i] == "ciao":
       8     del greetings[i]
       9 print("after the if:")
IndexError: list index out of range
```

↳

We start with the list `greetings` that we created in cell 15 (line 1). Then, we create a `for` loop that browses all the positions in the list (line 2). In the `for` loop, we use an `if` condition to check whether the current element is equal to the element to remove (line 7). If that is the case, then we remove the current element using the keyword `del` (line 8), which we learned in Chapter 7. In between the main commands, we print some messages to check the list changes at each iteration: a graphic separator for each loop (line 3), the number of the current iteration (line 4), a message specifying whether the following prints are before the `if` (line 5) or after the `if` (line 9), and the list before deletion (line 6) and after deletion (line 10). Note that for clarity of the following explanation, the printed lines are identified with letters, which are not printed when running the code.

Let's see how the list evolves:

- First loop (`i` is `0`): before the `if` condition, the list is complete `["ciao", "ciao", "hello"]` (print (c)). After the `if`, `greetings` contains only `["ciao", "hello"]` (print (e)) because the first element `"ciao"` was deleted (line 8). This deletion affects both the positions of the remaining elements in the list and the length of the list, as you can see in Figure 21.1. First, the yellow string `"ciao"` in position `0` before the is `if` is no more present after the `if`. Second, the element indices reset from `0` and thus the positions of the remaining elements changes: the green `"ciao"` moves from position `1` to position `0`, and `"hello"` moves from position `2` to `1`. And third, the number of elements in the list changes from 3 to 2. These changes will affect the second and third loops.

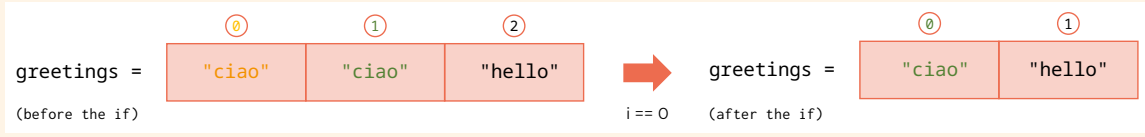


Figure 21.1. Change of list content, element positions, and list length after deletion of a list element.

- Second loop (`i` is `1`): before the `if`, the list is the same as at the end of the previous loop, that is, `["ciao", "hello"]` (print (i)). After the `if`, the list remains unchanged (print (k)) because the current element `greetings[1]`, which is now `"hello"`, does not satisfy the `if` condition. Did you notice that the green string `"ciao"` in position `0` was not deleted? This is because after the index shift in the previous loop, `"ciao"` moved to position `0`. As a result, its deletion was skipped!
- Third loop (`i` is `2`): before the `if`, the list is still `["ciao", "hello"]` (print (o)). Then, we encounter an index error at line 6, where the `if` conditions is. Do you know why? Because the range specified in the `for` loop header does not adapt to changes in the list length. Now, `i` is `2`, but `greetings[2]` no longer exists.

In conclusion, by using a `for` loop to delete an element in a list, we cause two errors: (1) we **skip list elements** that should be deleted due to the index shift and (2) we encounter an **out of range error** because the list has been shortened after the element deletion.

Let's code!

1. *Selling veggies at the market.* At your stand at the market, you started the day with the following items:

Item	N. of items	Price per item
carrots	10	0.7
zucchini	12	0.5
potatoes	11	0.2

   a. Create three lists: one for the items, one for the number of items, and one for their prices.
   b. Today you got 3 customers. You want to keep track of how much money each customer spent and how much produce they bought. Create and initialize a list called `total`, where each element corresponds to the amount spent by a customer (how long is the list? what is its content?).
   c. The first customer bought 2 carrots, 4 zucchini, and 3 potatoes. Create a list where each element is the number of bought items (i.e., the list will contain 3 elements, corresponding to number of carrots, zucchini, and potatoes, respectively).
   d. How much did the customer pay? Save the amount in the first position of the list `total` without creating an intermediate variable (*Hint*: if you don't know how to do it, first solve the task by using an intermediate variable, and then find a way to remove it).
   e. The second customer got 3 carrots and 3 potatoes. Create the corresponding item list. How much did the customer pay? Save the amount in the second position of the list `total`.
   f. The third customer wanted 6 carrots, 4 zucchini, and 1 potato. Create the corresponding item list.
   g. Did you have enough items to sell? Compute it.
   h. Given that the third customer is going to buy whatever is left (e.g., if they wanted 6 carrots, but only 2 were left, they bought 2), how do you modify their item list? Use an `if` construct.
   i. How much did the third customer pay? Save the amount in the third position of the list `total`.
   j. What was the average amount a customer spent at your stand?
   k. What was your most popular item today? And the one you sold the least of? Compute them!
2. *New year's countdown!* Given the following list: `numbers = [0,1,2,3,4,5,6,7,8,9]`, reverse it using:
   a. A list method.
   b. Slicing.
   c. A `for` loop.

   What are the differences among the three methods?
3. App store. You are running a market study on app store data. These are the prices of the apps in the store:

```
app_prices = [
7.99, 7.99, 2.99, 4.99, 7.99, 9.99, 9.99, 1.99, 1.99, 1.99,
4.99, 5.99, 3.99, 5.99, 0.99, 3.99, 3.99, 2.99, 1.99, 4.99,
8.99, 1.99, 3.99, 1.99, 1.99, 8.99, 6.99, 0.99, 6.99, 8.99,
3.99, 1.99, 0.99, 1.99, 0.99, 8.99, 1.99, 7.99, 3.99, 1.99,
8.99, 2.99, 4.99, 6.99, 4.99, 7.99, 8.99, 1.99, 2.99, 0.99,
7.99, 6.99, 7.99, 6.99, 2.99, 0.99, 0.99, 3.99, 2.99, 5.99,
0.99, 0.99, 7.99, 9.99, 5.99, 5.99, 1.99, 4.99, 5.99, 5.99,
6.99, 9.99, 5.99, 5.99, 1.99, 8.99, 9.99, 4.99, 9.99, 4.99,
0.99, 0.99, 2.99, 9.99, 3.99, 6.99, 8.99, 4.99, 1.99, 9.99,
0.99, 7.99, 1.99, 4.99, 4.99, 0.99, 3.99, 3.99, 1.99, 8.99,
3.99, 9.99, 5.99, 2.99, 2.99, 2.99, 5.99, 4.99, 3.99, 8.99,
5.99, 8.99, 8.99, 1.99, 9.99, 7.99, 6.99, 7.99, 4.99, 4.99,
7.99, 8.99, 7.99, 4.99, 5.99, 5.99, 0.99, 2.99, 8.99, 7.99,
1.99, 3.99, 3.99, 4.99, 9.99, 0.99, 1.99, 3.99, 9.99, 5.99,
4.99, 8.99, 6.99, 5.99, 6.99, 7.99, 1.99, 2.99, 9.99, 6.99,
```

```
9.99, 6.99, 8.99, 8.99, 2.99, 1.99, 9.99, 1.99, 7.99, 9.99,
4.99, 3.99, 9.99, 9.99, 6.99, 6.99, 7.99, 9.99, 2.99, 4.99]
```

a. How many apps are there?
b. How many apps cost 4.99? Calculate the result in two ways, once using a list method, and once using a `for` loop.
c. What is the percentage of apps that cost 4.99?
d. What are the unique prices of the apps in the store? Find them and sort them in ascending order.
e. How many apps are there for each price?
f. What is the most popular price for an app?

22. Overview of the for loop

Various ways of repeating commands on lists and beyond

You already know that the `for` loop is a construct used to repeat a group of commands a determined number of times (Chapter 8). You also know how to use the for loop in combination with lists—that is, to browse lists (Chapters 8 and 9), search for elements in lists (Chapter 10), change list elements (Chapter 11), and create lists by adding one element at a time (Chapter 12). In this chapter, we will first briefly review what you already know, for the sake of completeness. Then, we will explore new types and syntaxes of `for` loops that we can use with lists, each of them with their own characteristics and usage. As in the previous chapter, consider this chapter a comprehensive guide that you can return to whenever you have questions or doubts about the `for` loop. Ready? Follow along with Notebook 22!

1. Repeating commands

Let's quickly refresh how to use the `for` in its basic form, that is, to repeat commands.

- Print 3 random numbers between 1 and 10:

[1]:
```
1 import random
2
3 for _ in range(3):
4     number = random.randint(1, 10)
5     print(number)
6
4
3
```

```
import random

for underscore in range three
number is assigned random dot randint one ten
print number
```

We import the package `random` (line 1). Then, we implement the `for` loop (lines 3–5). We start with the header, which contains: (1) the keyword `for`, (2) a variable for the index, (3) the membership operator `in`, and (4) the built-in function `range()` (line 3). In this case, we use an underscore as a variable for the index because we do not need the index in the loop body. We will review the characteristics of the built-in function `range()` in the next paragraph. In the body of the `for` loop—which is always indented with respect to the header—we create a random number between 1 and 10 using the function `.randint()` from the package `random` (line 4), and we print the created number (line 5). The lines of code in the loop body are repeated at each loop or iteration—in this case, three times, as indicated by `range(3)`.

2. For loop with lists

There are at least 4 syntaxes for the `for` loop with lists, each with its specific functionality: `for` loop through indices, `for` loop through elements, `for` loop through indices and elements, and list comprehension. The `for` loop you already know is the `for` loop through indices, which we will briefly summarize in the next paragraph. Then, you will learn the other three types of `for` loop. Note that "through indices", "through elements", and "through indices and elements" are not technical terms; however, we will use them for convenience. On the contrary, "list comprehension" is a technical term that you can find in any Python book or coding website.

To explain and compare the four types of `for` loop, we will start with the following list, which contains three strings:

```
1  last_names = ["garcia", "smith", "zhang"]   last names is assigned garcia, smith, zhang
```

Our task will be to change the first letter of each string to upper case. For that, we will apply the method `.title()` to each list element, and we will overwrite the existing list whenever possible.

2.1 For loop through indices

- Capitalize each string using a `for` loop through indices:

```
[2]:
1  last_names = ["garcia", "smith", "zhang"]                    last names is assigned garcia,
                                                                smith, zhang
2
3  for i in range(len(last_names)):                             for i in range len last names
4      print("The element in position " + str(i) +             print The element in position
       " is: " + last_names[i])                                 concatenated with str i
                                                                concatenated with is concatenated
                                                                with last names in position i
5      last_names[i] = last_names[i].title()                    last names in position i is
                                                                assigned last names in position
                                                                i dot title
6
7  print(last_names)                                            print last names
```

```
The element in position 0 is: garcia
The element in position 1 is: smith
The element in position 2 is: zhang
['Garcia', 'Smith', 'Zhang']
```

We start with the list to modify (line 1). Then, we write the `for` loop header, which is composed of: (1) the keyword `for`, (2) the index variable `i`, (3) the membership operator `in`, and (4) the built-in function `range()` (line 3). **`range()`** can have three parameters: start, which we omit when it is `0`—like in this case; stop, which usually coincides with the length of the list; and step, which we omit when it is `1`—like in this example. If we need to browse only the first half of the list, we can write `range(len(last_names)//2)`, or if we want to browse only every second position of the list, we can write `range(0, len(last_names),2)`. Also, let's not forget that `range()` is a built-in function that can be used independently from a `for` loop **to create a range of integers**: for example, `list(range(4))` returns the list `[0,1,2,3]` and `list(range(0,4,2))` returns `[0,2]`. Why do we use `list()` combined with `range()` when creating a list? Because the **built-in function `list()` converts** the output of `range()`—which is its own data type—**to a list**. In the `for` loop body, we print the current value of the index `i` and the corresponding element `last_names[i]`, extracted by slicing (line 4). Then, we change the current element `last_names[i]` by applying the string method `.title()` and reassigning the result to `last_names[i]` itself (line 5). Finally, we print `last_names` to check the modified list (line 7).

2.1 For loop through elements

Let's learn the first new way of implementing the `for` loop: the `for` loop through elements. Read the example below and try to understand what it does:

- Capitalize each string using a `for` loop through elements:

```
[3]:
1  last_names = ["garcia", "smith", "zhang"]
2  last_names_upper = []
3
4  for last_name in last_names:
5      print("The current element is " + last_name)
6      last_names_upper.append(last_name.title())
7
8  print(last_names_upper)
```

```
The current element is: garcia
The current element is: smith
The current element is: zhang
['Garcia', 'Smith', 'Zhang']
```

```
last names is assigned garcia, smith, zhang
last names upper is assigned empty list

for last name in last names
print The current element is concatenated with last name
last names upper dot append last name dot title

print last names upper
```

As in the previous example, we start with the list to modify (line 1). We continue with a new empty list called `last_names_upper` that we will fill within the loop (line 2). Then, we create the `for` loop through elements (lines 4–6). The syntax of the header is: (1) the keyword `for`, (2) a variable, (3) the membership operator `in`, and (4) the list to browse. There are two differences with respect to the `for` loop through indices. First, the **variable in position (2)** is not named `index` or `i`, but it is usually **called with the singular version of the list name**—that is, if the list name is `last_names`, then the variable name is `last_name`; if the list name is `numbers`, then the variable name is `number`; and so on. This is not a rule but a useful convention among Python coders. The second difference is that we directly use the **list itself**—that is, `last_names`—**in position (4)**, instead of `range(len(last_names))`. To understand how the `for` loop through elements works, let's have a look at the loop body. First, we print the current element `last_name` (line 5). As you may notice, there is no slicing (that is, no `[i]`). This is because **the variable in position (2)**—that is, `last_name`—**takes each element of the list one by one in each iteration**. This is the opposite of what happens in a `for` loop through indices, where the variable in position (2)—that is, `i`—takes the values of the list positions without knowing the corresponding elements, and to get an element, we must use slicing (e.g., `last_name[i]`). See a schematic of the difference between the two loops in Figure 22.1.

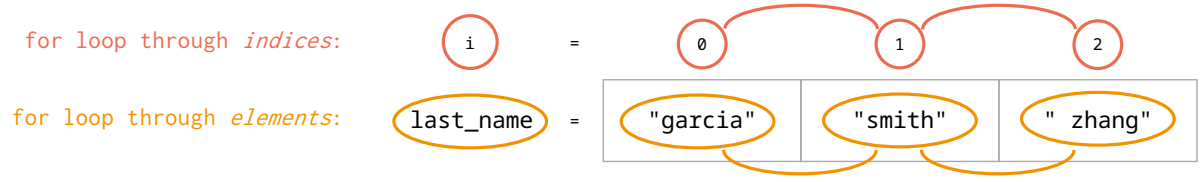


Figure 22.1. Comparison of a `for` loop through indices, where the loop index `i` gets the positions (orange), and a `for` loop through elements, where the loop variable `last_name` gets the elements (yellow).

Therefore, in the first iteration of the example, `last_name` is `"garcia"`; in the second iteration, it is `"smith"`; and in the third iteration, it is `"zhang"`. We conclude by applying the method `.title()` to the string `last_name` and appending the output to `last_names_upper` (line 6). Finally, we print `last_names_upper` (line 8). Why don't we directly modify `last_names`? Because in a `for` loop through elements, **we cannot modify the list** we are browsing. We can only create a new list (that is, `last_name_upper`) to which we append the modified elements (that is, `last_name.title()`). Let's see what happens if we try to use a `for` loop through elements to change elements:

```
1 for last_name in last_names:
2     print("last_name before change: " + last_name)
3     last_name = last_name.title()
4     print("last_name after change: " + last_name)
5 print(last_names)
```

```
for last name in last names
print last name before change: concatenated with last name
last names is assigned last name dot title
print last name after change: concatenated with last name
print last names
```

```
last_name before change: garcia
last_name after change: Garcia
last_name before change: smith
last_name after change: Smith
last_name before change: zhang
last_name after change: Zhang
['garcia', 'smith', 'zhang']
```

In the first iteration, the variable `last_name` is `"garcia"` (line 2), we change it to `"Garcia"` (line 3), and we print it (line 4). In the second iteration, `last_name` is `"smith"` (line 2), we change it to `"Smith"` (line 3), and we print it (line 4). The same procedure happens in the third iteration for `"zhang"`. However, when we print the final list, all strings are still lower case (line 5). This is because any **changes made to `last_name` do not affect the element in the list itself**. In other words, `last_name` takes the list element but cannot overwrite it. Finally, note that because there is no index, in a `for` loop through elements we **cannot keep track of the iteration number**. If we need to know the iteration number, we can either use a `for` loop through indices (Section 2.1) or a `for` loop through indices and elements (Section 2.3).

2.3 For loop through indices and elements

As the name implies, the `for` loop through indices and elements combines a `for` loop through indices with a `for` loop through elements. Its implementation is straightforward. Try to understand the example below before reading the subsequent explanation.

- Capitalize each string using a `for` loop through indices and elements:

[4]:
```
1 last_names = ["garcia", "smith", "zhang"]
2
3 for i, last_name in enumerate(last_names):
4     print("The element in position " + str(i) + " is: " + last_name)
5     last_names[i] = last_name.title()
6
```

```
last names is assigned garcia, smith, zhang

for i last name in enumerate last names
print The element in position concatenated with str i concatenated with is concatenated with last name
last names in position i is assigned last name dot title
```

```
7  print(last_names)                                  print last names
The element in position 0 is: garcia
The element in position 1 is: smith
The element in position 2 is: zhang
['Garcia', 'Smith', 'Zhang']
```

In a `for` loop through indices and elements, the `for` loop header consists of: (1) the keyword `for`, (2) two variables separated by comma, called `i` and `last_name`, (3) the membership operator `in`, and (4) the built-in function `enumerate()` with the list `last_names` as an argument (line 3). The differences with the other `for` loop headers is again in the components (2) and (4). The role of `i` and `last_name` is quite intuitive: `i` is the index that browses all the positions in the list—like in a `for` loop through indices—and `last_name` is the variable that browses all the elements in the list—like in a `for` loop through elements. The values to browse are provided by `enumerate()`, as we can see from the following command (where we use `list()` to convert `enumerate()`'s output into a list for printing):

```
   print(list(enumerate(last_names)))                print list enumerate last names
[(0, 'garcia'), (1, 'smith'), (2, 'zhang')]
```

The **built-in function `enumerate()` provides a list of coupled indices and elements**—that is, `(0, 'garcia')`, `(1, 'smith')`, and `(2, 'zhang')`. Each pair is between round brackets, which indicate a tuple. **Tuples are sequences of elements separated by comma and in between round brackets**. We will talk about tuple characteristics in Chapters 29 and 34. During the `for` loop in this example, the variable `i` is iteratively assigned the first element of each pair—that is, `0`, `1`, and `2`—and the variable `last_name` is assigned the second element of each pair—that is, `'garcia'`, `'smith'`, and `'zhang'`. In the remaining part of the example, first we print the position of each element `i` and its value `last_name` (line 4). Then, we apply the method `.title()` to `last_name`, and we assign the result to the element in the same position `last_names[i]` (line 5). Finally, we print the resulting list (line 7). The `for` loop through indices and elements is useful when we need to **extract both positions and elements of a whole list**.

2.4 List comprehension

The fourth and last method to use a `for` loop in combination with lists is called **list comprehension**, which is **a one-line command to create or modify a list in a fast and compact way**. It might look complex at first glance, but we are going to untangle it right away!

- Capitalize each string using a list comprehension containing a `for` loop through indices:

```
[5]: 1  last_names = ["garcia", "smith", "zhang"]     last names is assigned garcia, smith,
                                                     zhang
     2  last_names = [last_name.title() for          last names is assigned last name dot
                      i in range(len(last_names))]   title for i in range len last names
     3  print(last_names)                             print last names
     ['Garcia', 'Smith', 'Zhang']
```

At line 2, we see: (1) the list name, (2) the assignment operator, and (3) the list comprehension. In the list comprehension, there are two components embedded within a pair of **square brackets**: (1) the **value of the list element** that we are going to insert into the list—that is, `last_name.title()`—and (2) a **`for` loop header**—that is, `for i in range(len(last_names))`. To better understand the syntax, let's have a look at Figure 22.2 comparing the `for` loop through indices from cell 2 and the list comprehension from the cell above.

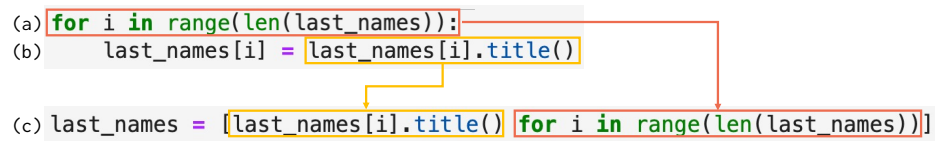


Figure 22.2. Comparison between a `for` loop through indices (lines a–b) and the corresponding list comprehension (line c).

As you can see, the components of a list comprehension are the same as the components of a `for` loop, just in a somewhat inverted position. In a `for` loop, first we write the header (line (a); orange rectangle), and then we assign the modified element (yellow rectangle) to the element itself (line (b)). In a list comprehension (line (c)), we write first the modified element (yellow rectangle) and then the `for` loop header (orange rectangle).

Can we write a list comprehension containing the header of a `for` loop through elements? Yes! Let's see how.

- Capitalize each string using list comprehension containing a `for` loop through elements:

```
[6]:
1 last_names = ["garcia", "smith", "zhang"]
2 last_names = [last_name.title() for
                last_name in last_names]
3 print(last_names)
['Garcia', 'Smith', 'Zhang']
```

```
last names is assigned garcia, smith, zhang
last names is assigned last name dot title
for last name in last names
print last names
```

Similarly to before, in the list comprehension we write first the new element of the list—that is, `last_name.title()`—and then the header of a `for` loop through elements—that is, for `last_name` in `last_names` (line 2). Let's compare the `for` loop through elements from cell 3 with the list comprehension in the cell above. This time, there is a big difference between the `for` loop and the corresponding list comprehension. Can you find it?

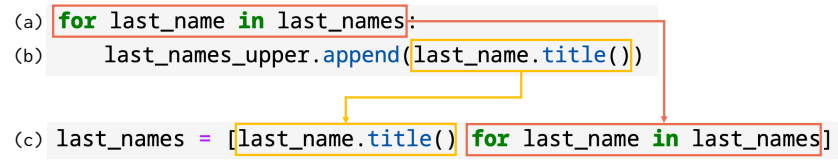


Figure 22.3. Comparison between `for` loop through elements (lines a–b) and the corresponding list comprehension (line c).

The difference is that in a **`for` loop through elements**, we must **create a new list**—that is, `last_names_upper` (line (b))—whereas in the **list comprehension**, we can **overwrite the existing list**—that is, `last_names` (line (c)). The remaining syntax correspondence is the same as above. In a `for` loop, first we write the header (line (a); orange rectangle), and then we modify an element (line (b); yellow rectangle). On the other hand, in a list comprehension (line (c)), we write first a modified element (yellow rectangle) and then a `for` loop header (orange rectangle).

Another interesting characteristic of list comprehensions is that they **can contain a conditional construct**. Let's have a look at it!

- Keep and capitalize only the elements shorter than 6 characters:

```
[7]:
1  last_names = ["garcia", "smith", "zhang"]
2  last_names = [last_name.title() for
                 last_name in last_names if
                 len(last_name) < 6]
3  print(last_names)
['Smith', 'Zhang']
```

```
last names is assigned garcia, smith, zhang
last names is assigned last name dot title
for last name in last names if len last
name less than six
print last names
```

We modify the code from cell 6 by adding an **`if` condition at the end** of the list comprehension (line 2). Once more, let's compare the construct of a list comprehension with the corresponding `for` loop.

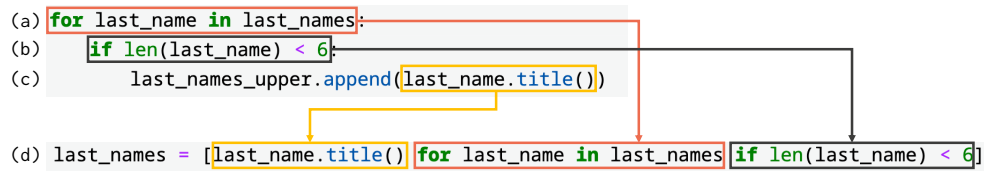


Figure 22.4. Comparison between `for` loop through elements with condition (lines a–c) and the corresponding list comprehension (line d).

Similarly to above, in the list comprehension (line (d)) first we write the new element, which is in the last line of the `for` loop body (yellow rectangle; line (c) in the `for` loop). Then, we restart from the beginning of the loop and add commands consecutively. Thus, we first write the `for` loop header (orange rectangle; line (a) in the loop) and then the `if` condition (black rectangle; line (c) in the loop).

Finally, list comprehensions are extremely useful to **delete list elements based on conditions**. In cell 16 of the previous chapter, we used a `while` loop containing `.remove()` to delete several elements with similar characteristics. Now, let's learn how to delete elements in a much more compact way with list comprehension.

- Delete elements that are composed of 5 characters:

```
[8]:
1  last_names = ["garcia", "smith", "zhang"]
2  last_names = [last_name.title()
                 for last_name in last_names
                 if len(last_name) != 5]
3  print(last_names)
['garcia']
```

```
last names is assigned garcia, smith, zhang
last names is assigned last name dot title
for last name in last names if len last
name not equal to five
print last names
```

When deleting elements with a list comprehension, **we have to think about the elements that we are going to keep, not about those that we are going to delete!** This is because in a list comprehension, in the first position we write the element that we are going to insert into the resulting list. Thus, to delete the elements whose length is 5, we need to reverse our thinking and write the condition that allows us to keep the elements whose length is not equal to 5—that is, `if len(last_name) != 5` (line 2).

✎ Complete the table

In this chapter, you have learned four different ways to write a for loop with lists. Which one can we use and when? Highlight the differences among the for loops by completing the following table with Yes or No.

Operation	For loop through indices	For loop through elements	For loop through indices and elements	List comprehension
Getting the current index				
Changing list elements				
Deleting list elements				
Browsing a full list				
Browsing only a part of a list				

3. Nested for loops

There is one last concept to learn about `for` loops: nested `for` loops. A **nested `for` loop** is simply a **`for` loop inside another `for` loop**. How does it work? Read the example below, and try to understand what happens.

- Given the following list of vowels:

```
[9]: 1 vowels = ["A", "E", "I", "O", "U"]          vowels is assigned A, E, I, O, U
```

We start with a list of strings (line 1).

- For each vowel, print all the vowels on the right:

```
[10]: 1 for i in range(len(vowels)):                for i in range len vowels
      2     print("-- " + vowels[i])                print dash dash concatenated with vowels
                                                    in position i
      3     for j in range(i + 1, len(vowels)):     for j in range i plus one len vowels
      4         print(vowels[i])                    print vowels in position i
-- A
E
I
O
U
-- E
I
O
U
-- I
O
U
-- O
U
-- U
```

The nested `for` loop in this example is composed of an **outer `for` loop**, whose header is at line 1, and an **inner `for` loop**, whose header is at line 3. In the outer `for` loop, the index `i` goes from `0` (omitted) to the length of the list; thus, `i` will browse all list positions. In the inner `for` loop, the index `j` goes from

`i+1` to the length of the list; thus, `j` will browse all remaining list positions on the right of the current position `i`. **For each iteration of the outer loop, the inner loop has to be completed before moving to the next iteration of the outer loop**. Here is what happens at each loop:

- In the first outer loop, `i` is `0`. We print `"-- " + vowels[0]`, which is `-- A` (line 2). Then, we run the whole inner `for` loop (lines 3–4). The index `j` will start at `i+1`—which is `0+1`, and thus `1`—and stop at `len(vowels)-1` for the plus one rule—that is, `4`. Thus, `j` will go through the positions: `[1, 2, 3, 4]`. Therefore, in the inner `for` loop:
  - In the first iteration, `j` is `1`. We print `vowels[1]`, which is `E`.
  - In the second iteration, `j` is `2`. Thus, we print `vowels[2]`, which is `I`.
  - In the third iteration, `j` is `3` and we print `vowels[3]`, which is `O`.
  - In the fourth iteration, `j` is `4` and we print `vowels[4]`, which is `U`. The inner loop is completed and we go back to the outer loop.
- In the second outer loop, `i` is `1`, thus we print `"-- " + vowels[1]`, which is `-- E` (line 2). Then, we run the whole inner `for` loop again (lines 3–4). The start of the inner loop is `i+1`, which is `1+1`—that is, `2`. Thus, `j` will go through the positions: `[2, 3, 4]`. Therefore, in the inner loop:
  - In the first loop, `j` is `2` and we print `vowels[2]`, which is `I`.
  - In the second loop, `j` is `3` and we print `vowels[3]`, which is `O`.
  - In the third loop, `j` is `4` and we print `vowels[4]`, which is `U`. Once again, the inner loop is completed and we go back to the outer loop.
- In the third outer loop, `i` is `2`, so we print `-- I`. Then, we run the full inner loop as above, with `j` browsing the positions `3` and `4`, corresponding to the elements `O` and `U`, respectively.
- In the fourth outer loop, `i` is `3`, so we print `-- O`. In the inner loop, `j` is assigned only the position `4`, corresponding to the elements `U`.
- In the last outer loop, `i` is `4`, so we print `-- U`. There is no inner loop, because the start, `i+1`, is `5` and exceeds the stop, which is `4`.

Can we have more loops nested within each other? Yes! As a convention, the index names are `i`, `j`, `k`, etc. However, it is strongly recommended not to use too many `for` loops because they are computationally very expensive, that is, they use a lot of memory and time. We will talk a bit more about nested `for` loops in the next chapter, where we will use them to browse lists of lists.

Recap

- When we use a `for` loop to repeat commands that do not need the index, we substitute the index with an underscore.
- There are at least 4 types of `for` loops with lists: through indices (uses `range()`), through elements, through indices and elements (uses `enumerate()`), and list comprehension.
- The built-in functions `list()` can be used to transform the output of `range()` and `enumerate()` into a list.
- The built-in function `enumerate()` simultaneously extracts coupled indices and elements from a list.
- Tuples are sequences of elements separated by commas and in between round brackets.
- Nested `for` loops are `for` loops within `for` loops.

Basics of Markdown

As you know, in Jupyter Notebooks we can use cells to either write code or to write text. Writing text is fundamental to embed our code into a story (or narrative) that explains the workflow—that is, how we go from the problem formulation to its computational solution. In Jupyter Notebooks, narrative is written in a markup language called Markdown—markup languages are basically coding languages used to write text. Markdown is a simplified version of HTML, the coding language used to program websites. The syntax of Markdown is very simple. The basic syntax rules are:

- Titles start with 1 hash symbol (#), subtitles with 2 hash symbols (##), sub-subtitles with 3 hash symbols (###), etc. to a maximum of 6 hash symbols (######).

Command	Rendering
`#Title`	Title
`##Subtitle`	Subtitle
`###Sub-subtitle`	Sub-subtitle

- To italicize text, we add 1 asterisk before and after a word or phrase; to bold text, we add 2 asterisks before and after a word or phrase.

Command	Rendering
`*italic text*`	*italic text*
`**bold text**`	**bold text**

- To display text as code, we add a backtick ` before and after the command.

Command	Rendering
`` `print('command in markdown')` ``	`print('command in markdown')`

Using Markdown, we can also create tables, add images, write ordered and unordered lists, etc., and integrate HTML code—in case you know it. You can find a comprehensive Markdown guide at the following website: `www.markdownguide.org/`.

Let's code!

1. *All you can eat.* These friends are at an all-you-can-eat restaurant:
   `friends = ["Geetha", "Huanxiang", "Megan", "Pedro"]`
   This is the finger food at the buffet: `food = ["sushi", "nachos", "samosa", "cheese"]`
   Each person tries each type of finger food. Print sentences like:
   Geetha eats sushi
   Geetha eats nachos
   ...
   for all the friends:
   a. Using nested `for` loops through indices.
   b. Using nested `for` loops through elements.

2. *Playing kids.* At kindergarten, kids are playing a game where they have to pair up with another kid every time the teacher rings a bell. Eventually, every kid will pair up with all the other kids. Given this list of kids:

   ```
   kids = ["Paul", "Juhee", "Luca", "Maria"]
   ```

   a. Print all the possible combinations starting from the first kid, that is:
   Paul plays with Juhee
   Paul plays with Luca
   Paul plays with Maria
   Juhee plays with Luca
   Juhee plays with Maria
   Luca plays with Maria
   b. Print all the possible combinations starting from the last kid (Maria).
3. *Cities of the world.* Given the following list of cities:

   ```
   cities = ["Bogota", "Riga", "Kinshasa", "Damascus", "New Delhi", "Auckland"]
   ```

   a. Using a `for` loop through indices, create a new list containing city names with more than 7 characters and change them to upper case.
   b. Using a `for` loop through elements, create a new list containing initials of cities with a number of characters between 7 and 10.
   c. Using a `for` loop through indices and elements, print each element in lower case and its position.
   d. Using a list comprehension, create a new list containing the city names with less than 7 characters and change them to lower case.
4. *Learning to count.* Print consecutive numbers from 10 to 29 using a nested `for` loop. The outer `for` loop will print the first digit, whereas the inner `for` loop will print the second digit, such as:

   10
   11
   12
   ...
   29
5. *Triangle of numbers.* Ask a user for a number. Then print a triangle of numbers where the maximum row is the queried number. For example:

   Input: 5
   Output:
   1
   2 2
   3 3 3
   4 4 4 4
   5 5 5 5 5
   Hint: Consider using the parameter `end` in the `print()` function. Look for examples on how to use `end` online.

23. Lists of lists

Slicing, nested for loops, and flattening

Let's conclude these three chapters dedicated to lists and `for` loops by exploring lists of lists. You will learn what lists of lists are, how to slice them, how to iterate through them using nested `for` loops, and how to flatten them. Follow along with Notebook 23. Let's go!

First of all: what is a list of lists? A **list of lists** is simply a **list whose elements are lists**. List of lists follow the same rules as lists. Let's look at some of their characteristics!

1. Slicing

To slice a list of lists, we extend the slicing rules that we learned for lists in Chapter 6 by simply **adding one extra layer of indices**. Let's see how it works!

- Given the following list of lists:

```
[1]:
1  animals = [["dog", "cat"], ["cow",
   "sheep", "horse", "chicken", "rabbit"],
   ["panda", "elephant", "giraffe",
   "penguin"]]
```

```
animals is assigned dog, cat, cow, sheep,
horse, chicken, rabbit, panda, elephant,
giraffe, penguin
```

The list of lists `animals` is composed of three elements, which are the lists `["dog", "cat"]`, `["cow", "sheep", "horse", "chicken", "rabbit"]`, and `["panda", "elephant", "giraffe", "penguin"]` (line 1). We call each of these lists **sub-lists** and their elements (`"dog"`, `"cat"`, `"cow"`, `etc.`) **sub-elements**. Let's learn how to slice sub-lists and sub-elements!

- Print the sub-lists containing pets, farm animals, and wild animals:

```
[2]:
1  print(animals[0])
2  print(animals[1])
3  print(animals[2])
```

```
print animals in position zero
print animals in position one
print animals in position two
```

```
['dog', 'cat']
['cow', 'sheep', 'horse', 'chicken', 'rabbit']
['panda', 'elephant', 'giraffe', 'penguin']
```

The sub-list containing pets—`["dog", "cat"]`—is in position `0`; thus, we print `animals[0]`. Similarly, the list containing farm animals—`["cow", "sheep", "horse", "chicken", "rabbit"]`—is in position 1, so we print it with the command `print(animals[1])` (line 2). Finally, the list containing wild animals—`["panda", "elephant", "giraffe", "penguin"]`—is in position `2`, and thus the command is `print(animals[2])` (line 3).

- Print the sub-elements `"cat"`, `"rabbit"`, and from `"panda"` to `"giraffe"`:

```
[3]: 1 print(animals[0][1])
     2 print(animals[1][-1])
     3 print(animals[2][:3])
```

```
cat
rabbit
['panda', 'elephant', 'giraffe']
```

```
print animals in position zero in position one
print animals in position one in position minus one
print animals in position two in position from the
beginning of the sub-list to three
```

To extract sub-elements, we use **double slicing**, where **the first slicing—indicated by the first pair of square brackets—extracts a sub-list** and the **second slicing—indicated by the second pair of square brackets—extracts one or more sub-elements**. To extract the sub-element `"cat"`, first we extract the sub-list of pets `["dog", "cat"]` with the command `animals[0]`—like in cell 2, line 1. Then, from the obtained sub-list, we slice `"cat"`, which is in position `1`. Thus, the complete command is `animals[0][1]` (line 1). The string `"rabbit"` is the last element of the second sub-list containing farm animals. Thus, to slice `"rabbit"`, we write `animals[1][-1]`, where the first slicing `[1]` extracts the sub-list of farm animals—as we did at cell 2, line 2—and the second slicing `[-1]` extracts the last sub-element of the sub-list—that is, `"rabbit"` (line 2). Finally, the sub-elements from `"panda"` to `"giraffe"` are in the sub-list of wild animals, which is `animals[2]`—as we saw in cell 2, line 3. Within this sub-list, `"panda"` is in position `0`, which we omit, and `"giraffe"` is in position `2`, to which we add `1` for the plus one rule. Thus, the final command is `print(animals[2][:3])`.

2. Browsing with nested for loops

To browse elements in a list of lists, we can use nested `for` loops, where the **outer loop browses the list of lists** and the **inner loop browses the sub-lists**. Try to understand what the following example does and then read the explanation.

- Given the following list of lists:

```
[4]: 1 sports = [["skiing", "skating",
     "curling"], ["canoeing", "cycling",
     "swimming", "surfing"]]
```

```
sports is assigned skiing, skating, curling,
canoeing, cycling, swimming, surfing
```

We start with a list of lists containing two sub-lists. The first sub-list contains 3 strings, and the second sub-list is composed of 4 strings (line 1).

- Print the sub-list elements one-by-one using nested `for` loops through indices:

```
[5]: 1 for i in range(len(sports)):
     2     for j in range(len(sports[i])):
     3         print(sports[i][j])
```

```
skiing
skating
curling
canoeing
cycling
swimming
surfing
```

```
for i in range len sports
for j in range len sports in position i
print sports in position i in position j
```

In the outer `for` loop, the index `i` iterates through the positions `0`—corresponding to the sub-list `["skiing", "skating", "curling"]`—and `1`—corresponding to the sub-list `["canoeing", "cycling", "swimming", "surfing"]`—(line 1). During each outer `for` loop, the inner `for` loop browses the current sub-list from `0` (omitted) to the length of the sub-list, which is `len(sports[i])` (line 2). At each iteration of the inner `for` loop, we print the current element `sports[i][j]` (line 3). In practice:

- In the first outer loop, `i` is `0`, and we execute a full inner loop to browse the first sub-list:
  - In the first inner loop, `j` is `0`, so we print `sports[0][0]`, which is `"skiing"`.
  - In the second inner loop, `j` is `1`, so we print `sports[0][1]`, which is `"skating"`.
  - In the third inner loop, `j` is `2`, so we print `sports[0][2]`, which is `"curling"`. The inner `for` loop is over, and we go to the second outer `for` loop.
- In the second outer loop, `i` is `1`, and we execute another full inner loop to browse the second sub-list:
  - In the first inner loop, `j` is `0`, so we print `sports[1][0]`, which is `"canoeing"`.
  - In the second inner loop, `j` is `1`, so we print `sports[1][1]`, which is `"cycling"`.
  - In the third inner loop, `j` is `2`, so we print `sports[1][2]`, which is `"swimming"`.
  - In the fourth inner loop, `j` is `3`, so we print `sports[1][3]`, which is `"surfing"`. The inner `for` loop is over; also, the outer `for` loop is concluded because there are no more sub-lists.

Can we do the same with a `for` loop through elements? Yes! Think about how we could do it before looking into the following code.

- Print the sub-list elements one-by-one using nested `for` loops through elements:

[6]:
```
1 for seasonal_sports in sports:
2     for sport in seasonal_sports:
3         print(sport)
```
```
skiing
skating
curling
canoeing
cycling
swimming
surfing
```
```
for seasonal sports in sports
for sport in seasonal sports
print sport
```

In the outer `for` loop, the variable `seasonal_sports` is assigned once the first sub-list and once the second sub-list (line 1). In the inner `for` loop, the variable `sport` is assigned each element of the current sub-list (line 2). For each iteration of the inner `for` loop, we print the current value of the variable `sport` (line 3). In other words:

- In the first iteration of the outer `for` loop, `seasonal_sports` is `["skiing", "skating", "curling"]` and the inner `for` loop browses all the sub-elements of `seasonal_sports` in the following way:
  - In the first inner loop, `sport` is `"skiing"`.
  - In the second inner loop, `sport` is `"skating"`.
  - In the third inner loop, `sport` is `"curling"`. The inner `for` loop ends, and we go back to the outer `for` loop.
- In the second iteration of the outer `for` loop, `seasonal_sports` is `["canoeing", "cycling", "swimming", "surfing"]`, and the inner `for` loop browses all the sub-elements of `seasonal_sports` in the following way:
  - In the first inner loop, `sport` is `"canoeing"`.
  - In the second inner loop, `sport` is `"cycling"`.
  - In the third inner loop, `sport` is `"swimming"`.

- In the fourth inner loop, `sport` is `"surfing"`. The inner `for` loop ends—as does the outer `for` loop because we went thought all the sub-lists.

3. Flattening

Flattening means transforming a list of lists into a list. In other words, we take the sub-elements out of their sub-lists and we put them in a list. There are many ways of performing this operation. We'll look at four different ways of doing so, but there can be more. For each method of flattening, try to implement it yourself first, and then look into the example and explanation below.

- Given the following list of lists:

```
[7]:
1  instruments = [["contrabass", "cello",
   "clarinet"], ["gong", "guitar"],
   ["tambourine", "trumpet", "trombone",
   "triangle"]]
```

```
instruments is assigned contrabass,
cello, clarinet, gong, guitar,
tambourine, trumpet, trombone, triangle
```

- Flatten the list using nested `for` loops through indices:

```
[8]:
1  flat_instruments = []
2  for i in range(len(instruments)):
3      for j in range(len(instruments[i])):
4          flat_instruments.append
                       (instruments[i][j])
5  print(flat_instruments)
```

```
flat instruments is assigned empty list
for i in range len instruments
for j in range len instruments in
position i
flat instruments dot append instruments
in position i in position j
print flat instruments
```

```
['contrabass', 'cello', 'clarinet', 'gong', 'guitar', 'tambourine', 'trumpet',
'trombone', 'triangle']
```

We start with the empty list `flat_instruments`, which we are going to fill out during the subsequent nested `for` loop (line 1). Then, for each position in the list of lists (line 2) and each position in each sub-list (line 3), we append the current sub-element `instruments[i][j]` to `flat_instruments` (line 4). Finally, we print the final list (line 5). As you can see, we flattened `instruments`, that is, we transformed a list of lists into a list whose elements are `instruments`'s sub-elements.

- Flatten the list using nested `for` loops through elements:

```
[9]:
1  flat_instruments = []
2  for group in instruments:
3      for instrument in group:
4          flat_instruments.append(instrument)
5  print(flat_instruments)
```

```
flat instruments is assigned empty list
for group in instruments
for instrument in group
flat instruments dot append instrument
print flat instruments
```

```
['contrabass', 'cello', 'clarinet', 'gong', 'guitar', 'tambourine', 'trumpet',
'trombone', 'triangle']
```

Like the previous example, we start with the empty list `flat_instruments` (line 1). We browse the sub-lists using the outer `for` loop (line 2), and within each sub-list, we browse the sub-elements using the inner `for` loop (line 3). We append the current sub-element to `flat_instruments` (line 4). Finally, we print the obtained flattened list (line 5).

- Flatten the list using a `for` loop and list concatenation:

```
[10]: 1 flat_instruments = []                    flat instruments is assigned empty list
      2 for group in instruments:                for group in instruments
      3     flat_instruments += group            flat instruments increased by group
      4 print(flat_instruments)                  print flat instruments
['contrabass', 'cello', 'clarinet', 'gong', 'guitar', 'tambourine', 'trumpet',
'trombone', 'triangle']
```

Once more, we start with the empty list `flat_instruments` (line 1). We write a `for` loop through elements to browse the sub-lists (line 2). We concatenate each sub-list to `flat_instruments` (line 3)—the corresponding explicit command is: `flat_instruments = flat_instruments + group`. Finally, we print `flat_instruments` (line 4). The advantage of this method is that we use only one `for` loop. As you might remember, `for` loops are computationally expensive—in terms of memory and time—and it is good practice to minimize their use.

- Flatten the list using list comprehension:

```
[11]: 1 instruments = [instrument for group in   instruments is assigned instrument for
         instruments for instrument in group]    group in instruments for instrument in
                                                 group
      2 print(instruments)                       print instruments
['contrabass', 'cello', 'clarinet', 'gong', 'guitar', 'tambourine', 'trumpet',
'trombone', 'triangle']
```

As you might remember from the previous chapter, when using list comprehension, we do not need to create a new list, but we can directly modify the current one—which is `instruments` in this example. In the list comprehension, we write: (1) what we want to add to the list, which is `instrument`; (2) the header of the outer `for` loop, that is, `for group in instruments`; and (3) the header of the inner `for` loop, which is `for instrument in group` (line 1). Note that within a list comprehension, we can use nested `for` loops through elements because we do not need element positions. Finally, we print the result (line 2).

Recap

- Lists of lists are lists with lists as elements.
- When slicing, we use two pairs of square brackets. In the first pair, we write the position of the sub-list to slice; in the second pair, we write the position of the sub-element(s).
- We can use nested `for` loops to browse sub-elements.
- We can flatten a list of lists with a nested `for` loop, a `for` loop combined with concatenation, or a list comprehension.

Digital images

Have you ever wondered what a digital image is and how it gets its colors? Digital images are composed of **pixels**, that is, small squares organized in a grid. We can think of the grid as a **list of lists** where each sub-element corresponds to a pixel. How do we add colors to pixels?

Let's consider a **black and white** image, like the one in Figure 23.1 representing a checkerboard.

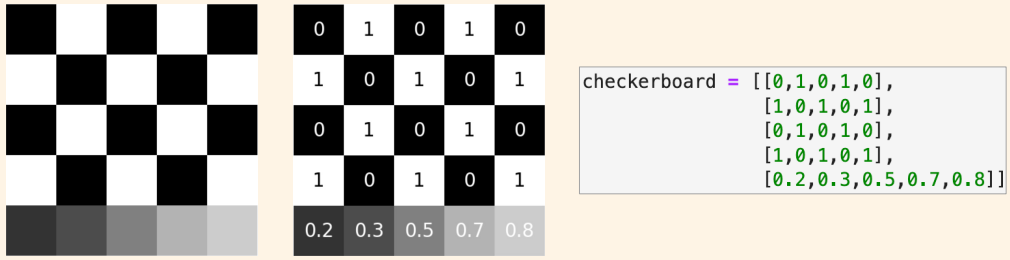


Figure 23.1. Digital representation of a checkerboard. Left: Image rendering. Center: Numerical values corresponding to the checkerboard colors. Right: List of lists encoding the checkerboard colors.

Each black square corresponds to a pixel containing `0`, and each white square corresponds to a pixel containing `1`. Thus, the first (and the third) row of the checkerboard are represented by the sub-list `[0,1,0,1,0]`, whereas the second (and the fourth) row are represented by the sub-list `[1,0,1,0,1]`. What if we want shades of gray, like in the last row of the checkerboard? In this case, each pixel corresponds to a decimal (float) number. Darker greys are represented by numbers closer to `0`—that is, to black—whereas brighter greys are represented by numbers closer to `1`—that is, to white.

What about **colored images**? In this case, each pixel is encoded by a red-green-blue (RGB) list composed of three numbers, each representing a different color: the first number is for the red (R) component, the second number for the green (G) component, and the third number for the blue (B) component. Let's have a look at Figure 23.2.

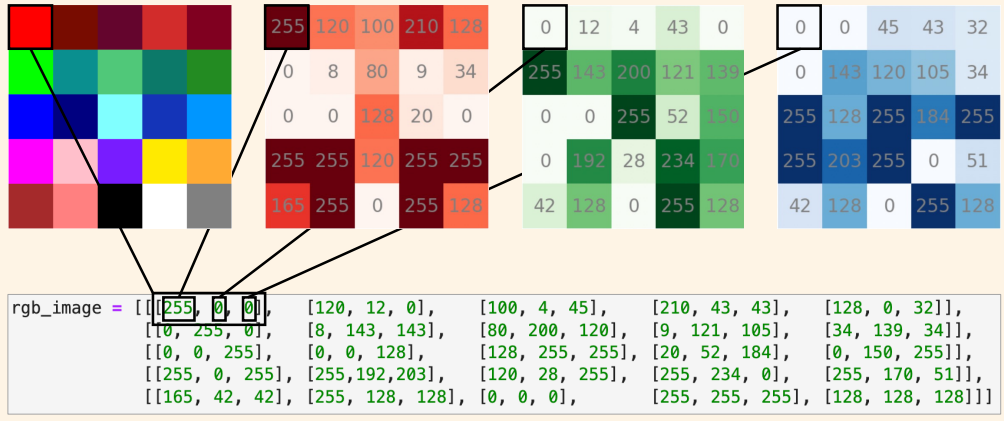


Figure 23.2. Digital representation of a colored image. Top (from left to right): RBG image, red components, green components, and blue components. Bottom: list of lists behind the rendered colored image.

For example, the top left pixel is red and is represented by the sub-list `[255, 0, 0]`, where `255` represents the (full!) amount of red, the first `0` is for the (absent!) amount of green, and the

second zero `0` is for the (absent!) amount of blue. Note that each row is a list of lists, enclosed in a list of lists of lists!

Finally, note that for both greyscale and colored images, the range of the numbers defining the color can go from `0` to `1` or from `0` to `255`.

Let's code!

1. *Playing around.* Given the following list of lists:
   `numbers = [[3,7,1],[7,6,5,4],[8,9,7,4,5]]`.
   a. How long is each sub-list?
   b. In the first sub-list, replace the third element with the sum of the previous two elements.
   c. In the second sub-list, sort the elements in ascending order.
   d. In the third sub-list, substitute the number `4` with the number `3`.
   e. How many number `7` are there in total? Save their positions in a list of lists (expected result: `[[0, 1], [1, 3], [2, 2]]`).
2. *Summing up.* Given the following list of lists:
   `numbers = [[1,3,5],[7,2,8],[3,4,9]]`.
   a. Create a list containing the sum of the numbers in each sub-list (expected result: [9, 17, 16]).
   b. Sum all the elements of the list of lists using (1) a `for` loop through indices and (2) a `for` loop through elements.
3. *Matrix time!* Give the following matrix:
   `matrix = [[4,1,3,9], [2,1,6,5], [4,0,3,8], [7,2,6,2]]`
   (If you are not familiar with matrices, think of a matrix as a table containing numbers.)
   a. Print the matrix as a 4x4 table (expected result:
      `[4, 1, 3, 9]`
      `[2, 1, 6, 5]`
      `[4, 0, 3, 8]`
      `[7, 2, 6, 2]`).
   b. Multiply all the elements on the main diagonal and print the result (expected result: 24). *Note*: The main diagonal goes from top-left to bottom-right. In this example, the main diagonal contains: 4,1,3,2.
   c. Sum the matrix values vertically (expected result: `[17, 4, 18, 24]`).

PART 7

DICTIONARIES AND OVERVIEW OF STRINGS

In the first three chapters of this part, you will learn a new data type called dictionary. In the last chapter, you will integrate your knowledge of strings with new methods and tricks. Let's start!

24. Inventory at the English bookstore

Dictionaries

You already know several data types: strings, lists, integers, floats, and Booleans. In this chapter, you will learn a new data type called dictionary. What are dictionaries and what can we do with them? Let's start from this example. Read the code below aloud and follow along with Notebook 24.

- You are the owner of an English bookstore, and these are some classics you sell:

```
[]:
1  classics = {"Austen": "Pride and Prejudice",
               "Shelley": "Frankenstein",
               "Joyce": "Ulyssessss"}
2  print(classics)
```

```
classics is assigned Austen: Pride
and Prejudice, Shelley: Frankenstein,
Joyce: Ulyssessss
print classics
```

- You are conducting an inventory, and you need to print authors and titles:

```
[]:
1  # as dict_items
2  print(classics.items())
3  # as a list of tuples
4  print(list(classics.items()))
```

```
as dict items
print classics dot items
as a list of tuples
print list classics dot items
```

- Then, you need to print authors and titles separately:

```
[]:
1  # authors as dict_keys
2  print(classics.keys())
3  # authors as a list
4  print(list(classics.keys()))
5
6  # titles as dict_values
7  print(classics.values())
8  # titles as a list
9  print(list(classics.values()))
```

```
authors as dict keys
print classics dot keys
authors as a list
print list classics dot keys

titles as dict values
print classics dot values
titles as a list
print list classics dot values
```

- You notice that the title of the last book is wrong, so you correct it:

```
[]:
1  print("Wrong title: " + classics["Joyce"])

2  classics["Joyce"] = "Ulysses"

3  print("Corrected title: " + classics["Joyce"])
```

```
print Wrong title: concatenated
with classics at key Joyce
classics at key Joyce is assigned
Ulysses
print Corrected title: concatenated
with classics at key Joyce
```

- Then you add two new books that have just arrived: Gulliver's Travels by Swift and Jane Eyre by Bronte:

```
[]:
1  # adding the first book (syntax 1)
2  classics["Swift"] = "Gulliver's travels"
3  print(classics)
4
5  # adding the second book (syntax 2)
6  classics.update({"Bronte": "Jane Eyre"})
7  print(classics)
```

```
adding the first book (syntax 1)
classics at key Swift is assigned
Gulliver's travels
print classics

adding the second book (syntax 2)
classics dot update Bronte: Jane Eyre
print classics
```

- Finally, you remove the books by Austen and Joyce because you have just sold them:

```
[]:
1  # deleting the first book (syntax 1)
2  del classics["Austen"]
3  print(classics)
4
5  # deleting the second book (syntax 2)
6  classics.pop("Joyce")
7  print(classics)
```

```
deleting the first book (syntax 1)
del classics at key Austen
print classics

deleting the second book (syntax 2)
classics dot pop Joyce
print classics
```

To continue discovering dictionaries, solve the following exercise!

✎ True or false?

1.	A dictionary is a Python type enclosed in squared brackets.	T	F
2.	In a dictionary, items are in pairs and are separated by commas.	T	F
3.	Items are composed of a key and a value separated by an exclamation mark.	T	F
4.	.items(), .keys(), .values(), .update(), and .pop() are dictionary elements.	T	F
5.	To add an item to a dictionary, we can use either the keyword del or the method .pop().	T	F

Computational thinking and syntax

Let's learn dictionaries step-by-step. Let's start by running the first cell.

```
[1]:
1  classics = {"Austen": "Pride and Prejudice",
               "Shelley": "Frankenstein",
               "Joyce": "Ulyssessss"}
2  print(classics)
```

```
classics is assigned Austen: Pride
and Prejudice, Shelley: Frankenstein,
Joyce: Ulyssessss
print classics
```

```
{'Austen': 'Pride and Prejudice', 'Shelley': 'Frankenstein', 'Joyce': 'Ulyssessss'}
```

At line 1, there is a variable called `classics` to which we assign a sequence of **items** separated by **comma** and enclosed within **curly brackets {}**. Each item (e.g., `"Austen":"Pride and Prejudice"`) is composed of a **key** (`"Austen"`) and a **value** (`"Pride and Prejudice"`), which are separated by a **colon**. Thus, we can define a dictionary as follows:

A **dictionary** is a sequence of **items** (or **key: value** pairs)
separated by commas `,` and in between curly brackets `{}`

At line 2, we print the dictionary.

Let's continue by running the second cell.

[2]:
```
1 # as dict_items                          as dict items
2 print(classics.items())                  print classics dot items
3 # as a list of tuples                    as a list of tuples
4 print(list(classics.items()))            print list classics dot items
```
```
dict_items([('Austen', 'Pride and Prejudice'), ('Shelley', 'Frankenstein'),
('Joyce', 'Ulyssessss')])
[('Austen', 'Pride and Prejudice'), ('Shelley', 'Frankenstein'), ('Joyce', 'Ulyssessss')]
```

To print the dictionary **items**, we use the method **`.items()`**, which **extracts items from a dictionary** (line 2). As you can see in the print, `.items()` returns a specific type called `dict_items`, which contains a list whose elements are the dictionary's items. Each item is enclosed in a tuple containing two elements (e.g., `('Austin', 'Pride and Prejudice')`), where the first element is the key (e.g., `('Austin')`) and the second element is the value (e.g., `('Pride and Prejudice')`). To extract the list of tuples, we enclose `classics.items()` in the built-in function `list()`, and to print it, we enclose the whole command in the built-in function `print()` (line 4).

What if we want to extract all keys and all values separately? Let's look at the following cell.

[3]:
```
1 # authors as dict_keys                   authors as dict keys
2 print(classics.keys())                   print classics dot keys
3 # authors as a list                      authors as a list
4 print(list(classics.keys()))             print list classics dot keys
5
6 # titles as dict_values                  titles as dict values
7 print(classics.values())                 print classics dot values
8 # titles as a list                       titles as a list
9 print(list(classics.values()))           print list classics dot values
```
```
dict_keys(['Austen', 'Shelley', 'Joyce'])
['Austen', 'Shelley', 'Joyce']
dict_values(['Pride and Prejudice', 'Frankenstein', 'Ulyssessss'])
['Pride and Prejudice', 'Frankenstein', 'Ulyssessss']
```

To extract dictionary **keys**, we use the method **`.keys()`** (line 2). Like `.items()`, `.keys()` returns a specific datatype called `dict_keys`. To extract the list of keys from the `dict_keys`, we use the built-in function `list()`, and to print it, we embed the whole command in `print()` (line 4). Finally, to extract dictionary **values**, we use the method **`.values()`**, which returns the list of values embedded in another datatype called `dict_values` (line 7). Once again, to extract the list of values, we use `list()`, and to print it, we embed the whole command in `print()` (line 9).

How do we extract a specific **value** and how do we change it? Let's run cell 4.

```
[4]:
1  print("Wrong title: " + classics["Joyce"])
2  classics["Joyce"] = "Ulysses"
3  print("Corrected title: " + classics["Joyce"])
```

```
Wrong title: Ulyssessss
Corrected title: Ulysses
```

```
print Wrong title: concatenated
with classics at key Joyce
classics at key Joyce is assigned
Ulysses
print Corrected title: concatenated
with classics at key Joyce
```

To **slice a value**, the syntax is **`dictionary[key]`** (pronunciation: *dictionary at key*), as we can see in `classics["Joyce"]` (line 1). Doesn't this look similar to the slicing syntax for lists? Let's compare dictionary slicing and list slicing with the help of Figure 24.1. In a **list**, there are elements (e.g., `"burger"`, `"salad"`, `"coke"`)—which are the content of a list—and corresponding indices (e.g., `0`, `1`, `2`)—which define the position of each element. When we want to extract (or slice) an element, we write the **name of the list and the index of the element that we want in between squared brackets** (`list[index]`). Thus, `todays_menu[0]` gives us `"burger"`. Similarly, in a **dictionary**, there are values (e.g., `"Pride and Prejudice"`, `"Frankenstein"`, `"Ulysses"`)—which are the content of a dictionary—and keys (e.g., `"Austen"`, `"Shelley"`, `"Joyce"`)—which define the position of each value. When we want to access (or slice) a value, we indicate the **name of the dictionary and the key corresponding to the value that we want in between squared brackets** (`dictionary[key]`). Thus, `classics["Austen"]` gives us `"Pride and Prejudice"`. The main difference between list slicing and dictionary slicing lies in the way we define the position of an element or value. In lists, indices order elements from position `0` to position `len(list)-1`, in a consecutive and progressive way (we cannot skip a position!). On the other side, in dictionaries, **keys have no specific order**. Also, note that **numbers cannot be used as keys!**

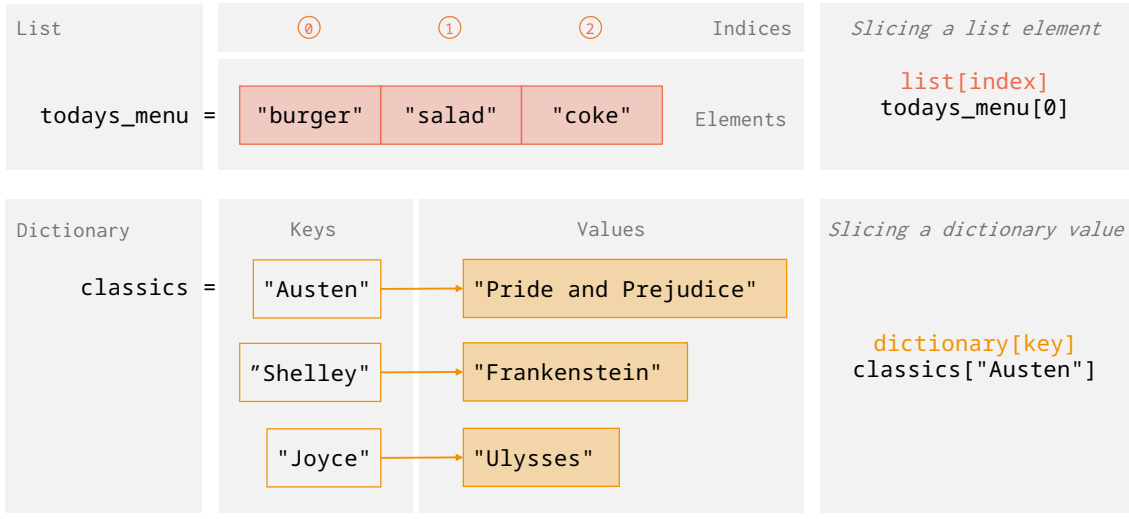


Figure 24.1. Comparison of structure and slicing syntax for lists (top) and dictionaries (bottom).

As we cannot change indices but only elements in lists, **we cannot change keys but only values in dictionaries**. As you might have noticed, in the item `"Joyce":"Ulyssessss"`, we need to correct `"Ulyssessss"` to `"Ulysses"`. To do so, we overwrite the value `"Ulyssessss"` using the same syntax as that used in slicing: `classics["Joyce"] = "Ulysses"` (line 2). Once more, this is the same syntax as that used in lists (e.g., if we want to change `"coke"` to `"water"` in the list in Figure 24.1, we write `todays_menu[2] = "water"`).

At the end of cell 4, we check the correction by printing a string (`"Corrected title: "`) concatenated with the sliced new value (`classics["Joyce"]`, which is `"Ulysses"`; line 3).

How do we **add a new item** to an existing dictionary? There are two ways. Let's learn them in cell 5!

```
[5]:
1  # adding the first book (syntax 1)
2  classics["Swift"] = "Gulliver's travels"
3  print(classics)
4
5  # adding the second book (syntax 2)
6  classics.update({"Bronte": "Jane Eyre"})
7  print(classics)
```

```
adding the first book (syntax 1)
classics at key Swift is assigned Gulliver's travels
print classics
adding the second book (syntax 2)
classics dot update Bronte: Jane Eyre
print classics
```

```
{'Austen': 'Pride and Prejudice', 'Shelley': 'Frankenstein', 'Joyce': 'Ulysses',
'Swift': 'Gulliver's travels'}
{'Austen': 'Pride and Prejudice', 'Shelley': 'Frankenstein', 'Joyce': 'Ulysses',
'Swift': 'Gulliver's travels', 'Bronte': 'Jane Eyre'}
```

The first way to add an item to a dictionary is to use a **slicing-like syntax**, where we write: (1) dictionary name (that is, `classics`), (2) new key in between square brackets (that is, `["Swift"]`), (3) assignment operator (`=`), and (4) new value (that is, `"Gulliver's travels"`) (line 2). The second way is to use the **method `.update()`**. As an argument, we use a `key: value` pair in between curly brackets—that is, a one item dictionary! (line 6). To make sure that we added items correctly, we print the dictionary after every modification (lines 3 and 7).

What about **deleting items**? Let's look into the last cell!

```
[6]:
1  # deleting the first book (syntax 1)
2  del classics["Austen"]
3  print(classics)
4
5  # deleting the second book (syntax 2)
6  classics.pop("Joyce")
7  print(classics)
```

```
deleting the first book (syntax 1)
del classics at key Austen
print classics
deleting the second book (syntax 2)
classics dot pop Joyce
print classics
```

```
{'Shelley': 'Frankenstein', 'Joyce': 'Ulysses', 'Swift': 'Gulliver's travels',
'Bronte': 'Jane Eyre'}
{'Shelley': 'Frankenstein', 'Swift': 'Gulliver's travels', 'Bronte': 'Jane Eyre'}
```

Also in this case, there are two possibilities. The first way to delete an item is to use the **keyword `del`**, followed by the dictionary name and the key enclosed within square brackets (that is, `classic["Austen"]`; line 2). The second way is to use the **method `.pop()`**, with the key of the item to delete as an argument (line 6)—once more, this is similar to lists, where we use the method `.pop()` to delete an element based on its position. After each deletions, we print the dictionary to check for correctness (lines 3 and 7).

✎ Complete the table

In this chapter, you have learned five dictionary methods. Summarize what they do by completing the following table.

Dictionary method	What it does
`.items()`	
`.keys()`	
`.values()`	
`.update()`	
`.pop()`	

Recap

- A dictionary is a Python type containing a sequence of `key: value` items separated by commas, and in between curly brackets `{}`.
- The dictionary methods `.items()`, `.keys()`, and `.values()` are used to access items, keys, and values, respectively.
- To change a dictionary value, we overwrite the existing value using slicing.
- To add a new item, we use a slicing-like syntax or the method `.update()`.
- To delete an item, we use the keyword `del` or the method `.pop()`.

Lists of dictionaries

Can we have lists of dictionaries? Yes! When dealing with them, we just have to remember that they are lists—and not dictionaries! Let's see how they work. Find the code below in Notebook 24.

- Given the following list of dictionaries:

```
[1]:
1  countries = [{"name": "China", "capital": "Beijing"},
                {"name": "France":"capital": "Paris"}]

2  print(countries)
```

countries is assigned name: China, capital: Beijing, name: France, capital: Paris

print countries

```
[{'name': 'China', 'capital': 'Beijing'}, {'name': 'France', 'capital': 'Paris'}]
```

We create a list called `countries`, composed of two elements that are dictionaries—that is, `{"name":"China", "capital":"Beijing"}` and `{"name":"France", "capital":"Paris"}`. Each dictionary is composed of two items, where the keys are `"name"` and `"capital"` (line1). At line 2, we print `countries`. ↳

- Add a list element:

```
[2]:
1  countries.append({"name": "Brazil",
                     "capital": "Brasilia"})
2  print(countries)
```

```
countries dot append name: Brazil,
capital: Brasilia
print countries
```

```
[{'name': 'China', 'capital': 'Beijing'}, {'name': 'France', 'capital': 'Paris'},
 {'name': 'Brazil', 'capital': 'Brasilia'}]
```

Because `country` is a list (and not a dictionary!), we use the method `.append()` (and not `.update`!). As an argument, we write the new dictionary that we want to add as the third element of the list (i.e., `{"name": "Brazil", "capital": "Brasilia"}`; line 1). Then, we print to check for correctness (line 2).

- Slice the second element:

```
[3]:
1  print(countries[1])
```

```
print countries in position one
```

```
[{'name': 'France', 'capital': 'Paris'}]
```

To slice the second element, we use the usual syntax, `list[index]`, and we obtain the desired element (line 1).

- Print list elements using a `for` loop through elements and a `for` loop through indices:

```
[4]:
1  # for loop though elements
2  print("-> for loop though elements")
3  for country in countries:
4      print(country)
5
6  # for loop though indices
7  print("-> for loop though indices")
8  for i in range(len(countries)):
9      print(countries[i])
```

```
for loop though elements
print -> for loop though elements
for country in countries
print country

for loop though indices
print -> for loop though indices
for i in range len countries
print countries in position i
```

```
-> for loop though elements
{'name': 'China', 'capital': 'Beijing'}
{'name': 'France', 'capital': 'Paris'}
{'name': 'Brazil', 'capital': 'Brasilia'}
-> for loop though indices
{'name': 'China', 'capital': 'Beijing'}
{'name': 'France', 'capital': 'Paris'}
{'name': 'Brazil', 'capital': 'Brasilia'}
```

In the `for` loop through elements (lines 3–4), `country` browses the list elements, which are dictionaries. Thus, in the first loop, `country` is `{"name": "China", "capital": "Beijing"}`; in the second loop, `country` is `{"name": "France", "capital": "Paris"}`; and in the third loop, `country` is `{"name": "Brazil", "capital": "Brasilia"}`. In the `for` loop through indices (lines 8–9), `i` iterates over the positions `0`, `1`, and `2`. Thus, `country[i]` browses the corresponding elements—that is, the three dictionaries. ↳

- Print the country names using a `for` loop through indices and a `for` loop through values:

```
[5]:
```

```
1  # for loop though elements
2  print("-> for loop though elements")
3  for country in countries:
4      print(country["name"])
5
6  # for loop though indices
7  print("-> for loop though indices")
8  for i in range(len(countries)):
9      print(countries[i]["name"])
```

```
for loop though elements
print -> for loop though elements
for country in countries
print country at key name

for loop though indices
print -> for loop though indices
for i in range len countries
print countries in position i at
key name
```

```
-> for loop though elements
China
France
Brazil
-> for loop though indices
China
France
Brazil
```

To print the country names, we add a layer of slicing to the `for` loops that we implemented in cell 4. As we mentioned above, in the first iteration of the `for` loop through elements (lines 3–4), `country` is the dictionary `{"name": "China", "capital": "Beijing"}`. To extract `"China"`, we need to slice at the key `"name"`—similarly for the other iterations. Thus, we print `country["name"]`. In the every iteration of the loop through elements (lines 8–9), `country[i]` is one of the dictionaries. To extract the value corresponding to the key `"name"`, we have to write `country[i]["name"]`—in other words: `country[i]` slices the current list element, and `["name]` slices the dictionary at the key `"name"`.

Let's code!

1. *Student's information.* For the following scenario, create code similar to that presented in this chapter. You work in a school Registrar's Office, and here are the data of a student:

   `student = {"First name":"Bruce", "Last name":"Zhiang", "Sex":"Male", "Age":21, "Course":"Literature", "Hobby":"Swimming"}`

   a. Print all the items.
   b. Print all the keys.
   c. Print all the values.
   d. Bruce has recently changed his study course from `Literature` to `Foreign Languages`, so you update his data.
   e. There are two pieces of information missing: `Address` and `Phone number`, so you add them (use two different syntaxes).
   f. Finally, because of new privacy policies, you have to remove `Sex` and `Hobby` (use two different syntaxes).

2. *New T-shirts in the store.* You are the owner of a clothing store, and you are getting ready for the summer season. Your supplier has just provided a new set of trendy T-shirts.

   a. You create a dictionary containing the characteristics of the new T-shirts: they are red, of size M, and

have a round neck.
   b. Then, you add more information: you received a total of 25 T-shirts and their logo's color is blue (use two different syntaxes).
   c. Summer is over, and your sales went well. You have sold 20 T-shirts, so you add a new item containing the number of sold T-shirts.
   d. Finally, you number the amount of T-shirts accordingly (calculate the quantity using the previously created item).

3. *Colosseum.* You are helping your neighbor's kid with her history assignment. She needs to collect data about the Colosseum. So, you go to the Wikipedia page (`https://en.wikipedia.org/wiki/Colosseum`) and look for some information.
   a. You start with some information in a table on the right side of Wikipedia's page. Create a dictionary containing location (Rome), construction years (70–80 AD), and type of structure (amphitheater).
   b. Then you read the text, and in the first paragraph, you learn that construction began in 72 AD and was completed in 80 AD. So, you remove the previous key about the year of construction. Then, you add two separate keys, one for the starting year and one for the completion year (using two different syntaxes).
   c. How many years did it take to build the Colosseum?
   d. How many years have passed since its construction started?

4. *At a pet clinic.* You are a vet at a pet clinic, and here are some of the pets you are currently taking care of:

```
pets = [{"name": "Toby", "animal type": "dog", "age": 2},
        {"name": "Kitty", "animal type": "cat", "age": 5},
        {"name": "Tiki", "animal type": "parrot", "age": 1}]
```

   a. You have just received a new patient, a 4-year-old horse called Sugar, and you add it to the list.
   b. Now, you need to print all the animal names. Do it first with a `for` loop through elements and then with a `for` loop through indices.
   c. Finally, you add an item that states that all the animals are currently in the clinic (what datatype do you use?).

5. *Juices!* You own a juice stand, and you need to keep track of juices and sales.
   a. Create a list of dictionaries containing 3 juice flavors (orange, lemon, and pomegranate), their prices, and their colors.
   b. For each juice, add a new item where the key is `in shop`, and the value is a Boolean.
   c. You just received a new order (grape juice), and you add it to your list.
   d. What is the average price of a juice?

25. Trip to Switzerland

Dictionaries with lists as values

In the previous chapter, you learned about dictionaries and lists of dictionaries. In this chapter, you will learn to code with dictionaries whose values are lists. Follow along with Notebook 25!

- Your friend is planning a trip to Switzerland, and he has asked you for some tips. You start with an empty dictionary to fill out:

```
[]: 1 tips = {}                                    tips is assigned empty dictionary
```

- He would like to visit some cities and taste typical food. Therefore, you add some recommendations:

```
[]: 1 tips["cities"] = ["Bern", "Lucern"]          tips at key cities is assigned Bern,
                                                   Lucern
    2 tips["food"] = ["chocolate", "raclette"]     tips at key food is assigned chocolate,
                                                   raclette
    3 print(tips)                                  print tips
```

- Because his stay is four days, you add two more cities and two more dishes:

```
[]: 1 tips["cities"].append("Lugano")              tips at key cities dot append Lugano
    2 print(tips)                                  print tips
```

```
[]: 1 tips["cities"] += ["Geneva"]                 tips at key cities increased by Geneva
    2 print(tips)                                  print tips
```

```
[]: 1 tips.get("food").append("onion tarte")       tips dot get food dot append onion tarte
    2 print(tips)                                  print tips
```

```
[]: 1 tips["food"] = tips.get("food") + ["fondue"] tips at key food is assigned tips dot
                                                   get food concatenated with fondue
    2 print(tips)                                  print tips
```

- You want to check that the dictionary is correct, so you print all items one by one:

```
[]: 1 for k,v in tips.items():                     for k v in tips dot items
    2     print(k,v)                               print k v
```

- Finally, you improve the print for improved readability:

```
[]: 1 for k,v in tips.items():                     for k v in tips dot items
    2     print("{:>6}: {}".format(k,v))           print symbols dot format k v
```

✎ True or false?

1. There are at list 3 ways to add an element to a list that is a dictionary's value. T F
2. .get() is a list method, and .append() is a dictionary method. T F
3. The built-in function print() can take comma-separated variables as an argument. T F

Computational thinking and syntax

Let's start analyzing the code above by running the first cell:

```
[1]:
1  tips = {}                                   tips is assigned empty dictionary
```

We **initialize an empty list** by **assigning curly brackets** to the variable `tips` (line 1).

Let's run the second cell:

```
[2]:
1  tips["cities"] = ["Bern", "Lucern"]         tips at key cities is assigned Bern,
                                               Lucern
2  tips["food"] = ["chocolate", "raclette"]    tips at key food is assigned chocolate,
                                               raclette
3  print(tips)                                 print tips
{'cities': ['Bern', 'Lucern'], 'food': ['chocolate', 'raclette']}
```

We fill out the empty dictionary `tips` with two new items. The first item has the string `"cities"` as a key and the list `["Bern", "Lucern"]` as a value (line 1). The second item has the string `"food"` as a key and the list `["chocolate", "raclette"]` as a value (line 2). To check for correctness, we print the dictionary (line 3).

We want to add new elements to the two lists that are `tips`'s values. How do we do that? Let's see four possibilities, one in each of the next four cells. In the first two cells we will add a city, and in the last two cells we will add two types of food. In all cases, the command will be composed of two steps: **(1) extracting the value** (i.e., the list) **corresponding to a certain key** and **(2) adding the new element to the list**.

Let's add the first city, which is `"Lugano"`:

```
[3]:
1  tips["cities"].append("Lugano")             tips at key cities dot append Lugano
2  print(tips)                                 print tips
{'cities': ['Bern', 'Lucern', 'Lugano'], 'food': ['chocolate', 'raclette']}
```

First, we **slice** the list from the dictionary—`tips["cities"]` is `["Bern", "Lucern"]`. Then, we add the new elements to the list using **`.append()`** (line 1). Finally, we print to check for correctness (line 2).

Let's add the second city, that is, `"Geneva"`:

```
[4]:
1  tips["cities"] += ["Geneva"]                tips at key cities increased by Geneva
2  print(tips)                                 print tips
{'cities': ['Bern', 'Lucern', 'Lugano', 'Geneva'], 'food': ['chocolate', 'raclette']}
```

Like above, we **slice** the list from the dictionary—`tips["cities"]` is now `["Bern", "Lucern", "Lugano"]`. Then, we use list **concatenation** as an alternative to the method `.append()`. As you might remember, when using list concatenation we must reassign the changed value to the variable. In this example, we combine assignment and concatenation with the `+=` operator—the extended command is `tips["cities"] = tips["cities"] + ["Geneva"]` (line 1). At line 2, we print `tips` to check the dictionary's content.

Let's now add the first type of food, which is `"onion tarte"`:

```
[5]:
1  tips.get("food").append("onion tarte")      tips dot get food dot append onion tarte
2  print(tips)                                 print tips
{'cities': ['Bern', 'Lucern', 'Lugano', 'Geneva'], 'food': ['chocolate', 'raclette',
'onion tarte']}
```

As an alternative to slicing, we can extract a value using the dictionary method **`.get()`**, which takes the corresponding key as an argument. In our case, `.get("food")` returns the list `["chocolate", "raclette"]`. Then, we add the new element (`"onion tarte"`) using the list method **`.append()`** (line 1). As you might have noticed, we created a "chain" of methods, combining a dictionary method (`.get()`) that returns a list, with a list method (`.append()`) that modifies the list. At the end of the cell, we print `tips` to check for correctness (line 2).

Finally, let's add the second type of food, that is, `"fondue"`:

```
[6]:
1  tips["food"] = tips.get("food") + ["fondue"]
2  print(tips)
```

```
tips at key food is assigned tips dot get food concatenated with fondue
print tips
```

```
{'cities': ['Bern', 'Lucern', 'Lugano', 'Geneva'], 'food': ['chocolate', 'raclette',
'onion tarte', 'fondue']}
```

Like above, we use the method **`.get()`** to extract the value corresponding to `"food"`, which is the list `["chocolate", "raclette", "onion tarte"]`. Then, we use **concatenation** to add the last element `"fondue"`. Note that in this case we cannot use the compact operator `+=` because we cannot reassign to `tips.get("food")`. We can only reassign the outcome to `tips["food"]` (line 1). Finally, we print the dictionary to check for correctness (line 2).

In summary, the four ways that we have to add an element to a list that is a value of a dictionary are a combination of **slicing or dictionary method `.get()`** to slice the value from the dictionary, and of **list method `.append()` or concatenation** to add a new element to the list. When coding, you can choose to use only one way or to alternate several ways. But it is important to know all ways to understand code written by somebody else.

In the examples above, you might have noticed that reading the print of a dictionary can be hard when several keys and values are displayed in one long line. Let's learn how to **print an item per line to improve readability**:

```
[7]:
1  for k,v in tips.items():
2      print(k,v)
```

```
for k v in tips dot items
print k v
```

```
cities ['Bern', 'Lucern', 'Lugano', 'Geneva']
food ['chocolate', 'raclette', 'onion tarte', 'fondue']
```

We use a **`for` loop through values** with two variables `k`—for the keys—and `v`—for the values. The two names could be different, but conventionally we use the initial of the variable they represent. `k` and `v` simultaneously browse the dictionary items returned by the `.items()` method (line 1). At each iteration, we print the current key `k` with the corresponding value `v` (line 2). Note that `k` and `v` are separated by comma. This is independent from the fact that we are printing the items of a dictionary. The built-in function **`print()`** can take **variables of different types separated by comma as an argument**. For example, we can use `print("The Swiss cities in the list are", 4)` as an alternative to `print("The Swiss cities in the list are" + str(4))`.

What if we want to **print only the keys or only the values**? Let's have a look!

```
1  for k in tips.keys():
2      print(k)
```

```
for k in tips dot keys
print k
```

```
cities
food
```

In the `for` loop header, we use only the variable `k` in combination with the method `.keys()` (line 1), and we print `k` only (line 2). Similarly for the values:

```
1 for v in tips.values():                    for v in tips dot values
2     print(v)                               print v
```

```
['Bern', 'Lucern', 'Lugano', 'Geneva']
['chocolate', 'raclette', 'onion tarte', 'fondue']
```

In the `for` loop header, we use only the variable `v` in combination with the method `.values()` (line 1), and we print `v` only (line 2).

Finally, let's have a look at one more elegant way to **print** dictionaries, where the **keys are aligned to the right and the values to the left**:

[8]:
```
1 for k,v in tips.items():                   for k v in tips dot items
2     print("{:>6}: {}".format(k,v))         print symbols dot format k v
```

```
cities: ['Bern', 'Lucern', 'Lugano', 'Geneva']
  food: ['chocolate', 'raclette', 'onion tarte', 'fondue']
```

The `for` loop header is the same as in cell 7: `k` and `v` iteratively browse keys and values returned by `.items()` (line 1). The argument of the built-in function `print()` at line 2 looks a bit more complicated. Let's disentangle it! There is a string—constituted by red characters in between quotes—followed by the string method `.format()`, which takes two arguments: `k` and `v`. The string contains two pairs of curly brackets, one with symbols `{:>6}`, and one empty `{}`. These pairs of **curly brackets** have nothing to do with dictionaries. They are **placeholders** for the arguments of the string method `.format()`. The first argument `k` will be printed at the place of `{:>6}` and the second argument `v` at the place of `{}`. What is the meaning of `:>6`? The symbol `:` indicates that we print the whole text; the symbol `>` specifies that the text is aligned to the right; and `6` indicates that the printing space is made of 6 characters—because `cities` has 6 characters. What about the colon between the two placeholders? It is simply the colon printed between each key and the corresponding value—e.g., `cities: ['Bern', ...]` Finally, what is the function of the string method **`.format()`**? It **formats the arguments and inserts them into the placeholders**.

✎ Insert into the right column

Insert string, list, and dictionary methods into the right column:

```
.keys(), .upper(), .insert(), .append(), .values(), .copy(), .lower(), .pop(), .count(),
.format(), .capitalize(), .index(), .extend(), .get(), .items(), .title(), .remove(), .clear(),
.update(), .pop(), .reverse(), .sort()
```

Dictionary methods	String methods	List methods

Recap

- To initialize a dictionary, we use a pair of empty curly brackets `{}`.
- The dictionary method `.get()` takes a key as an argument and returns the corresponding value.
- There are at least 4 different ways to access and modify dictionary values that are lists, by combining:
  - Slicing or `.get()` to extract a list from a dictionary.
  - List operations (such as concatenation) or methods (e.g., `.append()`) to modify a list.
- We can use the `for` loop through values to browse items, keys, and values of a dictionary.
- The built-in function `print()` can take several variables as an argument:
  - Separated by comma, or
  - Using placeholders `{}` in combination with the string method `.format()`.

Dealing with KeyError

When coding with dictionaries, key errors can occur. Let's see what it means and how to fix it! Let's consider the same example as in this chapter, and let's slice the value corresponding to the key `"cities"`:

```
[1]:  1  tips["city"]                    tips at key city
      ----------------------------------------
      KeyError       Traceback (most recent call last)
      Cell In[3], line 1
      ----> 1 tips ["city"]
      KeyError: 'city'
```

As you know, to understand an error, we start from the last line. It says: `KeyError: 'city'`, which means that we made an error on the key `'city'`. To know where the error is, we look for the green arrow, which shows that we need to correct at line 1. In this example, we just **misspelled** the key name. So, we fix the error by replacing `"city"` with `"cities"` in the code. Note that we can get the same error message when a **key does not exist**.

Let's code!

1. For each of the following scenarios, create code similar to that presented in this chapter.
   a. *Olympic Games*. You are a sports journalist, and your task is to collect a dictionary of summer and winter sports performed at the Olympic Games.
      a. Create an empty dictionary that you will fill out with some Olympic Games.
      b. Add two summer sports and two winter sports.
      c. The lists in the values look a bit short. Add two more summer sports and two more winter sports. Add each element with a different method.
      d. Print all items one by one in two different ways.
      e. Finally, print only the sports lists.

   b. *Teaching Python*. You are teaching Python to some students, and you want to list their names according to the course they are attending.
      a. Create an empty dictionary called `students`.
      b. So far, there are two students for the basic course and three students for the advanced course. Add their names to the dictionary.
      c. You have just received four new registrations: three for the basic course and one for the advanced course. So, you add the new students' names to the dictionary using four different ways.
      d. After checking the background of the students attending the basic course, you realize that one of them should be in the advanced course. So you move the student from the basic to the advanced course.
      e. To check for correctness, you print all items one by one in two different ways.
      f. Finally, you print the course names and the students' names separately.

2. *Furniture store*. You are the manager of a furniture store. Here are the pieces of furniture in storage:

```
store = {"furniture": ["chair", "table", "sofa"],
         "amount": [24, 7, 6],
         "price" : [200, 500, 1200]}
```

   a. A new customer comes in and buys 4 chairs. Update the dictionary using an arithmetic operation.
   b. After a few days, you receive new pieces of furniture: 9 carpets worth 150 each and 4 lamps worth 180 each. So, you add them to the dictionary (use different syntaxes).
   c. The owner of a restaurant comes to your shop and buys all the tables. Update the dictionary (use at least 2 different syntaxes).
   d. To better visualize what is left, you print the dictionary aligning the keys to the right and the values to the left.
   e. What is the total price of the furniture in storage?

3. *Shifting list elements*! Given the following dictionary:

   ```
   dictionary = {"numbers": [2,3,4,5,6,7,8,9,10]}
   ```

   a. Add an item where the key is the string `even` and the value is a list containing `True` for even numbers and `False` for odd numbers.
   (Expected result:
   ```
   {"numbers": [2, 3, 4, 5, 6, 7, 8, 9, 10],
    "even": [True, False, True, False, True, False, True, False, True]}).
   ```
   b. Subtract 1 from each number.
   c. How do you modify the Boolean list so that it corresponds to the new list of numbers? *Hint*: Just shift it!

4. *Numbers in a triangle*! Ask a player for an integer. Then, print a triangle where each row contains a consecutive integer between 1 and the number entered by the player. Additionally, each row should include a list containing the number from that row repeated the same number of times as the number itself. To do that, use a dictionary and allow the player to play as long as they want!
   (Example input: `5`.
   Expected output:
   ```
   1 [1]
   2 [2, 2]
   3 [3, 3, 3]
   4 [4, 4, 4, 4]
   5 [5, 5, 5, 5, 5]).
   ```

26. Counting, compressing, and sorting

What are dictionaries for?

In this chapter, the final one dedicated to dictionaries, you will learn some typical situations where using dictionaries is very convenient. Try to solve each example by yourself before looking into the solution. You can find the code in Notebook 26!

1. Counting elements

Dictionaries are extremely convenient when we need to save occurrences, that is, the number of times something happens. Let's understand what this means with the following example.

- Given the following string:

```
[1]:
1 greetings = "hello! how are you?"
```

greetings is assigned hello! how are you?

- Create a dictionary where the keys are the letters of the alphabet found in the string, and the corresponding values are the number of times each letter is present. Write the code in two ways: (1) using `if/else` and (2) using `.get()`.

1. Using `if/else`:

```
[2]:
1  letter_counter = {}

2

3  for letter in greetings:
4      if letter not in letter_counter.keys():
5          letter_counter[letter] = 1

6      else:
7          letter_counter[letter] += 1

8

9  for k,v in letter_counter.items():
10     print(k,v)
```

```
h 2
e 2
l 2
o 3
! 1
  3
w 1
a 1
r 1
y 1
u 1
? 1
```

letter counter is assigned empty dictionary

for letter in greetings
if letter not in letter counter dot keys
letter counter at key letter is assigned one
else
letter counter at key letter increased by one

for k v in letter counter dot items
print k v

We start with an empty dictionary called `letter_counter` (line 1). We browse each character of the string `greetings` using a `for` loop through elements (line 3)—**the `for` loop through elements works**

for strings in the same way as it does for lists. Then, for each character, we check if it is a key of `letter_counter` and we act accordingly (lines 4–7). First, we evaluate if the current character is not already a key of `letter_counter` by checking if `letter`, which is a string, is `not in` the output of `letter_counter.keys()` (line 4). Note that we can directly check the membership of `letter` in `dict_keys` (returned by `.keys()`) without having to transform into a list—in other words, we do not need to write `list(letter_counter.keys())`. If the condition at line 4 is satisfied, then we add a new item, where the key is `letter`, and the value is `1` (line 5). On the other hand, if the current character is already a key in `letter_counter` (`else` at line 6), then we add `1` to the already existing corresponding value (line7)—the explicit command would be `letter_counter[letter] = letter_counter[letter] + 1`. To better understand this, let's look at what happens at the third and fourth loops. At the third loop, `letter` is `l` (he<u>l</u>lo). Because `l` is not already a key in `letter_counter` (line 4), we create a new dictionary item, where `l` is the key and `1` is the value (line 5). At the fourth loop, `letter` is `l` again (he<u>l</u>lo). Because this time `l` is already a key (line 6), we slice the value at `letter_counter[l]`, which is `1`, add `1`, and we reassign it into the dictionary (line 7). We terminate the task by printing each letter and its corresponding amount with a `for` loop through keys and values (lines 9–10).

2. Using `.get()`:

[3]:

```
1  letter_counter = {}
2
3  for letter in greetings:
4      letter_counter[letter] =
         letter_counter.get(letter, 0) + 1
5
6  for k,v in letter_counter.items():
7      print(k,v)
```

```
letter counter is assigned empty
dictionary

for letter in greetings
letter counter at key letter is assigned
letter counter dot get letter zero plus
one

for k v in letter counter dot items
print k v
```

```
h 2
e 2
l 2
o 3
! 1
  3
w 1
a 1
r 1
y 1
u 1
? 1
```

Similarly to cell 2, we start with the empty dictionary `letter_counter` (line 1), continue with a `for` loop through elements (line 3), and conclude by printing the obtained dictionary to check the correctness of the results (lines 6–7). As opposed to what we saw above, the four lines of code containing the `if/else` construct (lines 4–7, cell 2) are replaced by one single line containing the following: an assignment, the method `.get()`, and a sum (line 4). The method `.get()` contains two arguments, `letter` and `0`, and it acts as follows: **if the key does not exist, `.get()` returns the second argument; if the key already exists, `.get()` returns the corresponding value**. In other words, this is what happens at line 4:

- If the current key `letter` does not exist yet—as in the third loop where `letter` is the first `l` in he<u>l</u>lo—then `.get(letter, 0)` returns `0`. Then, we add `1` to `0`, and we create a new item in the dictionary by

assigning the result to `letter_counter[letter]`.
- If the current key `letter` already exists—as in the fourth loop where `letter` is the second `l` in `hello`—then `.get(letter, 0)` returns the value corresponding to `letter`—that is, `1`. We add `1` to the returned `1` to increment the count, and we update the existing item in the dictionary by reassigning.

Why do we use `0` as the second argument? Since in this line of code we need to have `+1` to update the counts of the already existing letters, the only way we have to obtain `1` when we encounter a new letter is to sum to `0`.

2. Compressing information

Dictionaries are extremely convenient for compressing redundant information: for example, to store signals acquired by sensors over a long time. Think of a sensor used to detect vibrations in the case of an earthquake. Most of the time, the sensor just records zeros as there is no seismic event. However, when an earthquake occurs, the sensor registers a spike (or a group of spikes) whose magnitude is different from zero. Saving days and days of zeros in a list would not only require a significant amount of computer memory, but it would also be somewhat pointless because the signal information is in the spikes. To reduce the amount of storage memory while keeping the information, we can use a dictionary. How would you do it? And how would you then go back from the dictionary to the original list?

- Given the following list:

[4]:
```
sparse_vector = [0, 0, 0, 1, 0, 7, 0, 0,
4, 0, 0, 0, 8, 0, 0, 0, 6, 0, 0, 0, 0, 0,
0, 0, 9, 0, 0]
```

```
sparse vector is assigned list of numbers
```

We start with a list called `sparse_vector`, containing many zeros and a few integers spread among the zeros. (Note: in linear algebra, sparse vectors are vectors where the majority of components are zeros.)

- Convert it into a dictionary:

[5]:
```
# create the dictionary
sparse_dict = {}
for i in range(len(sparse_vector)):
    if sparse_vector[i] != 0:

        sparse_dict[i] = sparse_vector[i]


# save the list length
sparse_dict["length"] = len(sparse_vector)


# print
for k,v in sparse_dict.items():
    print(k,v)
```

```
3 1
5 7
8 4
12 8
```

```
create the dictionary
sparse dict is assigned empty dictionary
for i in range len sparse vector
if sparse vector in position i not equal
to zero
sparse dict at key i is assigned sparse
vector in position i

save the list length
sparse dict at key length is assigned
len sparse vector

print
for k v in sparse dict dot items
print k v
```

```
16 6
24 9
length 27
```

We start with an empty dictionary called `sparse_dict` (line 2). Then, we browse the list `sparse_vector` with a `for` loop through indices (line 3) to select and save the information—that is, the nonzero integers and their positions in the list. If the current list element `sparse_vector[i]` is not equal to zero (line 4), then we add a new item to the dictionary `sparse_dict`, where the key is the position of the element in the list—that is, `[i]`—and the value is the current nonzero element—that is, `sparse_vector[i]` (line 5). After the loop, we save an item where the key is the string `"length"`, and the value is the actual length of the list (`len(sparse_vector)`; line 8). This item will be useful to convert the dictionary back into a list, like we will see in the next cell. Finally, we print each dictionary item with a `for` loop through elements to check the correctness of our code (lines 11–12).

- How do we get back to the sparse vector?

[6]:
```
1  # create a list of zeros                  create a list of zeros
2  sparse_vector_back = [0] *                sparse vector back is assigned zero times
                sparse_dict["length"]        sparse dict at key length
3
4  # add nonzero values                      add nonzero values
5  for k,v in sparse_dict.items():           for k v in sparse dict dot items
6      if k != "length":                     if k not equal to length
7          sparse_vector_back[k] = v         sparse vector back at key k is assigned v
8
9  # print                                   print
10 print(sparse_vector_back)                 print sparse vector back
```

```
[0, 0, 0, 1, 0, 7, 0, 0, 4, 0, 0, 0, 8, 0, 0, 0, 6, 0, 0, 0, 0, 0, 0, 0, 9, 0, 0]
```

We start by creating a list of zeros called `sparse_vector_back` of the same length as the original list `sparse_vector`. To create `sparse_vector_back`, we use list replication, where we replicate a list containing a zero (`[0]`) for a number of times equal to the length of the original list—whose value we saved at key `"length"`. Then, we overwrite the nonzero values into the list. With a `for` loop, we browse each item in the dictionary (line 5). If the current key is not equal to `"length"` (line 6)—we need to make sure that we do not access that item!—then we assign the current value `v`, which represents the magnitude of a spike, to the list `sparse_vector_back` in position `k` (line 7). Finally, we print the list to check for correctness (line 10).

3. Sorting dictionaries

In this last example about dictionaries and their applications, we will learn how to sort dictionaries according to their keys or values. Consider a simplified city registry containing citizens' names as keys and their ages as values. Officers might need to sort the registry according to names to send out letters, or according to age to distinguish the kids from the elderly. Let's see how to do it!

- Given the following dictionary:

[7]:
```
1  registry = {"Shaili": 4, "Chris": 90,     registry is assigned Shaili: four, Chris:
   "Maria": 70}                              ninety, Maria: seventy
```

- Sort the dictionary items **according to their keys**:

[8]:
```
1  # create a new dictionary                create a new dictionary
2  sorted_registry = {}                      sorted registry is assigned empty
                                             dictionary
3
4  # sort the keys                           sort the keys
5  sorted_keys = list(registry.keys())       sorted keys is assigned list registry dot
                                             keys
6  sorted_keys.sort()                        sorted keys dot sort
7
8  # fill out the new dictionary             fill out the new dictionary
9  for k in sorted_keys:                     for k in sorted keys
10     sorted_registry[k] = registry[k]       sorted registry at key k is assigned
                                             registry at key k
11
12 print(sorted_registry)                    print sorted registry
{'Chris': 90, 'Maria': 70, 'Shaili': 4}
```

We start with an empty dictionary called `sorted_registry` that will have the same content as `registry`, but the items will be sorted according to the keys (line 2). To sort the keys, we execute two steps. First, we extract the keys using the dictionary method `.keys()`. Then, we convert the output type from `dict_keys` to a list using the built-in function `list()` (line 5). Next, we sort the obtained keys—`['Shaili', 'Chris', 'Maria']`—in alphabetical order using the list method `.sort()`, obtaining `['Chris', 'Maria', 'Shaili']` (line 6). Finally, we browse the list of sorted keys using a `for` loop through elements (line 9) to fill out `sorted_registry`. For each key `k`, we extract the corresponding value in `registry` (`registry[k]`) and assign it to `sorted_registry` at key `k` (`sorted_registry[k]`), thus creating a new dictionary item. For example, in the first loop, `k` is `"Chris"`, so we extract `90` from `registry` (`registry[k]`), and we assign it to `"Chris"` in `sorted_registry` (`sorted_registry[k]`). Then, we do the same for the keys `"Maria"` and `"Shaili"`. Finally, we print `sorted_registry[k]` to check for correctness (line 12).

- Sort the dictionary items **according to their values**:

[9]:
```
1  # create a new dictionary                create a new dictionary
2  sorted_registry = {}                      sorted registry is assigned empty dictionary
3
4  # sort keys according to values           sort keys according to values
5  sorted_keys = sorted(registry,            sorted keys is assigned sorted registry key
                        key=registry.get)    is assigned registry dot get
6
7  # fill out the new dictionary             fill out the new dictionary
8  for k in sorted_keys:                     for k in sorted keys
9      sorted_registry[k] = registry[k]      sorted registry at key k is assigned
                                             registry at key k
10
11 print(sorted_registry)                    print sorted registry
{'Shaili': 4, 'Maria': 70, 'Chris': 90}
```

To sort a dictionary according to values, we use the same procedure as above: we create an empty dictionary (line 2); we sort the keys (line 5); we fill out the empty dictionary using a `for` loop through elements that browses the sorted keys (line 8) and adds sorted `key: value` pairs to the dictionary

(line 9); and we print to check for correctness (line 11). What is different is the way we sort the keys, that is, according to dictionary values. To do that, we use the **built-in function `sorted()`** (line 5), which takes two arguments: (1) the dictionary whose keys we want to sort and (2) the parameter `key` to which we assign the dictionary `registry` followed by `.get`— without round brackets! In general, `sorted()` can be used as an alternative to the list method `.sort()`. The difference is that **`sorted()` returns a new list** (e.g., `sorted_list = sorted(original_list)`), whereas **`.sort()` modifies the existing list** (e.g., `original_list.sort()`).

Recap

- Some typical examples of dictionary use include counting elements, compressing information, and sorting a dictionary according to keys and values.
- The dictionary method `.get(key, initial value)` can be used to initialize an item in a dictionary and fill it up during a `for` loop.
- The built-in function `sorted()` is used to sort a dictionary; note that it creates a new variable.

Remaining dictionary methods

Dictionaries have 11 methods. In the past three chapters, we have learned six dictionary methods: `.items()`, `.keys()`, `.values()`, `.get()`, `.update()`, and `.pop()`. Here are the remaining 5 methods:

- `.clear()`: Deletes all the elements from the dictionary (makes the dictionary empty).
- `.copy()`: Provides a copy of the dictionary and thus allows separate modification.
- `.fromkeys()`: Creates a dictionary with the keys specified in a list and a default value.
- `.popitem()`: Removes the last inserted item.
- `.setdefault()`: Returns the value of the specified key. If the key does not exist, then it inserts the new item into the dictionary.

Create your examples

In a notebook, write an example for each of the new dictionary methods introduced in the *In more depth* section above: `.clear()`, `.copy()`, `.fromkeys()`, `.popitem()`, and `.setdefault()`. If you want, you can start from this dictionary:

```
fruit_colors = {"strawberry": "red", "banana": "yellow", "kiwi": "green"}
```

Let's code!

1. *From dictionary to list of lists and back!* Given the following dictionary:

   ```
   cars = {"sports car": 4, "convertible": 5, "limousine": 2}
   ```

   a. Transform the dictionary into a list of lists.
      (Expected result:`[['sports car', 4], ['convertible', 5], ['limousine', 2]]`).
   b. Transform the list of lists back to the original dictionary.

2. *Multiplication table game*! You are a programmer at an educational game company. Your task is to create a game where a kid enters a number, and you display the corresponding multiplication table. To implement the game, create a dictionary where the keys are numbers from 1 to 10 and the values are the results of the multiplications between the key and the value entered by the kid. Use a `for` loop and allow the kid to play as long as they want.
   (Example input: `4`
   Example output:
   `1 x 4 = 4`
   `2 x 4 = 8`
   `3 x 4 = 12`
   `4 x 4 = 16`
   `5 x 4 = 20`
   `6 x 4 = 24`
   `7 x 4 = 28`
   `8 x 4 = 32`
   `9 x 4 = 36`
   `10 x 4 = 40`).

3. *Spices and herbs*. You work in a grocery store selling spices and herbs. Here are the spices and herbs in the shop:

   ```
   spices_herbs = ["basil", "cinnamon", "licorice", "mint", "rosemary", "thyme", "cardamom",
   "turmeric", "cilantro", "oregano", "pepper", "chili", "dill", "cayenne pepper", "ginger",
   "garlic", "marjoram", "nutmeg", "sage", "saffron", "star anise", "bay leaves"]
   ```

   a. You have to change the labels on the containers and give them a more modern look. The length of the new labels is proportional to the length of the word written on it. Create a dictionary where keys are word lengths and values are lists of words with that length.
   b. You need to know how many labels you have to cut for each length. Create another dictionary where keys are word lengths in an ascending order, and values are the number of labels you have to cut for each length.
   c. What is the most common label? How many letters does it correspond to? Compute it!

27. Overview of strings

Operations, methods, and printing

In this chapter, we will summarize the characteristics of strings, similar to what we did for lists in Chapter 21. You'll notice a lot of commonalities between the two data types, but also some important differences. Consider this chapter as a comprehensive guide that you can return to whenever you have questions or doubts about strings. Let's start! Follow along with Notebook 27. As usual, try to solve the tasks before looking into the solution.

1. String slicing

String slicing works like list slicing (see Chapter 12). Take a look at the two examples below as a refresher.

- Given the following string:

```
[1]: 1 two_ways = "rsecwyadrkd"                    two ways is assigned some characters
```

We start with a string of characters (line 1). You might remember that in coding we use the word **characters** instead of letters.

- Extract every second character:

```
[2]: 1 print(two_ways[:,:,2])                      print two ways in positions from the beginning of the
                                                   list to the end of the list with a step of two
     reward
```

The start is the beginning of the string, so we can omit it. Similarly, the stop is the end of the string, so we can omit it too. The step is `2`. The outcome is `reward` (line 1).

- Extract every second character and invert the outcome:

```
[3]: 1 print(two_ways[:,:,-2])                     print two ways in positions from the beginning of the
                                                   list to the end of the list with a step of minus two
     drawer
```

Opposite to the above, the start is the end of the string, and the stop is the beginning of the string; therefore, we can omit both. Since we are going backwards, the step is `-2` (note the minus symbol). In this case, the outcome is `drawer`. (Did you know that the reverse of `reward` is `drawer`?)

2. "Arithmetic" operations on strings

There are two "arithmetic" operations on strings: concatenation and replication. They follow the same principles as for lists. Let's have a quick look.

2.1 String concatenation

- Concatenate two strings:

```
[4]:
1 first = "sun"
2 second = "screen"
3 combo = first + second
4 print(combo)
sunscreen
```

```
first is assigned sun
second is assigned screen
combo is assigned first concatenated with second
print combo
```

Given two separate strings—`"sun"` (line 1) and `"screen"` (line 2)—we can merge them using the concatenation operator `+` to obtain `"sunscreen"` (line 3). We print the result to check for correctness (line 4).

2.2 String replication

- Replicate a string 5 times:

```
[5]:
1 one_smile = ":-)"
2 five_smiles = one_smile * 5
3 print(five_smiles)
:-):-):-):-):-)
```

```
one smile is assigned smiley face
five smiles is assigned one smile replicated by five
print five smiles
```

Given a string containing some characters—for example, a smiley face (line 1)—we replicate it by using the replication symbol `*` and the number of times we want to replicate (`5` in this case; line 2). Finally, we print the obtained five smileys (line 3).

3. Replacing or removing substrings

Substrings are parts of strings. In many of the following examples, we will use substrings composed of only one character for simplicity. However, the rules in the examples are also valid for substrings composed of multiple characters. Let's learn how to replace or remove substrings in a string based on a substring position or content.

3.1 Changing a substring based on its position using slicing and concatenation

- Given the following string:

```
[6]:
1 favorites = "I like cooking, my family,
  and my friends"
```

```
favorites is assigned I like cooking, my
family, and my friends
```

We start with a string containing a sentence (line 1).

- Replace the character at position `0` with `"U"` using slicing and assignment. What happens?

```
[7]:
1 favorites[0] = "U"
---------------------------------------
TypeError          Traceback (most recent call last)
Cell In[7], line 1
----> 1 favorites [0] = "U"
TypeError: 'str' object does not support item assignment
```

```
favorites in position zero is assigned U
```

Why do we get the type error: `'str' object does not support item assignment`? Because in Python **strings are immutable**, that is, **they cannot be changed by assignment**. To change a string—or parts of it—we have to use slicing combined with concatenation or string methods. Let's have a look.

- Redo the same task using slicing and concatenation:

```
[8]: 1 from_position_one = favorites[1:]
     2 favorites = "U" + from_position_one
     3 print(favorites)
U like cooking, my family, and my friends
```

```
from position one is assigned favorites in
positions from one to the end of the string
favorites is assigned U concatenated with
from position one
print favorites
```

The first way to change a substring is to use a combination of slicing and concatenation. We slice the part of the string that we want **to keep**, that is, from the character in position `1`—the space after `I`—to the end, and we save it in the variable `from_position_one` (line 1). Then, we concatenate the desired character in position `0`—that is, `"U"`—to the string `from_position_one` (line 2). Obviously, we can compress the two lines of code into one line: `favorites = "U" + favorites[1:]`—here they are separated for clarity of explanation. Finally, we print the resulting string (line 3).

3.2 Changing a substring based on its position using the method `.split()` and concatenation

- Redo the same task using the string method `.split()`:

```
[9]: 1 favorites = "I like cooking, my family, and
     my friends"
     2
     3 parts = favorites.split("I")
     4 print(parts)
     5
     6 favorites = "U" + parts[1]
     7 print(favorites)
['', ' like cooking, my family, and my friends']
[U like cooking, my family, and my friends]
```

```
favorites is assigned I like cooking,
my family, and my friends

parts is assigned favorites dot split I
print parts

favorites is assigned U concatenated
with parts in position one
print favorites
```

The second way to change a substring is to combine the method `.split()` and concatenation. We start with the string to modify (line 1)—we need to rewrite the original string because we changed it in the previous cell. Then, we apply the method **`.split()`**, whose argument is **the substring around which we want to split the original string**—in our case, the character `"I"`—and we assign the output to the variable `parts` (line 3). As we can see from the print of `parts` (line 4), `.split()` returns a list with two elements. The first element contains the characters that are before the argument `"I"`—that is, an empty string because `"I"` is in position `0`. The second element represents the characters that are after the argument `"I"`—that is, `' like cooking, my family, and my friends'` (notice the space in the first position). As another example, if we want to split the string at the word `"cooking"`, we can write: `parts = favorites.split("cooking")`, and we obtain: `['I like ', ', my family, and my friends']`. We conclude the string modification by concatenating `"U"` with the second element in `parts` (line 6), and we print the final result (line 7).

3.3 Modifying a substring based on its content using the method `.replace()`

- Replace the commas with semicolons using the string method: `.replace()`:

```
[10]: 1 favorites = "I like cooking, my family, and my friends"
      2 favorites = favorites.replace(",", ";")
      3 print(favorites)
```

```
I like cooking; my family; and my friends
```

```
favorites is assigned I like cooking, my family, and my friends
favorites is assigned favorites dot replace comma semicolon
print favorites
```

We start by rewriting the original string (line 1). Then, we use the method **`.replace()`**, which takes two arguments: **the substring that we want to remove**, and **the substring that we want to add**. Note that we reassign the outcome to the original string `favorites` to make the change effective (line 2). Finally, we print to check for correctness (line 3).

3.4 Removing a substring based on its position using slicing and concatenation

To remove a substring based on its position, we can just use a combination of slicing (or `.split()`) and concatenation. For example: if we want to remove `cooking` from the string `favorites`, we can write: `favorites = favorites[:6] + favorites[15:]`, and we get: `I like my family, and my friends`.

3.5 Removing a substring based on its content using the method `.replace()`

To remove a substring based on its content, we need to use a trick. Let's have a look at it!

- Remove the commas:

```
[11]: 1 favorites = "I like cooking, my family, and my friends"
      2 favorites = favorites.replace(",", "")
      3 print(favorites)
```

```
I like cooking my family and my friends
```

```
favorites is assigned I like cooking, my family, and my friends
favorites is assigned favorites dot replace comma empty string
print favorites
```

After rewriting the original string (line 1), we use the method `.replace()`, where the first argument is a comma—the substring we want to remove—and the **second argument is an empty string**. With this trick, **we remove the unwanted substring and we do not substitute it with any new substring** (line 2). Finally, we print `favorites` as a check (line 3). Fun fact: How does the meaning of the sentence change when you remove the comma?

4. Searching a substring in a string

How do we find a substring in a string? Let's see below!

4.1 Searching for an existing substring using the method `.find()`

- Given the following string:

```
[12]: 1 us = "we are"
```

```
us is assigned we are
```

We start with a short string named `us` (line 1).

- Find the positions of the character `"e"` using the method `.find()`:

```
[13]:
1  positions = us.find("e")
2  print(positions)
1
```

```
positions is assigned us dot find e
print positions
```

We use the method **`.find()`** that **takes the substring that we want to find as an argument**—in our case, `"e"`. We assign the outcome to the variable `positions` (line 1) and we print it (line 2). Anything unexpected in the outcome? We get only the position `1`, whereas in `us`, `"e"` is at positions `1` and `5`. This happens because the method `.find()` returns **only the position of the first substring** that it finds. How can we find the position of all substrings `"e"`? Try to answer this question before looking into the solution below!

- Find the positions of the character `"e"` using an alternative way:

```
[14]:
1  # initializing positions
2  positions = []
3
4  # find all positions of e
5  for i in range(len(us)):
6      if us[i] == "e":
7          positions.append(i)
8  print(positions)
[1, 5]
```

```
initializing positions
positions is assigned empty list

find all positions of e
for i in range len us
if us in position i equals e
positions dot append i
print positions
```

We initialize the variable `positions`—which will contain the positions of all the substrings `"e"`—as an empty list (line 2). Then, we create a `for` loop through indices to browse all the positions in `us` (line 5). If the character at the current position `i` is equal to `"e"` (line 6), then we append `i` to the list `positions` (line 7). Finally, we print `positions` to check for correctness (line 8). Note that we used the same approach to find the positions of a repeated element in a list (Chapter 21).

4.2 Searching for a nonexistent substring using the method `.find()`

What happens if we look for a substring that is not in the string? Let's have a look!

- Find the positions of the character `"f"` using the method `.find()`:

```
[15]:
1  positions = us.find("f")
2  print(positions)
-1
```

```
positions is assigned us dot find f
print positions
```

Similarly to cell 13, we use the method `.find()` to look for the substring `"f"` in the string `us`, and we assign the outcome to the variable `positions` (line 1). Then, we print `positions` (line 2). The outcome is `-1`. Thus, **when we search for a substring that is not in the string**, **`.find()` returns `-1`**. This is a trick that is often used in conditions, such as: `if us.find("f") == -1: print("Character not found!")`.

5. Counting the number of substrings in a string

- Given the following string:

```
[16]:
1  hobbies = "I like going to the movies,
   traveling, and singing"
```

```
hobbies is assigned I like going to the
movies, traveling, and singing
```

We start with a string containing text about hobbies (line 1).

- Count the numbers of substrings `"ing"` using the method `.count()`:

```
[17]: 1 n_substrings = string.count("ing")
      2 print(n_substrings)
      4
```

```
n substrings is assigned string dot count ing
print n substrings
```

We use the **method `.count()`** which takes **the substring whose occurrence we want to count**—in our case `"ing"`—**as an argument**, and we save the outcome in the variable `n_substrings` (line 1). Then we print the result (line 2). The substring is present 4 times: `I like going to the movies, traveling, and singing`.

6. String to list and back

It can be convenient to separate the words in a string into list elements or to merge strings that are elements of a list into a single string. Let's see how to do both operations.

6.1 From list to string using the method `.split()`

- Given the following string:

```
[18]: 1 string = "How are you"
```

```
string is assigned How are you
```

We start with a string containing three words: `How`, `are`, and `you` (line 1).

- Transform the string into a list of strings where each element is a word:

```
[19]: 1 list_of_strings = string.split()
      2 print(list_of_strings)
      ['How', 'are', 'you']
```

```
list of strings is assigned string dot split
print list of strings
```

Words are separated by spaces. Thus, we can use the method `.split(" ")` with a space as an argument. However, an empty string `" "` is the default argument for `.split()`, thus we can omit it—in other words, writing `.split()` is equivalent to writing `.split(" ")`. We assign the outcome to the variable `list_of_strings` (line 1), and we print it (line 2). As you can see, `list_of_strings` is a list containing three elements, each of them corresponding to one of the words in `string`.

6.2 From string to list using the method `.join()`

How do we go back to a list? Let's learn it in the next cell!

- Transform the list of strings into a string using the method `.join()`:

```
[20]: 1 string_from_list = " ".join(list_of_strings)

      2 print(string_from_list)
      How are you
```

```
string from list is assigned space dot
join list of strings
print string from list
```

The method **`.join()` connects the elements of the list in the argument, separating them with the string it refers to**. In our case, the list in the argument is `list_of_strings`, which contains the three strings `"How"`, `"are"`, and `"you"`. The string to which `.join()` is applied is a space—that is, `" "` (line 1). The command might look odd at first because we apply the method directly to the string value—`" ".join()`. As an alternative, we could assign the space to a variable—`space = " "`—and then apply

the method to the variable—`space.join()`. To conclude the task, we print `list_of_strings` to check for correctness (line 2).

7. Changing character cases

There are several options when changing character cases. Let's have a quick look at them with the simple example below.

- Given the following string:

```
[21]: 1  greeting = "Hello! How are you?"          greeting is assigned Hello! How are you?
```

We start with a string where the first character of `"Hello"` and `"How"` are uppercase and all the other characters are lowercase.

- Modify the string to uppercase and lowercase; change to uppercase only the first character of the string, and then each word of the string; finally, invert the cases:

```
[22]: 1  # uppercase                                  uppercase
      2  print(greeting.upper())                      print greeting dot upper
      3  # lowercase                                  lowercase
      4  print(greeting.lower())                      print greeting dot lower
      5  # change the first character of the         change the first character of the string
         string to uppercase                          to uppercase
      6  print(greeting.capitalize())                 print greeting dot capitalize
      7  # change the first character of each word    change the first character of each word
         to uppercase                                 to uppercase
      8  print(greeting.title())                      print greeting dot title
      9  # invert cases                               invert cases
      10 print(greeting.swapcase())                   print greeting dot swapcase

HELLO! HOW ARE YOU?
hello! how are you?
Hello! how are you?
Hello! How Are You?
hELLO! hOW ARE YOU?
```

To change the string to uppercase, we use the method **`.upper()`** (line 2) and we get: `HELLO! HOW ARE YOU?`. Inversely, to change the string to lowercase, we use **`.lower()`** (line 4) and we obtain: `hello! how are you?`. To change to uppercase only the first character of the string, we use the method **`.capitalize()`** (line 6), and the string becomes `Hello! how are you?`, where only the `H` of `Hello` is uppercase. To change to uppercase the first characters of all the words, we use the method **`.title()`** (line 8). In our example, the outcome is `Hello! How Are You?`, where `H`, `H`, `A`, and `Y` are uppercase. Finally, to swap characters from uppercase to lowercase and vice versa, we use the method **`.swapcase()`** (line 10). We obtain: `hELLO! hOW ARE YOU?`, where `h` from `hELLO` and `h` from `hOW` are lowercase, and all the other characters are uppercase.

8. Printing variables

Printing is particularly useful in coding to check for correctness of operations and algorithms. In the previous chapters, we learned that the arguments of the built-in function `print()` can be either con-

catenated variables (Chapter 2), variables separated by commas (Chapter 25), or a string in combination with the method `.format()` (Chapter 25). Beyond refreshing these printing modalities and pointing out some peculiarities, we will learn f-strings and easier ways to better print numerical variables. Let's start!

8.1 Printing strings

- Given the following string:

```
[23]: 1 part_of_day = "morning"                    part of day is assigned morning
```

We start with the variable `part_of_day` containing the string `"morning"` as a value (line 1).

- Print `Good morning!` in 4 different ways, using (1) string concatenation, (2) comma separation, (3) the method `.format()`, and (4) f-strings:

```
[24]: 1 # (1) string concatenation                 (one) string concatenation
      2 print("Good " + part_of_day + "!")        print Good concatenated with part of day
                                                  concatenated with exclamation mark
      3 # (2) variable separation by comma        (two) variable separation by comma
      4 print("Good", part_of_day, "!")           print Good part of day exclamation mark
      5 # (3) the method .format()                (three) the method .format()
      6 print("Good {}!".format(part_of_day))     print Good placeholder exclamation mark
                                                  dot format part of day
      7 # (4) f-strings                           (four) f-strings
      8 print(f"""Good {part_of_day}!""")         print f Good part of day exclamation mark
Good morning!
Good morning !
Good morning!
Good morning!
```

To print `Good morning!` using concatenation, we concatenate `part_of_day` to two strings: `"Good "` and `"!"` (line 2). Note that `"Good "` contains a **space** as the last character; without that space, we would print `"Goodmorning!"`. As an alternative to concatenation, we can use comma separation—that is, we separate the variables by comma (line 4). Note that the printed line contains a space between `morning` and the exclamation mark (`morning !`). This happens because **in comma-separated printing, variables are always separated by a space**. Another way is to use the string method **`.format()`**, which **places its argument in the placeholder `{}` in the string** (line 6). In our case, the value of `part_of_day`—which is the argument of `.format()`—is positioned in the curly brackets in `"Good {}!"`. A last method is to use **f-strings**, where f stands for formatted. Within the round brackets of `print()`, we write: (1) `f`, (2) tree opening double quotes `"""`, (3) what we want to print, and (4) tree closing double quotes `"""`. In our case, what we want to print is composed of some characters (e.g., `Good` and `!`) and a **variable** that must be **enclosed in a pair of curly brackets**—that is, `{part_of_day}` (line 8).

8.2 Printing strings and numbers

- Given a string and a numerical variable:

```
[25]: 1 part_of_day = "morning"                    part of day is assigned morning
      2 time_of_day = 10                           time of day is assigned ten
```

We consider two variables: `part_of_day`—containing the string `"morning"` (line 1)—and `time_of_day`— containing the integer `10` (line 2).

- Print
  `Good morning!`
  `It's 10a.m.`
  using the same four methods above (note that the sentences are on two separate lines):

```
[26]:
1  # (1) string concatenation
2  print("Good " + part_of_day + "!\nIt's "
   + str(time_of_day) + "a.m.")

3  # (2) variable separation by comma
4  print("Good", part_of_day, "!\nIt's",
   time_of_day, "a.m.")
5  # (3) the method .format()
6  print("Good {}!\nIt's {}a.m."
   .format(part_of_day, time_of_day))

7  # (4) f-strings
8  print(f"""Good {part_of_day}!

9  It's {time_of_day}a.m.""")
```

```
Good morning!
It's 10a.m.
Good morning !
It's 10 a.m.
Good morning!
It's 10a.m.
Good morning!
It's 10a.m.
```

```
(one) string concatenation
print Good concatenated with part of
day concatenated with exclamation mark
backslash n It's concatenated with str
time of day concatenated with a.m.
(two) variable separation by comma
print Good part of day exclamation mark
backslash n It's time of day a.m.
(three) the method .format()
print Good placeholder exclamation mark
backslash n It's placeholder a.m. dot
format part of day time of day
(four) f-strings
print f three double quotes Good part of
day exclamation mark
It's time of day a.m. three double quotes
```

When using string concatenation (line 2), we have to consider a few aspects. First, we embed the escape character **`\n`**—which indicates newline—into the string `"!\nIt's"`, where `!` is the last character of the first line, and `It's` is the beginning of the second line. Second, since `time_of_day` is an integer, we need to **transform** it **into a string** by using the built-in function `str()` for concatenation. And finally, we have to leave a space after `Good` and `It's` to have the correct spaces in the printout. When using comma separation (line 4), the code looks similar to concatenation. However, we do not need to change the numerical variable `time_of_day` into a string. Also, we do not need to add spaces in the strings `"Good"` and `"!\nIt's"`. In the print, we can notice again that there is a **space** between `morning` and `!`, and between `10` and `a.m.`. When using `.format()`, we have to include two placeholders, one for `part_of_day` and one for `time_of_day`. Note that both variables are arguments of `.format()` (line 6). Finally, when using f-strings, we include the two variables directly in between their placeholders—that is, `{part_of_day}` and `{time_of_day}`. We also can go to the new line without having to write `\n`, but just by **writing the text on two consecutive lines** (lines 8–9).

8.3 Printing numbers with fewer decimals

- Given the numerical variable:

```
[27]:
1  number = 1.2345
```

```
number is assigned one dot two three four five
```

We start with a variable containing a float with 4 decimals (line 1).

- Print: `The number is 1.23`—note only the first two decimals—using the four methods above:

[28]:
```
1 # (1) string concatenation
2 print("The number is " + str(round(number, 2)))
3 # (2) variable separation by comma
4 print("The number is", round(number, 2))
5 # (3) the method .format()
6 print("The number is {:.2f}".format(number))
7 # (4) f-strings
8 print(f"""The number is {number:.2f}""")
```

```
The number is 1.23
The number is 1.23
The number is 1.23
The number is 1.23
```

```
(one) string concatenation
print The number is concatenated
with str round number two
(two) variable separation by comma
print The number is round number
two
(three) the method .format()
print The number is colon dot two f
dot format number
(four) f-strings
print f three double quotes The
number is number colon dot two f
three double quotes
```

When using string concatenation (line 2) or comma separation (line 4), we can use the **built-in function `round()`**, which **takes two arguments: the variable that we want to round** (`number`) and **the number of decimals that we want to keep** (2). As we have seen previously, when using concatenation, we need to transform the numerical variable into a string, whereas we do not when using comma separation. When using `.format()`, we add `:.2f` in the placeholder, where `:` indicates the start of the formatted part and `.2f` specifies that we want to keep 2 floating digits after the dot (line 6). A similar formatting is present in f-strings, with the addition that before the colon `:`, we need to indicate the variable to print—that is, `number` (line 8).

At this point, you might wonder: **which of these four ways should I use when printing?** The answer is the one that you prefer! It is just recommended to use one single way thought your code for consistency.

In this chapter, we have summarized or introduced several ways of dealing with strings using concatenation, assignment, slicing, and methods. Strings have a total of 47 methods, and we have learned 11 of them so far. We will learn 6 more methods in Chapter 30. For the remaining methods, you can consult the appendix at the end of this chapter.

✎ Complete the table

In this chapter, you have learned or refreshed 11 string methods. Summarize what they do by completing the following table (continued on the next page).

String method	What it does
`.capitalize()`	
`.count()`	
`.find()`	
`.format()`	
`.join()`	
`.lower()`	
`.replace()`	
`.split()`	
`.swapcase()`	
`.title()`	
`.upper()`	

Recap

- In strings, slicing and the "arithmetic" operations (concatenation and replication) work the same way as for lists.
- Strings are immutable and thus assignment is not possible.
- Strings have 47 methods. Of these, the 11 methods learned so far are: `.capitalize()`, `.count()`, `.find()`, `.format()`, `.join()`, `.lower()`, `.replace()`, `.split()`, `.swapcase()`, `.title()`, and `.upper()`.
- There are at least four ways to combine strings and numerical variables when printing: concatenation, comma separation, method `.format()`, and f-strings.
- To round a number to a wanted number of decimals, we use the built-in function `round()`.

Escape characters

Escape characters are special characters that can be used when creating strings or when printing. Let's see some examples:

- `\n` (newline): As you already know, it is used to print a new line. All the characters or variables after `\n` will be printed on a new line. For example:

```
[1]:
1 print("Shopping list:\napples\noranges")
Shopping list: backslash n apples backslash n oranges

Shopping list:
apples
oranges
```

- `\t` (horizontal tab): It is used to create a tab, that is, to indent text towards the right. For example:

```
[2]:
1 print("Dear friend,")
2 print("\tI hope you are doing fine. I have some news...")
print Dear friend,
print backslash t I hope you are doing fine. I have some news...

Dear friend,
        I hope you are doing fine. I have some news...
```

- `\"` (double quote): It is used when you need to print double quotes in string delimited by double quotes. For example:

```
[3]:
1 print("The wise said: \"To live a happy life...\"")
print The wise said: backslash quotes To live a happy life... backslash quotes

The wise said: "To live a happy life..."
```

- `\'` (single quote or apostrophe): Similarly to above, it is used to print a single quote when the string is enclosed in between single quotes. For example:

```
[4]:
1 print('It\'s the best time!')
print It backslash apostrophe s the best time!

It's the best time!
```

Let's code!

1. *Famous quotes.* Given the following string:

   `quote = "The future belongs to those who believe in the beauty of their dreams - Eleanor Roosevelt"`

   Use string methods to:

   a. Remove `"to those who"`.
   b. Replace `"belongs"` with `"until"`.
   c. Add `"seems impossible"` after `"future"`.
   d. Remove `"The future"`.
   e. Replace `"believe in the beauty of their dreams"` with `"it's done"`.

   f. Replace `"Eleanor Roosevelt"` with `"Nelson Mandela"`.
   g. Add `"It always"` at the beginning of the string.

   What quote will you get at the end? Make sure that words are separated by spaces.

2. *Commonalities.* Given the following strings:

   ```
   dessert = "lemon meringue pie"
   sweet = "honeypot"
   ```

   a. What characters do the two strings have in common? Save the common characters in a list.
   b. How many times do the common characters appear in `dessert`? Save the result in a dictionary created in two different ways, that is, using (1) a dictionary method and (2) a string method.

3. *Palindromes.* Palindromes are words that read the same backward as forward, such as *anna* or *madam*. Given the following list of strings:

   ```
   words = ["noon", "dog", "dad", "elephant", "jungle", "otto", "night", "bright", "kayak",
   "yeah", "wow"]
   ```

   Save palindrome words in a new list of strings. *Hint*: Consider using string slicing.

Appendix: String methods

In the following table, you can find all the 47 string methods available in Python. The methods with an asterisk are presented in this book—including the ones that will be introduced in Chapter 30.

String method	*What it does*
`.capitalize()*`	Converts the first character to uppercase and all the others to lowercase. E.g.: `print("hello".capitalize())` returns: `Hello`
`.casefold()`	Converts a string into lowercase. Differently from `.lower()`, it can handle complex cases. E.g.: `print("Straße".casefold())` returns: `"strasse"`. `print("Straße".lower())` returns: `"straße"` (Straße is street in German)
`.center()`	Returns a string centered within a given number of characters. E.g.: `print("hi".center(6))` returns: `"  hi  "`
`.count()*`	Returns the number of times a specified value is present in a string. E.g.: `print("singing".count("ing"))` returns: `2`
`.encode()`	Returns an encoded version of the string using the specified encoding—encodings define how characters are rendered on a screen. E.g.: `print("hello".encode(encoding='utf-8'))` returns: `.b'hello'`
`.endswith()`	Returns `True` if the string ends with the specified value. E.g.: `print("hello".endswith('lo'))` returns: `True`
`.expandtabs()`	Make the tabs in the string of the length defined by the arguments. E.g.: `print("h\te\tl\tl\to".expandtabs(3))` returns: `h  e  l  l  o`
`.find()*`	Return the first position of a substring. E.g.: `print("singing".find("ing"))` returns: `1`
`.format()*`	Formats the string using the specified arguments. E.g.: `print("Hello, {}".format("how are you?"))` returns: `Hello, how are you?`
`.format_map()`	Formats specified values—defined in a dictionary—in a string. E.g.: `print("My dog name is age years old".format_map({"name": "Ninja", "age": 7}))` returns: `My dog Ninja is 7 years old`
`.index()*`	Finds the first substring of a substring. E.g.: `print("hello".index("l"))` returns: `2`
`.isalnum()*`	Checks if all characters in the string are alphanumeric. E.g.: `print("123hello".isalnum())` returns: `True`
`.isalpha()*`	Checks if all characters in the string are alphabetic. E.g.: `print("hello".isalpha())` returns: `True`
`.isascii()`	Checks if all characters in the string are ASCII. E.g.: `print("ë".isascii())` returns: `False`
`.isdecimal()`	Checks if all characters in the string are decimals. E.g.: `print("123".isdecimal())` returns: `True`
`.isdigit()*`	Checks if all characters in the string are digits. E.g.: `print("123".isdecimal())` returns: `True`
`.isidentifier()`	Checks if the string is a valid identifier, that is, if it only contains alphanumeric letters (a–z and 0–9) or underscores (_), and it does not start with a number nor contain spaces. E.g.: `print("my string".isidentifier())` returns: `False`
`.islower()*`	Checks if all characters in the string are lowercase. E.g.: `print("hello".islower())` returns: `True`
`.isnumeric()`	Checks if all characters in the string are numeric. E.g.: `print("123".isnumeric())` returns: `True`
`.isprintable()`	Checks if all characters in the string are printable. E.g.: `print("\n".isprintable())` returns: `False`
`.isspace()`	Checks if all characters in the string are space. E.g.: `print(" ".isspace())` returns: `True`
`.istitle()*`	Checks if the string is title-cased. E.g.: `print("Hello, How Are You?".istitle())` returns: `True`
`.isupper()*`	Checks if all characters in the string are uppercase. E.g.: `print("HELLO".isupper())` returns: `True`

`.join()*`	Joins the strings of a list with the specified separator. E.g.: `print(", ".join(["hello", "hi"]))` returns: `hello, hi`
`.ljust()`	Left-justifies the string. E.g.: `print("hello".ljust(10, '-'))` returns: `hello-----`
`.lower()*`	Converts the string to lowercase. E.g.: `print("HELLO".lower())` returns: `hello`
`.lstrip()`	Removes characters at the beginning of the string (`l` is for left). E.g.: `print("hhello".lstrip("h"))` returns: `ello`
`.maketrans()`	Transforms the transformation of the characters of the first argument into the characters of the second argument. To print the outcome, we need to use the method `.translate()`. E.g.: `transformation = "bake".maketrans("b","c");` `print("bake".translate(transformation))` returns: `cake`
`.partition()`	Partitions the string into tuple elements. E.g.: `print("hello, how are you?".partition(" "))` returns: `('hello,', ' ', 'how are you?')`
`.removeprefix()`	Removes the specified prefix from the string. E.g.: `print("hello, how are you?".removeprefix("hello, "))` returns: `how are you?`
`.removesuffix()`	Removes the specified suffix from the string. E.g.: `print("hello, hi".removesuffix(", hi"))` returns: `hello`
`.replace()*`	Replaces substrings in strings. E.g.: `print("Hello, how is she?".replace("she", "he"))` returns: `Hello, how is he?`
`.rfind()`	Finds the last substring of a string (`r` is for right). E.g.: `print("hello".rfind('l'))` returns: `3`
`.rindex()`	Finds the last substring of a string. E.g.: `print("hello".rindex('ll'))` returns: `2`
`.rjust()`	Right-justifies the string. E.g.: `print("hello".rjust(10, '-'))` returns: `-----hello`
`.rpartition()`	Partitions the string into tuple elements starting from the end. E.g.: `print("hello, how are you?".rpartition(" "))` returns: `('hello,', ' ', 'how are you?')`
`.rsplit()`	Splits the string from the end. E.g.: `print("hello, how are you".rsplit(","))` returns: `['hello', ' how are you']`
`.rstrip()`	Removes characters from the end of the string. E.g.: `print("!!!hello!!!".rstrip("!"))` returns: `!!!hello`
`.split()*`	Splits the string into a list. E.g.: `print("hello, how are you?".split(","))` returns: `['hello', ' how are you?']`
`.splitlines()`	Splits the string at line breaks. E.g.: `print("hello\nhow are you?".splitlines())` returns: `['hello', 'how are you?']`
`.startswith()`	Checks if the string starts with the specified prefix. E.g.: `print("hello, how are you?".startswith("hello"))` returns: `True`
`.strip()`	Removes characters on the left and on the right. E.g.: `print("!!!hello!!!".strip("!"))` returns: `hello`
`.swapcase()*`	Swaps the case of all characters in the string. E.g.: `print("Hello, How Are You?".swapcase())` returns: `hELLO, hOW aRE yOU?`
`.title()*`	Converts the string to title case. E.g.: `print("hello, how are you?".title())` returns: `Hello, How Are You?`
`.translate()`	Maps the character of a string through a given translation table (see `.maketrans()`). E.g.: `transformation = "bake".maketrans("b","c");` `print("bake".translate(transformation))` returns: `cake`
`.upper()*`	Converts the string to uppercase. E.g.: `print("hello".upper())` returns: `HELLO`
`.zfill()`	Adds `0` at the beginning of the string until the string reaches the defined length. E.g.: `print("5".zfill(4))` returns: `0005`

PART 8
FUNCTIONS

In this part, you'll apply the coding syntax and computational thinking you have learned so far to build reusable units of code called functions. Let's dive in!

28. Printing thank you cards

Function inputs

To this point, you've learned Python data types—lists, strings, Booleans, dictionaries, integers, and floats. You've also learned how to combine these data types with operators—assignment, membership, arithmetic, comparison, and logical—to write commands, `if/else` conditions, `for` loops, and `while` loops. The next step is to learn how to combine these elements into units of code called functions. You are already somewhat familiar with functions because we have frequently used Python built-in functions, such as `print()`, `len()`, `range()`, etc. In this Part, you will learn what's behind functions and how to write them. Let's begin in this chapter by learning the components of a function and how to provide inputs. Let's start! Follow along with Notebook 28.

1. Basic thank you cards

- You recently hosted a party, and you want to send thank you cards to those who attended. Create a function that takes a first name as an argument and prints a thank you message containing an attendee's name (e.g., `Thank you Maria`):

```
[]:
1  def print_thank_you(first_name):
2      """Prints a string containing
       "Thank you" and a first name
3
4      Parameters
5      ----------
6      first_name : string
7          First name of a person
8      """
9
10     print("Thank you", first_name)
```

```
def print thank you first name
Prints a string containing Thank you and
a first name

Parameters

first name : string
First name of a person


print Thank you first name
```

- Print two thank you cards:

```
[]:
1  print_thank_you("Maria")
```

```
print thank you Maria
```

```
[]:
1  print_thank_you("Xiao")
```

```
print thank you Xiao
```

What can you deduce about functions from this example? Get some hints by completing the following exercise!

✎ True or false?

1. `def` is the keyword that introduces a function definition. T F
2. `first_name` is a function input. T F
3. Function documentation is enclosed in single double quotes. T F
4. `print("Thank you", first_name)` is executed when we run the first cell. T F
5. `print_thank_you("Maria")` and `print_thank_you("Xiao")` are function calls. T F

Computational thinking and syntax

Let's start by analyzing how a function works. Let's run the first cell:

```
[1]:
def print_thank_you(first_name):
    """Prints a string containing
    "Thank you" and a first name

    Parameters
    ----------
    first_name : string
        First name of a person
    """

    print("Thank you", first_name)
```

```
def print thank you first name
Prints a string containing Thank you and
a first name

Parameters

first name : string
First name of a person

print Thank you first name
```

What happens? Apparently nothing! Let's run the two following cells:

```
[2]:
print_thank_you("Maria")
```

```
Thank you Maria
```

```
print thank you Maria
```

```
[3]:
print_thank_you("Xiao")
```

```
Thank you Xiao
```

```
print thank you Xiao
```

For each cell, a message is printed that says `"Thank you"` followed by the name of a person—that is, `"Maria"` in cell 2 and `"Xiao"` in cell 3. So, how do functions work?

In cell 1, there is a function **definition**, which specifies **what a function does**. When we run cell 1, we just tell our computer to **"memorize" the function**. To actually **execute the function**—that is, to make it do what we want it to do—we must **call the function**, which is what we do at cells 2 and 3—each of them containing **a function call**. If we do not call a function, then the function will never be executed!

How do we get `"Thank you Maria"` after running cell 2 and `"Thank you Xiao"` after running cell 3? Let's understand it with the help of Figure 28.1.

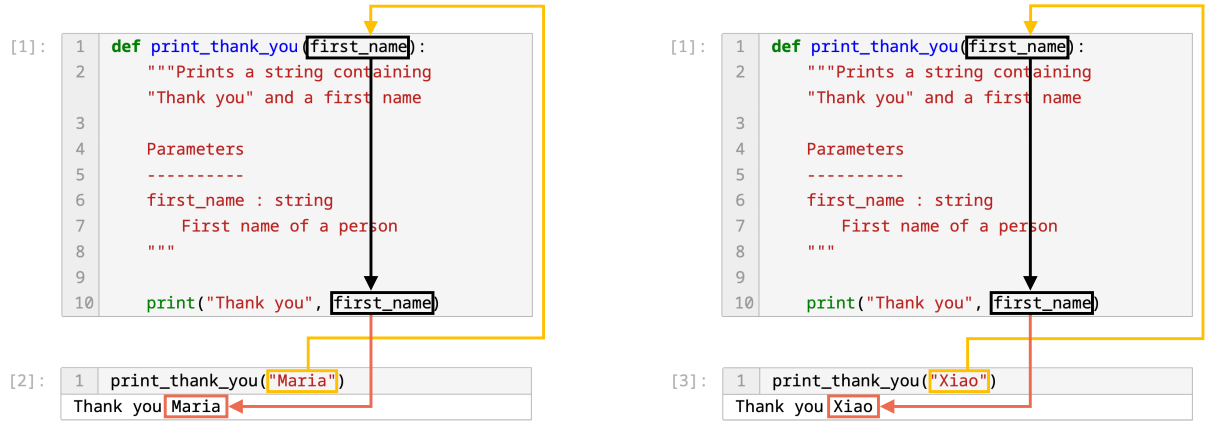


Figure 28.1. Path of a function input.

Let's start by understanding how we obtain the printed text `"Thank you Maria"` (left side of Figure 28.1). In cell 2, we provide the string `"Maria"` to the function call as an **input** —i.e., `print_thank_you("Maria")`.

When we run the cell, `"Maria"` passes from the call to the variable `first_name` located in the function header in cell 1, line 1 (yellow arrow). The variable `first_name`—which now contains the value `"Maria"`—is then used in the command at line 10 (black arrow), which produces the print `"Thank you Maria"` (orange arrow). Let's now see how we get `"Thank you Xiao"` (right side of Figure 28.1). Similarly to before, in cell 3, we call the function `print_thank_you()` with the string `"Xiao"` as an input. When we run the cell, the value `"Xiao"` is assigned to the variable `first_name` in the function header (yellow line), then used in the function command (black arrow), and finally printed to screen (orange line). In summary, when we call a function, we **pass** the variables **from a function call** (cell 2 or 3) to a **function definition** (cell 1) as an **input**. Then, the variable will be **used in the function commands**. To be more precise, in Python, we call the input variable **parameter** when it is **in the function definition**—`first_name` in cell 1 is a parameter—and **argument** when it is **in the function call**—`"Maria"` in cell 2 is an argument, as well as `"Xiao"` in cell 3. Finally, note that the same mechanism applies when we call any Python built-in function. For example, when we write `len("hello")`, we pass the argument `"hello"` to the definition of `len()`, which has a syntax similar to the function in cell 1 and contains commands that count the number of characters.

Let's now look into function syntax. In cell 1, we define the function `print_thank_you()`. Any **function definition**—which we usually just call a **function**—is composed of **two parts**: a header (line 1) and a body (lines 2–10). The **header** is made of: (1) **keyword `def`**, (2) function name, (3) parameters embedded in round brackets, and (4) a colon (line 1). **Function names** follow the same rules as variable names, that is, they are lowercase and the words that compose them are separated by an underscore.

A function **body** contains two components: (1) documentation (lines 2–8) and (2) code (line 10), and it is always **indented** with respect to the header. In Python, the **documentation** is embedded in between double quotes repeated three times (`"""`; lines 2 and 8) and is called **docstring**, which is a compact word for "documentation string". A function documentation can follow various styles, and in this book we will use the **NumPy style**, which has the following structure:

- **Short summary** (line 2): A one line summary about what the function does. It is written next to the three opening double quotes.
- **Parameters** (lines 4–7): A description of the parameters—that is, the inputs—of the function. It contains: (1) the title `Parameters` (line 4), (2) a sequence of minus signs that act as an underline (line 5), and (3) a list of parameters (lines 6–7)—there is only one parameter in this example; there will be more in the coming examples. Each parameter is described on two consecutive lines. In the first line, we include: (1) a parameter name (`first_name`), (2) a space, (3) a colon, (4) a space, and (5) a parameter type (i.e., `string`) (line 6). In the second line, we write a short description of the parameter. Note that this line is indented.
- Other specifications that we will see in the next chapter.

Finally, the **code** component of a function **body** can contain as many lines of code as needed to execute the desired task—in this initial example, there is only one command (line 10).
Let's conclude this first example by providing a formal definition of function:

A **function** is a **block of code** that **accomplishes a specific task**

What about the inputs? Must a function have inputs? No, there can be functions **without inputs** (see exercise 4 in the *Let's code* section at the end of this chapter). Can a function contain **more than one input**? Yes! Let's look into the next example—the differences with the function in cell 1 are underlined.

2. Formal thank you cards

- After a second thought, you decide that it is more appropriate to print formal thank you cards. Modify the previous function to take three arguments—prefix, first name, and last name—and to print a thank you message containing them (e.g., `Thank you Mrs Maria Lopez`):

[4]:
```
1  def print_thank_you(prefix,
       first_name, last_name):
2      """Prints a string containing
       "Thank you" and the inputs
3
4      Parameters
5      ----------
6      prefix : string
7          Usually Ms, Mrs, Mr
8      first_name : string
9          First name of a person
10     last_name : string
11         Last name of a person
12     """
13
14     print("Thank you", prefix,
       first_name, last_name)
```

```
def print thank you prefix first name
last name
Prints a string containing Thank you and
the inputs

Parameters

prefix : string
Usually Ms, Mrs, Mr
first name : string
First name of a person
last name : string
Last name of a person


print Thank you prefix first name last
name
```

- Print two formal thank you cards:

[5]:
```
1  print_thank_you("Mrs", "Maria","Lopez")
```
```
Thank you Mrs Maria Lopez
```
```
print thank you Mrs Maria Lopez
```

[6]:
```
1  print_thank_you("Mr", "Xiao","Li")
```
```
Thank you Mr Xiao Li
```
```
print thank you Mr Xiao Li
```

Computational thinking and syntax

In this function with several inputs (cell 4), we observe two changes with respect to the same function with one single input (cell 1). First, in the function header, there are now three **parameters**—`prefix`, `first_name`, and `last_name`—which are **separated by comma**. Second, in the docstrings (lines 6–11), we describe each parameter **in the same order** as in the function header—that is, first `prefix`, then `first_name`, and finally `last_name`. Note that for each parameter we use the same syntax that we described above—that is, **parameter name and type in the first line**, and **parameter description in the second line**. How many parameters can we have in a function? As many as we want! The only requirement is that **all parameters must be used** within the function. In this example, the parameters are used in one single line of code to print the desired message (line 14), but in general, parameters can be used in one or more lines of code.

What about the function calls (cells 5 and 6)? When we **call** the function, we have to make sure that we insert the inputs **in the same order as in the function header**—that is, first `prefix`, then `first_name`, and finally `last_name`. What happens if one of the arguments is **missing** like in the example below?

[6]:
```
1 print_thank_you("Mr", "Xiao")
```
print thank you Mr Xiao

```
---------------------------------------
TypeError          Traceback (most recent call last)
Cell In[6], line 1
----> 1 print_thank_you("Mr", "Xiao")
TypeError: print_thank_you() missing 1 required positional argument: 'last_name'
```

We get an **error message** saying that the function is `missing 1 required positional argument: 'last_name'`. This means that we did not write the third argument in the call. How can we modify the function to avoid this error? Let's have a look at cells 7–9! Like before, the function modifications are underlined.

3. Last name missing!

- You are very happy with the thank you cards, but you suddenly realize that some participants did not provide their last names! Adapt the function so that the last name has an empty string as a default value:

[7]:
```
1  def print_thank_you(prefix,
       first_name, last_name=""):
2      """Prints a string containing
       "Thank you" and the inputs
3
4      Parameters
5      ----------
6      prefix : string
7          Usually Ms, Mrs, Mr
8      first_name : string
9          First name of a person
10     last_name : string
11         Last name of a person. The default
           value is an empty string
12     """
13
14     print("Thank you", prefix,
       first_name, last_name)
```
def print thank you prefix first name last name is assigned empty string
Prints a string containing Thank you and the inputs

Parameters

prefix : string
Usually Ms, Mrs, Mr
first name : string
First name of a person
last name : string
Last name of a person. The default value is an empty string

print Thank you prefix first name last name

- Print two thank you cards, one with a last name and one without a last name:

[8]:
```
1 print_thank_you("Mrs", "Maria", "Lopez")
```
print thank you Mrs Maria Lopez

```
Thank you Mrs Maria Lopez
```

[9]:
```
1 print_thank_you("Mr", "Xiao")
```
print thank you Mr Xiao

```
Thank you Mr Xiao
```

Computational thinking and syntax

In the function header, we **assign a default value to the input that can be missed** when calling the function. In our case, we assign an empty string to the variable `last_name` (line 1)—note the lack of space

before and after the assignment symbol, as recommended by the Python style guide. We call `last_name` **default parameter**, and we specify the default value in its description in the docstrings (line 11).

What happens when we call the function? If we provide all three arguments (cell 8), then the function works exactly like in cell 4—that is, `"Mrs"` is passed to `prefix`, `"Maria"` to `first_name`, and `"Lopez"` to `last_name`. If we provide only `prefix` and `first_name` but not `last_name` (cell 9), the function prints `"Mr"` for `prefix` and `"Xiao"` for `first_name`, and the default empty string for `last_name`—we do not see it printed! What if the missing parameter when calling the function in cell 4 is not the last one but, for example, the first one—i.e., `prefix`? Let's have a look:

```
[6]:  1  print_thank_you("Xiao", "Li")                    print thank you Xiao Li
```

```
---------------------------------------
TypeError                 Traceback (most recent call last)
Cell In[6], line 1
----> 1 print_thank_you("Xiao, "Li")
TypeError: print_thank_you() missing 1 required positional argument: 'last_name'
```

We skipped the prefix, but the error tells us that we skipped the last name! This is because the function always **assumes that the missing argument is the last one**. If we want to skip arguments in other positions—that is, `prefix` or `first_name`—then we have to make a final modification to our function, as you can see underlined in the code below.

4. Prefix and/or first name missing!

- Finally, you realize that prefix and/or first name are also missing for some guests. Modify the function accordingly:

```
[10]:
1  def print_thank_you(prefix="",
       first_name="", last_name=""):
2      """Prints each input and a string con-
       catenating "Thank you" and the inputs
3
4      Parameters
5      ----------
6      prefix : string
7          Usually Ms, Mrs, Mr. The default
           value is an empty string
8      first_name : string
9          First name of a person. The default
           value is an empty string
10     last_name : string
11         Last name of a person. The default
           value is an empty string
12     """
13
14     print("Prefix:", prefix)
15     print("First name:", first_name)
16     print("Last name:", last_name)
17     print("Thank you", prefix,
       first_name, last_name)
```

```
def print thank you prefix is assigned
empty string first name is assigned
empty string last name is assigned empty
string
Prints a string containing Thank you and
the inputs

Parameters

prefix : string
Usually Ms, Mrs, Mr. The default value
is an empty string
first name : string
First name of a person. The default
value is an empty string
last name : string
Last name of a person. The default value
is an empty string


print Prefix: prefix
print First name: first name
print Last name: last name
print Thank you prefix first name last
name
```

- Print a thank you card where the first name is missing:

```
[11]: 1 print_thank_you(prefix="Mrs", last_name="Lopez")
```

```
Prefix: Mrs
First name:
Last Name: Lopez
Thank you Mrs  Lopez
```

```
print thank you prefix is assigned Mrs last name is assigned Lopez
```

- Print a thank you card where the prefix is missing:

```
[12]: 1 print_thank_you(first_name="Xiao", last_name="Li")
```

```
Prefix:
First name: Xiao
Last Name: Li
Thank you  Xiao Li
```

```
print thank you first name is assigned Xiao last name is assigned Li
```

Computational thinking and syntax

In the function header, we **assign a default value to each parameter**—in our case, an empty string (cell 10, line 1)—and we add this information to the docstrings (lines 7 and 11). In this example, we also print each parameter to clarify what happens when we call the function (lines 14–16), as you will see in a bit.

What about the function calls? When we call `print_thank_you()` with the arguments `prefix="Mrs"` and `last_name="Lopez"` (cell 11), `first_name` is automatically assigned its default value, that is, an empty string—see the print from line 15. Similarly, when we call the function with the arguments `first_name="Xiao"` and `last_name="Li"` (cell 12), `prefix` is assigned the default empty string—see the print from line 14. In addition, in the print `Thank you Mrs  Lopez` (from line 17), there are two spaces between `Mrs` and `Lopez`. This occurs because when we print using comma separation, a space is automatically inserted between variables. Thus, one space separates `Mrs` and the first name—which is missing—and one space separates the first name and `Lopez`. In the same way, in the print `Thank you Xiao Li`, there is an extra space due to the absence of a prefix. What if we want to be precise and ensure that there is one single space between the variables? We could write an `if/elif/else` construct like the following: `if prefix == "": | print("Thank you", first_name, last_name) | elif first_name == "": | print("Thank you", prefix, last_name) | else: | print("Thank you", prefix, first_name)`.

Finally, do we always need to provide default values to the parameters in a function? Not necessarily, especially when there are no appropriate default values or when it's essential that all arguments are specified when calling the function.

Why do we create functions?

At this point, you might wonder, why do we need to create functions? Can we not just write the `print()` command whenever we need it? The functions in cells 1, 4, and 7 contain only one single line of code, so writing a function might seem unnecessary. However, consider the function in cell 10. It

has four lines of code, and if we want to reuse them in several cases, we have to keep copying and pasting. As you might remember, copy-pasting should be avoided not only because it is tedious but also because it increases the risk of errors. Grouping lines of code into a function is a very efficient way to **reuse code** across various parts of a project. In addition, functions help us **divide and conquer** tasks. Each function should contain commands that solve one specific subtask, allowing us to **modularize** our code—that is, breaking it into manageable chunks that are easier to read, modify, and reuse.

Recap

- Functions are blocks of code that accomplish a specific task. They are crucial for code reuse and modularization.
- A function comprises at least three components:
  - A header, which starts with the keyword `def`, followed by the name of the function, and round brackets containing the parameters separated by comma. Parameters can have default values.
  - Docstrings, which describe what the function does and its parameters.
  - Code that solves a task.
- To call a function, we write the function name followed by round brackets containing the arguments separated by comma.
- Parameters and arguments are function inputs. Technically, we call parameters the variables listed in the function header, and arguments the variables in the function call.
- Docstrings are fundamental when writing and using functions and can be accessed using the built-in function `help()`—see the *In more depth* section below.

Why is function documentation important?

Writing docstrings is fundamental **for both our future selves and for others** who may use our code. When we write a function, there is a good chance that we will need to reuse it months or even years later. Without function documentation, it could take us hours to recall what the function does or the types of its inputs—and outputs, as you will see in the next chapter. Investing a few minutes in writing clear documentation can save us countless hours in the future! Similarly, if somebody else needs to use our functions, they need to understand what the function does and the type and roles of its inputs and outputs. Have you ever tried to use an undocumented function? It can be incredibly frustrating!

How can we **access function documentation**? Do we always have to look at the function definition? Fortunately no! We can use the built-in function **`help()`**! For example, let's have a look at the documentation of the function `print_thank_you()` that we created earlier in this chapter. ↳

```
[1]: 1 help(print_thank_you)                                    help print thank you
Help on function print_thank_you in module __main__:

print_thank_you(prefix='', first_name='', last_name='')
    Prints each input and a string containing "Thank you" and the inputs

    Parameters
    ----------
    prefix: string
        Usually Ms, Mrs, Mr. The default is an empty string
    first_name: string
        First name of a person. The default is an empty string
    last_name: string
        Last name of a person. The default is an empty string
```

As you can see, `help()` displays the docstrings we wrote in cell 10. Notice that `help()` requires only the **function name as an argument**—without round brackets or parameters. Be aware that `help()` can be used for any functions, including Python built-in functions, like you can see here:

```
[2]: 1 help(len)                                                help len
Help on built-in function len in module builtins:

len(obj, /)
    Return the number of items in a container.
```

The description is a bit technical, but you can think of `containers` as data types like lists, strings, or dictionaries. So, `Return the number of items in a container` means that `len()` gives us the number of elements in a list, or characters in a string, or items in a dictionary, etc. We will learn more about what "returning" means in the next chapter.

Let's code!

1. *String cases.* Write a function that prints a given string in lower case, upper case, title case, capitalized, and with swapped cases. Then, call the function twice once using a string made of one word, and once using a string made of at least two words. Finally, call the function for each element of the following list of strings using a `for` loop:

   `summer_vacation = ["Hiking trails", "weekEnd campIng", "enjoying nature", "fishing"]`.
2. *String lengths.* Write a function that takes a list of strings and an integer, and prints only the strings whose length matches the given integer. Call the function using two different word lengths.
3. *Multiple numbers.* Write a function that takes a list of numbers and an integer, and prints only the numbers divisible by the integer. If the user does not provide a number, then the function divides by 2 by default. Call the function using two different divisors. Finally, call the function without a divisor.
4. *Doubling numbers.* Write a function that asks a user for a number and prints a dictionary where the keys are numbers up to the input number, and values are the double of each key. Note that the function does not take any argument. The `input()` function to ask for the number must be inside the function.
   (Example user input: `5`
   Expected print: `{1: 2, 2: 4, 3: 6, 4: 8, 5: 10}`)

29. Login database for an online store

Function outputs and modular design

In the previous chapter, we learned about functions and their inputs. In this chapter, we will dive into function outputs. In addition, we'll take a look at designing and organizing multiple functions in a larger project. Let's tackle all this by solving the following task. Follow along with Notebook 29!

- You are the owner of an online store and need to securely store the usernames and passwords of your customers. Create a database where usernames are composed of the initial of the customer's first name followed by their last name (e.g., `jsmith`), and passwords consist of a four-digit code.

First we have to create a database. A **database** is an organized **collection of data** that can be easily accessed and managed. Examples of databases include an inventory at a grocery store, a library catalog, or a phone contact list. In our case, the database will be a collection of customers' usernames and passwords. In general, simple databases can be implemented as **dictionaries**.

How would you create this database, and how would you insert usernames and passwords? What variables would you use and of what types? How many functions would you write, and what would each function do? Take some time to think about your solution before proceeding to the next paragraph!

To solve our task, the first thing to do is to divide and conquer by defining what variables and functions we need to create. Let's start with the **variables and their data types**. For each customer, we need two **strings**—one for the username and one for the password. We'll save them in a **dictionary**—that is, a database—where the usernames will be the keys and the passwords will be the values. Let's now think about how to **modularize** the code—that is, how to organize it into functions. We can write three **functions**: one to create a username, one to create a password, and one that calls the previous two functions and adds the created usernames and passwords to a database. Let's implement this solution!

1. Creating a username

Read the following text and code and try to deduce what the code does.

- Write a function that creates a username composed of the initial of the first name and the last name:

[1]:
```
1  def create_username(first_name, last_name):
2      """Creates a lowercase username made of
       initial of first name and full last name
3
4      Parameters
5      ----------
6      first_name : string
7          First name of a person
8      last_name : string
9          Last name of a person
10                                              ↳
```

```
def create username first name last name
Creates a lowercase username made of
initial of first name and full last name

Parameters

first name : string
First name of a person
last name : string
Last name of a person
```

```
    Returns                                  Returns
    -------                                  
    username : string                        username : string
        Created username                     Created username
    """                                      
                                             
    # concatenate initial of first name      concatenate initial of first name and
    and last name                            last name
    username = first_name[0] + last_name     username is assigned first name in
                                             position zero concatenated with last
                                             name
    # make sure the username is lowercase    make sure the username is lowercase
    username = username.lower()              username is assigned username dot lower
                                             
    # return username                        return username
    return username                          return username
```

- Create the username for two customers:

[2]:
```
username_1 = create_username("Julia", "Smith")   username one is assigned create
                                                 username Julia Smith
print(username_1)                                print username one
```
```
jsmith
```

[3]:
```
username_2 = create_username("Mohammed", "Seid")   username two is assigned create
                                                   username Mohammed Seid
print(username_2)                                  print username two
```
```
mseid
```

What's going on in the three cells above? Get some hints by solving the following exercise!

✎ True or false?

1. The function has three parameters. T F
2. In docstrings, we must specify the name, type, and description of the output (also called return), like we do for the parameters. T F
3. The username is composed of first name and last name. T F
4. return is the keyword used to return function outputs. T F

Computational thinking and syntax

In cell 1, there is a function that creates a username. It takes two parameters—`first_name` and `last_name` (line 1)—which are used to create a username in two consecutive steps. First, we concatenate the initial of the first name—that is, `first_name` in position `0`—with the last name, and we assign the result to the variable `username` (line 18). Then, we apply the method `.lower()` to `username` to ensure it is lowercase (line 20). What happens at line 23? We return `username`, meaning that we "push" `username` out of the function. Where does it go? Let's look into the function calls. In the first line of cell 2, we call the function `create_username()` with the arguments `"Julia"` and `"Smith"`. The two arguments are automatically passed to the function header (cell 1, line 1). In the function, the first

character of `"Julia"` and the whole string `"Smith"` are concatenated into `"JSmith"` and saved in the variable `username` (line 18). Next, `username` is modified to lowercase and becomes `"jsmith"` (line 20). At the end of the function, we **return** `username` (line 23)—that is, `"jsmith"` is sent out of the function—and we **assign** it to the variable `username_1` (cell 2, line 1). Finally, we print `username_1` (cell 2, line 2). Similarly, in the second function call (cell 3, line 1), we pass the arguments `"Mohammed"` and `"Seid"` to the function `create_username()` (cell 1, line 1), where the username `"mseid"` is created (lines 18 and 20). The username is then returned (line 23) to be assigned to the variable `username_2` (cell 3, line 1) and then printed (cell 3, line 2). As above, we use `return` to send a variable from a function body back to the function call. You can see the path of the output variables in Figure 29.1.

Figure 29.1. Path of a function output.

As is now clear, **`return`** is the **keyword** we use to **transfer output variables from the function body to the function call**. But it has another important property: it **marks the end of a function**. This means that any line of code written after `return` will never be executed!

You might have realized that you have already used numerous returned variables throughout our learning journey. For example, the Python built-in function `int(14.45)` returns `14`, which means that in the function `int()`, the last line of code is something similar to `return integer_number`. Similarly, the method `.lower()` applied to the string `"JSmith"` returns `"jsmith"` because the last line of code is something similar to `return lower_case_string`.

Finally, let's have a look at the **documentation** of the function in cell 1. As you can see, we specify the **returned variables** (lines 11–14). The syntax is the same as for the parameters (lines 4–9). First, we write `Returns` as a title (line 11), followed by a series of minus signs that act as an underline (line 12). Then, for each returned variable—in this example, there is only one—we write (1) variable name (e.g., `username`),

(2) space, (3) colon, (4) space, and (5) type (e.g., `string`) (line13). On the following line, indented, we write the definition of the returned variable (line 14).

2. Creating a password

We need to implement a function that creates a password composed of four integers. How would you do it? Try to implement it yourself before looking at the solution below.

- Write a function that creates a password composed of four random integers:

[4]:
```
1  import random
2
3  def create_password():
4      """Create a password composed of four
       random integers
5
6      Returns
7      -------
8      password : string
9          Created password
10     """
11
12     # create a random number with four digits
13     password = str(random.randint(1000, 9999))
14
15     # return password
16     return password
```

```
import random

def create password
Create a password composed of four
random integers

Returns

password : string
Created password


create a random number with four digits
password is assigned str random dot
randint 1000 9999

return password
return password
```

- Create the password for two customers:

[5]:
```
1  password_1 = create_password()
2  print(password_1)
```
```
4883
```

```
password one is assigned create password
print password one
```

[6]:
```
1  password_2 = create_password()
2  print(password_2)
```
```
5005
```

```
password two is assigned create password
print password two
```

To generate a password with four integers, we'll use a simple trick: we create a random number between 1000 and 9999, which is the range of all the existing numbers with four digits! Then, we transform the obtained number into a string—using the built-in function `str()`—and we assign the result to the variable `password` (cell 4, line 13). Why are we converting the four-digit integer into a string? Because a password does not have any numerical meaning—that is, we do not use it in arithmetic operations such as addition or multiplication. Finally, we return `password` at line 16. Note that this function **does not have any inputs**. Thus, there are no parameters in between the round brackets in the header (line 3), there is no `Parameters` section in the documentation (lines 4–10), and we do not write any arguments in between the round brackets when we call the function (cells 5 and 6, line 1). The returned variable `password` (cell 4, line 16) is saved as `password_1` and `password_2`, at line 1 of cells 5 and 6, respectively. Finally, we print the passwords to check for correctness (line 2 of cells 5 and 6).

3. Creating a database

- Write a function that, given a list of lists of customers, creates and returns a database—i.e., a dictionary—of usernames and passwords. The function also returns the number of customers in the database:

[7]:

```
1  def create_database(customers):
2      """Creates a database as a dictionary with
       usernames as keys and passwords as values
3
4      Parameters
5      ----------
6      customers : list of lists
7          Each sublist contains first name and
           last name of a customer
8
9      Returns
10     -------
11     db : dictionary
12         Created database (shorted as db)
13     n_customers : int
14         Number of customers in the database
15     """
16
17     # initialize dictionary (i.e. database)
18     db = {}
19
20     # for each customer
21     for customer in customers:
22
23         # create username
24         username = create_username(
           customer[0], customer[1])
25
26         # create password
27         password = create_password()
28
29         # add username and password to db
30         db[username] = password
31
32     # compute number of customers
33     n_customers = len(db)
34
35     # return dictionary and its length
36     return db, n_customers
```

```
def create database customers
Creates a database as a dictionary
with usernames as keys and passwords
as values

Parameters

customers : list of lists
Each sublist contains first name and
last name of a customer

Returns

db : dictionary
Created database (shorted as db)
n customers : int
Number of customers in the database


initialize dictionary (i.e. database)
db is assigned empty dictionary

for each customer
for customer in customers

create username
username is assigned create username
customer in position zero customer in
position one

create password
password is assigned create password

add username and password to db
db at key username is assigned password

compute number of customers
n customers is assigned len db

return dictionary and its length
return db n customers
```

Let's analyze the function before calling it in the cells below. Let's begin with the input and the outputs. The input is a variable called `customers`, as we can see in the function header (line 1). From the documentation, we learn that `customers` is a list of lists where each sublist contains a first name and a last name (lines 6–7). The outputs are two variables called `db` and `n_customers`, as we can see in the last

line of the function after the keyword `return` (line 36). From the documentation, we learn that `db` is a dictionary that will contain the database (lines 11–12), whereas `n_customers` is an integer that will store the number of customers in the database (lines 13–14). Let's continue with the analysis of the function body. We initialize the variable `db` as an empty dictionary (line 18), which we will fill out within the function and eventually return. Then, for each customer in the list of lists (line 21), we perform three actions. First, we create a username by calling the function `create_username()` that we wrote in cell 1. The inputs are the first name—`customer[0]`—and the last name—`customer[1]`—of the current customer. We save the output in the variable `username` (line 24). Then, we create the password by calling the function `create_password()` from cell 4, and we save the output in the variable `password` (line 27). Finally, we add the username and the password to the database by assigning the variable `password` as a value to the corresponding key `username` in the database `db` (line 30). Once we complete the creation of username and password for each customer and exit the loop, we calculate the number of customers, which corresponds to the length of the dictionary. We use the built-in function `len()`, and we save the output in the variable `n_customers` (line 33). Finally, we return both `db` and `n_customers` (line 36). As you can see, to **return multiple variables**, we write them **after the keyword `return` and separated by commas**.

It is always very important to **test the correctness of a function by calling it**. So let's call the function and test its behavior!

- Given the following list of customers:

[8]:
```
1 customers = [["Julia", "Smith"], ["Mohammed",
  "Seid"], ["Maria", "Lopez"]]
```

```
customers is assigned Julia,
Smith, Mohammed, Seid, Maria,
Lopez
```

We create a list of lists called `customers` that contains three sublists (cell 8).

- Create the database using two different syntaxes:

[9]:
```
1 # create the database - separate returns

2 database, number_customers =
  create_database(customers)
3
4 # print the outputs
5 print("Database:", database)
6 print("Number of customers:", number_customers)
```

```
create the database - separate
returns
database number customers is
assigned create database customers


print the outputs
print Database: database
print Number of customers: number
customers
```

```
Database: {'jsmith': '6891', 'mseid': '3189', 'mlopez': '7097'}
Number of customers: 3
```

When returning multiple outputs, there are two possible syntaxes for a function call. In this first case (cell 9), we create two **output variables separated by a comma** (line 2). The first variable—`database`—contains the returned variable `db` from cell 7, line 36. When we print it at line 5, we see the dictionary containing usernames and passwords for each customer. The second variable—`number_customers`—contains the returned variable `n_customers` (cell 7, line 36) and when we print it at line 6, we see the dictionary length, which is 3. Let's look at the other possible syntax for the outputs.

```
[10]:
1  # create the database - single return
2  outputs = create_database(customers)
3  print("Output tuple:", outputs)
4
5  # get and print the database
6  database = outputs [0]
7  print("Database:", database)
8
9  # get and print the number of customers
10 number_customers = outputs[1]
11 print("Number of customers:", number_customers)
```

```
create the database - single return
outputs is assigned create database customers
print Output tuple: outputs

get and print the database
database is assigned outputs in position zero
print Database: database

get and print the number of customers
number customers is assigned outputs in position one
print Number of customers: number customers
```

```
Output tuple: ({'jsmith': '7863', 'mseid': '1953', 'mlopez': '6350'},3)
Database: {'jsmith': '7863', 'mseid': '1953', 'mlopez': '6350'}
Number of customers: 3
```

In this second syntax, we assign both returned variables to **one single variable** called `outputs` (line 2). As we can see from the print (line 3), `outputs` is a **tuple** that contains the database and the number of customers. As you might recall from Chapter 22, a tuple is a sequence of elements separated by commas and contained within round brackets. Tuple elements are **immutable**, which means that we cannot overwrite, add, or delete any element. However, we can extract the elements by using the same **slicing** principles that we learned for lists and strings. Thus, to get the dictionary, we slice `outputs` in position `0` (line 6), and we print it as a check (line 7). Similarly, to get the number of customers, we slice `outputs` in position `1` (line 10) and print it as a check (line 11). In alternative, we could have directly printed the sliced variable in both cases—that is, `print("Database:", outputs [0])` and `print("Number of customers:", outputs [1])`.

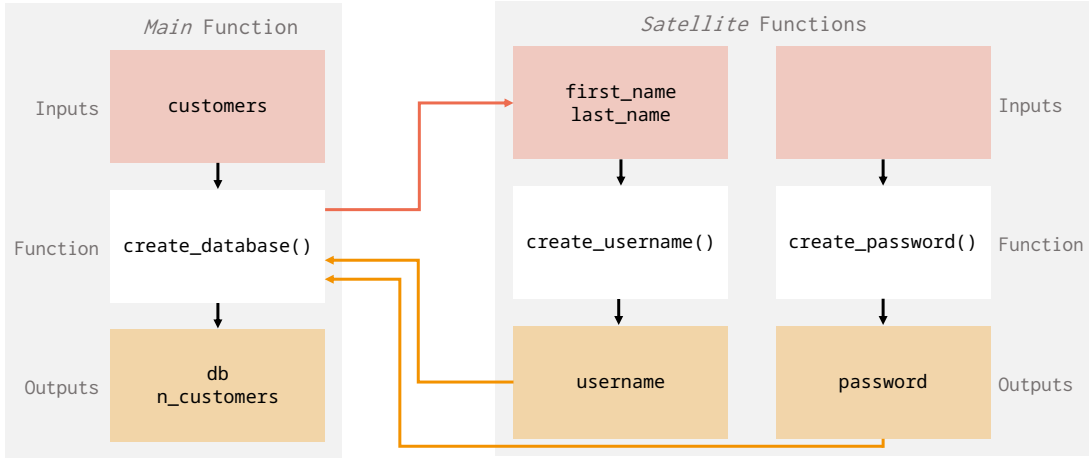


Figure 29.2. Modular organization of code: Main and satellite functions.

Before concluding this chapter, let's briefly analyze how we **modularized** our code with the help of Figure 29.2. To divide and conquer our task, we created three functions, each of them with a different role. The function `create_database()`—on the left side of Figure 29.2—is the **main function**, because it (1)

receives the input for the task to solve—a list of customer first names and last names; (2) **performs the flow of operations needed to solve the task**—that is, it creates the database of usernames and passwords; and (3) **provides the final output**—that is, it returns the dictionary and the number of customers. In other Python examples you might come across, you may find that the main function is actually called `main()`. The other two functions—`create_username()` and `create_password()`, on the right side of Figure 29.2—are called **satellite functions** because each of them **performs one specific task**. How do main function and satellite functions **interact**? Through the flow of **inputs and outputs**! The main function sends inputs to the satellite functions—in our case, `first_name` and `last_name` are sent to `create_username()` (orange line in Figure 29.2)—and receives outputs—`username` from `create_username()` and `password` from `create_password()` (yellow lines). The received outputs can then be used to **create new variables**—such as the dictionary `db` in our example—or **as inputs for subsequent satellite functions**, as you will see in the coding exercise at the end of the chapter.

We'll conclude this chapter with an important note about the **joy and difficulties of coding**. Most likely, when you read the task at the beginning of this chapter and started drafting your own solution, what came to your mind was somewhat different from the solution you found. Maybe your idea was more complicated, maybe less structured, or maybe you got stuck and frustrated. Want to know a secret? The solution that you learned in this chapter did not come out of my mind the way it looks now. The initial idea was messy, redundant, and at times I was uncertain about what to do. In some cases, I started with the lines of code that do the actual job and then wrote a function around them. Other times, when things were very clear to me, I just wrote a function from top to bottom. It required hours of tweaking, corrections, and adjustments to get the code to be structured, clean, and simple-looking. So, don't worry if it takes you quite some time before coming to a final, clean solution when solving a task. That's normal! Do you remember the *In more depth* section in Chapter 17 entitled *Writing code is like writing an email*? We write a draft solution, then we modify it, then we modify it again, and again, and finally we arrive at a satisfactory result. The most important thing in coding is **persistence**!

Recap

- The keyword `return` has two roles:
  - It transfers output variables from the function body to the function call. When multiple variables are returned, they are separated by commas both in the function body and in the function call. In the latter, they can also be collected into a tuple.
  - It marks the end of a function. Commands written after `return` do not get executed.
- Tuples are a data type where elements are immutable, meaning they cannot be changed. Tuple slicing follows the same rules as list (or string) slicing.
- In docstrings, the syntax of returned variables is the same as the syntax of input parameters.
- It is important to test function correctness by calling them with appropriate arguments.
- A project is often composed of a main function and some satellite functions. The main function executes the solution to the whole task, whereas each satellite function executes one specific subtask.

What is None?

Have you ever got `None` as an output when calling a function? The **keyword `None`** indicates that the **function has no return**. In other words, at the end of the function there is no keyword `return` followed by one or more variables. Let's look at the following example, modified from cell 4:

```
[4]:
```

```
1  import random
2
3  def create_password():
4      """Create a password composed of
       four random integers
5
6      Returns
7      -------
8      password : string
9          Created password
10     """
11
12     # create a random number with four digits
13     password = str(random.randint(1000, 9999))
14
15     # print password
16     print(password)
```

```
import random

def create password
Create a password composed of
four random integers

Returns

password : string
Created password


create a random number with
four digits
password is assigned str
random dot randint 1000 9999

print password
print password
```

At line 16, we substitute `return password` with `print(password)`. Let's see what changes when calling the function:

```
[5]:
```

```
1  password_1 = create_password()

2  print(password_1)
```

```
4883
None
```

```
password one is assigned
create password
print password one
```

The first print—`4883`—comes from the command `print(password)` in cell 4, line 16. The second print—`None`—is from the command `print(password_1)` in cell 5, line 2. Because we did not return `password` at the end of the function body (cell 4, line 16), `password_1` will contain the keyword `None`, which is what we see when printing at line 2. Thus, in general, when a function does not return a value, either **call the function on its own**—without assigning its output to a variable—or **modify the function to return what you need**.

Let's code!

1. *What does Bill Gates tweet about?* Bill Gates is highly active on Twitter (now known as "X"), with 65.7 million followers as of November 2024. But what does he tweet about? To answer this question, in this exercise, you'll learn how to extract the most common words from some text using basic techniques from natural language processing (NLP), an interesting and challenging computational field focused on analyzing human language. In steps a, b, and c, you will implement some techniques to preprocess text—that is, to clean text for the main

analysis—and in step d, you will create the main function to identify the most common words Bill Gates uses.

a. *Removing punctuation.* When preprocessing text, a common initial step is to remove punctuation. Write a function that, given a string, returns the same string without any punctuation. *Hint*: You can use this punctuation string: `!"#$%&'+,-./:;<=>?@[]^_`{|}~`
(Example input: `Hello! How are you?`. Expected output: `Hello How are you`).

b. *Converting to lowercase and segmenting into words.* The next steps are to standardize all words to lowercase and to segment the text, that is, to split the text into individual words. Write a function that, given a string: (1) converts the string to lowercase, (2) segments the text by splitting it into a list of words, and (3) returns the list of words. *Hint*: You only need two string methods!
(Example input: `Hello How are you`. Expected output: `['hello', 'how', 'are', 'you']`).

c. *Removing stop words.* Another important step in NLP is removing stop words, which are common words that typically don't add meaningful information to the text (e.g., prepositions). Write a function that, given a list of strings, returns the list without stop words. *Hint*: Use the following list of stop words:
`["all", "how", "to", "what", "are", "the", "a", "of", "in", "it", "from", "and", "for", "about", "my", "on", "can"]`.
(Example input: `['hello', 'how', 'are', 'you']`. Expected output: `['hello', 'you']`).

d. *Counting the most common words.* It's finally time to answer our question: What does Bill Gates tweet about? Create a function that calls the previous functions and returns the most common words in the tweets. More specifically, write a function that, given a list of strings, (1) executes text preprocessing—that is, removes punctuation, converts to lowercase, segments text into words, and removes stop words; (2) creates a dictionary in which the keys are each unique word and the values are their corresponding counts; and (3) returns the 10 most frequent words from the dictionary. *Hint*: To sort dictionary keys based on their values you can use the following command:
`sorted_keys = sorted(my_dictionary, key=my_dictionary.get, reverse=True)`
Use the following list of strings as an input:

```
tweets = ["Meaningful action from business leaders will require the courage to take risks that many
companies aren't used to taking.",
"Thanks to @andersoncooper, @SeaArtsLectures, and everyone who joined our virtual conversation about
climate change. Great to have so much support from my hometown for this important work.",
"Great to see this important step as the United States resumes our global leadership on climate change.
Looking forward to working with @POTUS and Congress on a plan to ensure we reach net zero by 2050.",
"Thanks to @streickercenter for hosting the launch of my virtual book tour. It was great to hear so
many thoughtful questions about what we can all do to help avoid a climate disaster.",
"When I talk to people about climate change, I almost always get asked the same question: What can I do
to help? Here are some actions individuals can take to move us closer to a zero-carbon future",
"Thanks for inviting me on the podcast, @karaswisher.",
"Thank you, AlokSharma_RDG. We have a lot of work ahead of us to reach net-zero emissions by 2050 and
avoid a climate disaster. Your leadership of #COP26 is critical. Thanks also to @howtoacademy,
@penguinlive and @Waterstones for hosting the London stop of my virtual book tour!",
"It was great to talk with @alroker about my new book and the solutions we need to fight climate
change."
"Thanks for another great conversation @Trevornoah!",
"I wrote How to Avoid a Climate Disaster because I see not just the problem of climate change; I also
see an opportunity to solve it. Here's how: http://gatesnot.es/3dh2kQy",
"I had a great time working with @FortuneMagazine on this special digital issue about climate change.
The entire business community has a role to play—and we need to start now.",
"How to Avoid a Climate Disaster is available now. I hope you'll check out the book, but more
importantly, I hope you'll do what you can to help keep the planet livable for generations to come:
http://b-gat.es/climatebook"]
```

30. Free ticket at the museum

Input validation and output variations

What happens if we provide wrong inputs to a function? Sometimes the function breaks—meaning we get an error—and some other times we get a meaningless result. In both cases, it might be difficult to understand what went wrong. In this chapter, we will learn how to make sure that function inputs are of the right type and value. In addition, we will also learn how to return outputs in specific cases, that is, based on conditions or directly as values. Let's tackle all this through the code below. Follow along with Notebook 30!

- You work at a museum and have to update the online system to buy tickets. The update is that people who are 65 and older now qualify for a free ticket. Write a function that asks visitors to enter their prefix, last name, and age; checks the types and values of these inputs; and returns a message telling the visitor if they are eligible for a free ticket.

[1]:

```
def free_museum_ticket(prefix, last_name, age):

    """Returns a message containing inputs
    and free ticket eligibility based on age

    Ex.: Mrs. Holmes, you are eligible for a
    free museum ticket because you are 66


    Parameters
    ----------
    prefix : string
        Ms, Mrs, Mr
    last_name : string
        Last name of a visitor
    age : integer
        Age of a visitor

    Returns
    -------
    string
        Message containing inputs and
        eligibility
    """

    # --- checking parameter types ---

    # the type of prefix must be string
    if not isinstance(prefix, str):
        raise TypeError("prefix must be a
        string")
```

```
def free museum ticket prefix last
name age
Returns a message containing inputs
and free ticket eligibility based
on age
Ex.: Mrs. Holmes, you are eligible
for a free museum ticket because
you are 66

Parameters

prefix : string
Ms, Mrs, Mr
last name : string
Last name of a visitor
age : integer
Age of a visitor

Returns

string
Message containing inputs and
eligibility


checking parameter types

the type of prefix must be string
if not isinstance prefix str
raise TypeError prefix must be a
string
```

```
    # the type of last_name must be string
    if not isinstance(last_name, str):
        raise TypeError("last_name must be
        a string")

    # the type of age must be integer
    if not isinstance(age, int):
        raise TypeError("age must be an integer")



    # --- checking parameter values ---

    # prefix must be Ms, Mrs, or Mr
    if prefix not in ["Ms", "Mrs", "Mr"]:
        raise ValueError("prefix must be Ms, Mrs,
        or Mr")

    # last_name must contain only characters
    if not last_name.isalpha():
        raise ValueError("last_name must contain
        only characters")

    # age has to be between 0 and 125
    if age < 0 or age > 125:

        raise ValueError("age must be between
        0 and 125")



    # --- returning output ---

    if age >= 65:
        return prefix + ". "  + last_name +
        ", you are eligible for a free
        museum ticket because you are " + str(age)


    else:
        return prefix + ". "  + last_name +
        ", you are not eligible for a free museum
        ticket because you are " + str(age)
```

```
the type of last name must be
string
if not isinstance last name str
raise TypeError last name must be a
string

the type of age must be integer
if not isinstance age int
raise TypeError age must be an
integer



checking parameter values

prefix must be Ms, Mrs, or Mr
if prefix not in "Ms" "Mrs" "Mr"
raise ValueError prefix must be Ms,
Mrs, or Mr

last name must contain only
characters
if not last name dot isalpha
raise ValueError last name must
contain only characters

age has to be between zero and 125
if age less than zero or age
greater than 125
raise ValueError age must be
between zero and 125



returning output

if age greater than or equal to 65
return prefix concatenated with dot
space concatenated with last name
concatenated with you are eligible
for a free museum ticket because
you are concatenated with str age
else
return prefix concatenated with
dot space concatenated with last
name concatenated with you are not
eligible for a free museum ticket
because you are concatenated with
str age
```

✎ True or false?

1.	In the docstrings, after the description, we can add an example for further clarification.	T	F
2.	The built-in function isinstance() checks a variable type and returns an integer.	T	F
3.	raise TypeError() and raise ValueError() stop the function and provide an error message.	T	F
4.	raise is a function.	T	F
5.	int and str are the same as int() and str().	T	F

Computational thinking and syntax

Let's begin to analyze the function by taking a look at what it does. In the docstring description, we see that the aim of `free_museum_ticket()` is to return a message composed of a concatenation of the inputs and the eligibility for a free ticket based on age (line 2). The description is followed by a message **example**, for further clarification (line 3). Adding an example is good practice to make the function **outcome more quickly and easily understood**. Let's continue by looking at the inputs. The function has 3 parameters: `prefix`, `last_name`, and `age` (line 1), whose types and values are described in the documentation (lines 5–12) and further checked in the first 6 blocks of code (lines 20–47). The blocks have a similar structure, composed of an if condition followed by a `raise` statement. Let's have a closer look. The first three blocks check the **parameter types** (lines 20–32). In the first block (lines 22–24), we check if the first parameter `prefix` is a string. To do so, we write an if condition (line 23) composed of (1) the keyword `if`, (2) the logical operator `not`, and (3) the built-in function **`isinstance()`**, which **checks if a variable is of a specific type**. It takes 2 parameters: the **variable to check**—`prefix`—and the **wanted type**—that is, **`str`**. Other possible types are **`int`**, **`list`**, **`dict`**, etc. Types are not followed by round brackets and should not be confused with the built-in functions `str()`, `int()`, etc. The function `isinstance()` **returns a Boolean**, that is, `True` if the variable is of the desired type—e.g., if `prefix` is a `str`—and `False` otherwise. Why do we use the logical operator **`not`** in the if condition? To make the condition true when we want it to be executed. In Boolean terms, we can say that if `prefix` is not a string, then the command `if not isinstance()` becomes `if not False`, which is the same as `if True` (see Chapter 19), and thus the following statement gets executed. The statement is composed of (1) the keyword **`raise`**, which **stops the function**, and (2) the built-in **exception `TypeError()`**, which **specifies the nature of the error**—type—and **provides a message** indicating **what must be done to avoid the error** (line 24). To see the effect of these lines of code, let's call the function using the wrong type for `prefix` and analyze what happens.

[2]:
```
1 # checking prefix type
2 message = free_museum_ticket(1, "Holmes", 66)
3 print(message)
```

```
checking prefix type
message is assigned free
museum ticket one Holmes 66
print message
```

```
---------------------------------------
TypeError       Traceback (most recent call last)
Cell In[2], line 2
       1 # checking prefix type
----> 2 message = free_museum_ticket(1, "Holmes", 66)
       3 print(message)

Cell In[1], line 24, in free_museum_ticket(prefix, last_name, age)
      20 # --- checking parameter types ---
      21
      22 # the type of prefix must be string
      23 if not isinstance(prefix, str):
----> 24     raise TypeError("prefix must be a string")
      25
      26 # the type of last_name must be string
      27 if not isinstance(last_name, str):

TypeError: prefix must be a string
```

We use 1 for `prefix`—that is, an integer instead of a string—and correct types for `last_name`—a string—and `age`—an integer (line 2). We assign the function output to the variable `message`, and we print it (line 3). We get an error message. Let's dig deeper! As usual, we start from the last line. Here, we read the type of exception—`TypeError()`—and the string we wrote as an argument—`prefix must be a string` (cell 1, line 24). Do you remember seeing type errors before? In the *In more depth* sections of Chapter 9, entitled *Dealing with TypeError*, and Chapter 14, entitled *Don't name variables with reserved words!*, we learned how to read type error messages and how to fix the code to avoid them. Now, we know what's behind the scenes, that is, how to create a TypeError message! You've learned quite a lot since then, haven't you? Let's complete the analysis of the error message by looking at the arrows pointing at specific lines of code. The **top arrow** points at line 2 of cell 2, telling us **where the error happens in the current cell**—that is, where we called the function. The **second arrow** points at line 24 of cell 1, which is **where the error originated**, that is, where we raised the `TypeError()`. Note that since the error happens at cell 2 and thus the code stops, we do not see any print because the command at line 3 is not executed.

Let's continue by checking the type of the second parameter `last_name` (lines 26–28). As for `prefix`, `last_name` must be a string. Thus, we simply reuse the commands at lines 23–24, substituting `prefix` with `last_name` in the if statement (line 27) and in the `TypeError()` message (line 28). Let's test whether the type error works, by calling the function with the wrong type for `last_name`—starting from this cell, only the relevant part of the error message is reported from brevity.

[3]:
```
1  # checking last_name type
2  message = free_museum_ticket("Mrs", 1.2, 66)

3  print(message)
```
```
checking last name type
message is assigned free
museum ticket Mrs 1.2 66
print message
```
```
---------------------------------------
Cell In[3], line 2
----> 2 message = free_museum_ticket("Mrs", 1.2, 66)
Cell In[1], line 28, in free_museum_ticket(prefix, last_name, age)
       27 if not isinstance(last_name, str):
----> 28     raise TypeError("last_name must be a string")
TypeError: last_name must be a string
```

As expected, in the last line of the message, we get: `TypeError: last_name must be a string`, which is the error that occurred at line 2 of the current cell, and originated at line 28 of cell 1.

Let's conclude the check of the parameter types with `age` (lines 30–32). In this case, the parameter must be an integer, not a string. Thus, in the built-in function `isinstance()`, the two inputs are the variable `age` and the type `int` (line 31). In `TypeError()`, the message becomes `age must be an integer` (line 32). Let's test the correctness of this code with the following function call.

[4]:
```
1  # checking age type
2  message = free_museum_ticket("Mrs", "Holmes", "Hi")

3  print(message)
```
```
checking age type
message is assigned free
museum ticket Mrs Holmes Hi
print message
```
```
---------------------------------------
Cell In[4], line 2
----> 2 message = free_museum_ticket("Mrs", "Holmes", "Hi")
Cell In[1], line 32, in free_museum_ticket(prefix, last_name, age)
       31 if not isinstance(age, int):
----> 32     raise TypeError("age must be an integer")
TypeError: age must be an integer
```

We enter the string `"Hi"` as the third parameter. As expected, the error occurs at line 2 of cell 4 and originated at line 32 of cell 1.

The following three blocks of code of `free_museum_ticket()` check the **parameter values** (lines 35–47). Similarly to before, each block contains an if construct composed of an if condition and a statement raising an exception. In the condition, we assess the parameter values by establishing some **criteria specific to the context** of the task. For example, for `prefix`, we establish that the possible values are `"Ms"`, `"Mrs"`, or `"Mr"`. Thus, we enclose the three strings into a list, and we check **if the value** of `prefix` **is in that list** (line 38). If not, we raise a `ValueError()` in the following statement (line 39). **`ValueError()`** is the **exception** specific **for value errors**, and it works the same way as `TypeError()`. Within the round brackets, we write a message indicating what must be done to avoid the error—in our case, `prefix must be Ms, Mrs, or Mr`. Let's check what happens when raising the value error for `prefix` in the following function call.

```
[5]:
1  # checking prefix value
2  message = free_museum_ticket("Dr", "Holmes", 66)

3  print(message)
```

```
checking prefix value
message is assigned free
museum ticket Dr Holmes 66
print message
```

```
---------------------------------------
Cell In[5], line 2
----> 2 message = free_museum_ticket("Dr", "Holmes", 66)
Cell In[1], line 39, in free_museum_ticket(prefix, last_name, age)
       38 if prefix not in ["Ms", "Mrs", "Mr"]:
----> 39     raise ValueError("prefix must be Ms, Mrs, or Mr")
ValueError: prefix must be Ms, Mrs, or Mr
```

For `prefix`, we use `"Dr"`, which is not in the list of possible values, `["Ms", "Mrs", "Mr"]`. Thus, we get a value error, as the message in the last line specifies. The error happens at line 2 of cell 5 and originated at line 39 of cell 1, as we can see from the two arrows in the pink area.

Let's continue with checking the possible values for `last_name`. What condition should we use? Should we list all the possible last names in the world? What if some are not registered or new? In cases like this, we can look into the **types of characters** composing the string. For last names, we can require that all the characters are letters of the alphabet and, to do so, we can use the string method **`.isalpha()`**(line 42)—for simplicity, we'll consider only last names composed of characters and not containing punctuation, such as O'Connor, or a space, such as García Lopez. In other contexts, we can perform the check using methods such as **`.isdigit()`**, **`.isalnum()`**, **`.islower()`**, **`.isupper()`**, **`.istitle()`**—see the appendix of Chapter 27—depending on the characteristics that the string must have. If the condition is not met, then we raise a value error saying that the last name must contain only characters (line 43). Let's test the execution of these two lines of code by calling the function with an incorrect value for `last_name`.

```
[6]:
1  # checking last_name value
2  message = free_museum_ticket("Mrs", "82", 66)

3  print(message)
```

```
checking last name type
message is assigned free
museum ticket Mrs 82 66
print message
```

```
---------------------------------------
Cell In[6], line 2
----> 2 message = free_museum_ticket("Mrs", "82", 66)
Cell In[1], line 43, in free_museum_ticket(prefix, last_name, age)
       42 if not last_name.isalpha()
----> 43     raise ValueError("last_name must contain only characters")
ValueError: last_name must contain only letters
```

In the function call (line 2), we use the string `"82"` for `last_name`. We get the value error with the message that we entered at line 43 in cell 1.

Let's finally check the value of the last parameter `age`. What constraint should we use this time? One reasonable option is to raise a value error if `age` is not within the **range** of a human lifetime. How do we define the range? The minimum is obviously 0 years old. What about the maximum? According to Wikipedia[1], the oldest person ever was Jeanne Calment who died when she was 122 years and 164 days old! So, we can keep a bit of margin and define 125 as the maximum. Therefore, we check if `age` is less than 0 or greater than 125 (line 46). If so, we raise the `ValueError()` with the message suggesting the proper age range to use (line 47). Let's test these commands by calling the function with an age

[1] `https://en.wikipedia.org/wiki/List_of_the_verified_oldest_people`

out of range!

[7]:
```
1 # checking age value
2 message = free_museum_ticket("Mrs", "Holmes", 130)
3 print(message)
```

```
checking age type
message is assigned free
museum ticket Mrs Holmes 130
print message
```

```
---------------------------------------
Cell In[7], line 2
----> 2 message = free_museum_ticket("Mrs", "Holmes", 130)
Cell In[1], line 47, in free_museum_ticket(prefix, last_name, age)
       46 if age < 0 or age > 125:
----> 47     raise ValueError("age must be between 0 and 125")
ValueError: age must be between 0 and 125
```

In the function call (line 2), we provide the integer `130` for the parameter `age`, and we get the value error that we created at line 47 of the function, as expected.

At this point, you might ask yourself: do I have to implement the input check in every function I write? Nope! In Python, we assume that the **docstrings clearly indicate expected type and value** of the parameters and that a coder passes valid arguments to a function. So, why did we learn it? Because a parameter check is useful in **main functions** or when there are **user-provided inputs**—for example, when using the built-in function **`input()`**, as you will see in the coding exercise at the end of this chapter.

Let's conclude this chapter by analyzing the returns. In `free_museum_ticket()`, we **return different outputs based on conditions** (lines 50–55). To do that, we use an if/else construct where each statement contains the keyword **`return`**. If the age of the visitor is greater than or equal to 65 (line 52), then we return the string indicating the eligibility to a free ticket (line 53), otherwise (line 54) we return a string indicating the ineligibility to a free ticket (line 55). As you might remember from the previous chapter, `return` not only "pushes" the variable out of a function, but it also **stops the function**. Thus, any command following the executed return statement (line 53 or 55) will never be executed. In this function, we also **directly return a value** without creating an intermediate variable—this can be done in any function. In other words, we could have created a variable called `message` to which we assign the concatenation: `prefix + ". " + last_name + ", you are eligible for a free museum ticket because you are " + str(age)`, and then returned it as `return message`. However, we directly return the concatenation. Note that **in the docstrings we only indicate the type**—`string`—of the return as there is no variable name (line 16).

Let's conclude by calling the functions with the correct input types and values to test the correctness of the two returns.

[8]:
```
1 # person is eligible
2 message = free_museum_ticket("Mrs", "Holmes", 66)
3 print(message)
```

```
person is eligible
message is assigned free
museum ticket Mrs Holmes 66
print message
```

```
Mrs. Holmes, you are eligible for a free museum ticket because you are 66
```

[9]:
```
1 # person is not eligible
2 message = free_museum_ticket("Mrs", "Choi", 38)
3 print(message)
```

```
person is not eligible
message is assigned free
museum ticket Mrs Choi 38
print message
```

```
Mrs. Choi, you are not eligible for a free museum ticket because you are 38
```

In both cells, the inputs pass the type and value checks, thus the function executes the code and returns a message according to age. In the first case (cell 8), the age is 66 (line 2)—which is greater than 65—so the visitor is eligible for a free ticket. In the second case (cell 9), the age is 38 (line 2)—which is less than 65—so the visitor is not eligible for a free ticket.

✎ Match the sentence halves

1. `raise` is a	a. built-in function.
2. `TypeError()` and `ValueError()` are	b. keyword.
3. `int` is a	c. built-in function.
4. `int()` is a	d. exceptions.
5. `isinstance()` is a	e. type.

Recap

- We implement parameter checks in main functions or in presence of external inputs. The check is executed using an if/else construct. In the if condition:
  - When checking a type, we use the logical operator `not` followed by the built-in function `isinstance()`, whose parameters are the variable to check and the wanted type. Possible types are `str`, `int`, `list`, `dict`, etc.
  - When checking a value, we have to define constraints. We can use membership to a list, variable methods—such as `.isalpha()` for strings—or intervals for numbers.

  In the statement, we use `raise` followed by the exception `TypeError()` or `ValueError()` with a message indicating how to avoid the error.
- When we want to return different outputs based on conditions, we can use an if/else construct where the statements contain the keyword `return` followed by the wanted output.
- It is possible to return values instead of variables. In this case, in the docstrings we indicate only the type.
- In docstrings, it is possible to write an example after the function definition to enhance clarity.

How can I avoid interrupting the flow?

As you learned in this chapter, `raise TypeError()` and `raise ValueError()` stop the function and provide an error message. But what if we do not want to interrupt the flow of our code? Imagine that `free_museum_ticket()` is a satellite function called by the main function within a larger project. Every time the type or value of an input is not correct, `free_museum_ticket()` stops, displays the pink error message, and the flow of the whole project is interrupted, creating inconvenience. What can we do to avoid that? One possibility is to make the satellite function provide a Boolean as a return. For example, line 24 of the first block of code of `free_museum_ticket()` could be modified as following:

```
[1]:
22      # the type of prefix must be string
23      if not isinstance(prefix, str):
24          return prefix, False
```

```
the type of prefix must be string
if not isinstance prefix str
return prefix False
```

↳

The main function—which calls `free_museum_ticket()`—would receive both `prefix` and `False`, indicating that there is something wrong about `prefix`. Then, the following code in the main function could handle the situation by asking the user to reenter a correct prefix. The coding exercise at the end of this chapter—especially step c.—contains a more complete example of this concept. So, let's start coding!

Let's code!

1. *Let's play hangman!* Everybody knows hangman! It is a game where the aim is to guess a hidden word by suggesting letters. You have all the knowledge to implement hangman by yourself! Think about the task you have to implement (divide!) and go for it (conquer!). You can then compare your implementation with the one suggested in the following steps. The first three steps will invite you to implement satellite functions, each of them representing a sub-task. The last step will suggest you how to write the main function.
    a. *Pick a random word.* Create a satellite function that given a list of words, returns a randomly selected word in lowercase.
    (Example input: `["spoon", "Fork", "KNIFE"]`. Example output: `"fork"`).
    b. *Show the guessed letters.* Create a satellite function that given a word and a guessed letter, shows the word with the guessed letter revealed in its correct positions, whereas the remaining, unguessed letters are shown as underscores.
    (Example input: `("l", "hello")`. Example output: `_ _ l l _`).
    c. *Check the user input.* Create a satellite function that takes a string as an argument and returns the same string in lowercase and a Boolean, which is `True` if the string is a letter, and `False` otherwise. In addition, the function prints specific messages depending on the input string. If the string:
        - Is composed of multiple letters, print: `Please, enter a single letter`;
        - Is composed of one or more numbers, print: `Please, enter a letter not a number`;
        - Is composed of a combination of letters and numbers, print: `Please, enter a letter, not a combination of letters and numbers`;
        - Contains a symbol, print: `Please, enter a letter, not a symbol`.

    (Example input: `e1`. Example output: `e1, False`). *Hint*: Which strings methods will you use? See the table in the appendix of Chapter 27.
    d. *Assemble the hangman game.* Create a main function that given a list of words:
        - Randomly chooses a word from the list;
        - Displays the word with missing letters;
        - Asks the player for a character. If the character is a valid letter, then the game continues; otherwise, the player is prompted until they enter a valid letter;
        - Checks for repeated guesses. If the player had already guessed that letter, print the message: `You already guessed this letter. Choose again!`;
        - Checks if the letter is in the word, and updates the word accordingly. Make sure to update also when letters are double (e.g., double `"l"` in `"hello"`);
        - Keeps asking for a new letter until the word is complete;
        - At the end, congratulates the player and asks them if they want to play again. If so, the game restarts with a new word, otherwise the game stops.

    (Example input: `["garden", "cave", "quilt", "bubble", "secretary", "light"]`)

31. Factorials

Recursive functions

In this chapter, you will learn a particular type of function called recursive functions. They can be challenging to understand and implement, but they are very useful in certain situations, as you will see. To better understand how recursive functions work, let's compute factorials. Have you ever heard of them? A factorial is the product of all positive integers that are less than or equal to a given positive integer. For example, the factorial of 4 is 24, calculated as 1x2x3x4—or 4x3x2x1. How would you write a function that calculates the factorial of an integer? Write your own function before looking at the proposed solution below. You will find the code for this chapter in Notebook 31.

- Write a function that calculates the factorial of a given integer using a `for` loop:

[1]:

```
def factorial_for(n):
    """Calculates the factorial of a given
    integer using a for loop

    Parameters
    ----------
    n : integer
        The input integer

    Returns
    -------
    factorial : integer
        The factorial of the input integer
    """

    # initialize the result to one
    factorial = 1

        # for each integer between 2 and
        the input integer
        for i in range(2, n+1):
            # multiply the current result
            by the current integer
            factorial *= i

        # return the result
        return factorial

# call the function
fact = factorial_for(4)
print(fact)
```

```
24
```

```
def factorial for n
Calculates the factorial of a given
integer using a for loop

Parameters

n : integer
The input integer

Returns

factorial : integer
The factorial of the input integer


initialize the result to one
factorial is assigned one

for each integer between two and the
input integer
for i in range two n plus one
multiply the current result by the
current integer
factorial multiply by and reassign i

return the result
return factorial

call the function
fact is assigned factorial for four
print fact
```

Does your implementation look similar to the one above?

- Compare the previous function with the following recursive function:

[2]:
```
1  def factorial_rec(n):                          def factorial rec n
2      """Calculates the factorial of a given     Calculates the factorial of a given
       integer using recursion                    integer using recursion
3
4      Parameters                                 Parameters
5      ----------
6      n : integer                                n : integer
7          The input integer                      The input integer
8
9      Returns                                    Returns
10     -------
11     integer                                    integer
12         The factorial of the input integer     The factorial of the input integer
13     """
14
15     # if integer is greater than 1             if integer is greater than one
16     if n > 1:                                  if n greater than one
17         # execute the recursion                execute the recursion
18         return factorial_rec(n-1) * n          return factorial rec n minus one times n
19     # otherwise                                otherwise
20     else:                                      else
21         # return 1                             return one
22         return 1                               return one
23
24 # call the function                            call the function
25 fact = factorial_rec(4)                        fact is assigned factorial rec four
26 print(fact)                                    print fact
```
```
24
```

What similarities and differences do you notice between the two functions? Get some hints while solving the following exercise!

✎ True or false?

1. Both functions have one parameter and one return. T F
2. Both function contain a for loop. T F
3. In the recursive function, there is only one return statement in the if/else construct. T F
4. We can call a function in the same cell where we write the function. T F

Computational thinking and syntax

Let's start by analyzing the function `factorial_for()` in cell 1. In the docstrings, we see that the input is an integer called `n`—which stands for number—(lines 4–7) and the output is another integer called `factorial` (lines 9–12). In the function body, we first initialize the output `factorial` to `1` (line 16). Then, we create a `for` loop where the index `i` will be assigned all the consecutive numbers from 2 to the input number `n` plus 1 (line 19)—do you remember the plus one rule for the stop in `range()` from

Chapter 8? Within the loop, we calculate the product between the current value of `factorial` and the current value of `i`, and we reassign the result to `factorial` (line 21). Let's quickly go through the three iterations for more clarity:

- In the first iteration, `factorial` is `1` and `i` is `2`, so the result of `factorial*i`—that is, `1*2`—is `2`, which is reassigned to `factorial`.
- In the second iteration, `factorial` is `2` and `i` is `3`, so the result of `factorial*i`—that is, `2*3`—is `6`, which is reassigned to `factorial`.
- In the third iteration, `factorial` is `6` and `i` is `4`, so the result of `factorial*i`—that is, `6*4`—is `24`, which is reassigned to `factorial`—and is the final value.

We conclude the function by returning `factorial` (line 24). To test the function, we call it with the number `4` as an input, and we assign the returned value to the variable `fact` (line 27), which we print in the following command (line 28). Note that a function **code** and **call** can be **in the same cell** for convenience. In general, we call functions like `factorial_for()` **iterative functions** because they **contain a loop to repeat some parts of their code**.

Let's now move to the recursive function `factorial_rec()` (cell 2) and identify similarities and differences with `factorial_for()` (cell 1). From the docstrings, we see that the function takes an integer `n` as an input (lines 4–7)—similarly to `factorial_for()`—and returns a **value** as an output (lines 9–12)—differently from `factorial_for()`, which returns the **variable** `factorial`. The main difference is in the function body, where `factorial_for()` contains a `for` loop, whereas `factorial_rec()` contains an `if/else` construct (lines 15–22). In this construct, if `n` is greater than 1 (line 16), we return the output of `factorial_rec()` calculated for the consecutive smaller integer—that is, `n-1`—multiplied by `n` (line 18), otherwise (line 20), we return `1` (line 22). Noticed anything unusual? In the first statement (line 18), we call `factorial_rec()` itself! This is because a **recursive function is a function that calls itself several times, until it reaches a base case**. In other words, recursive functions create a **cascade of function openings and executions** until a certain point where the path is **reversed to return the outputs and close the functions**. Let's understand this mechanism better with the help of Figure 31.1.

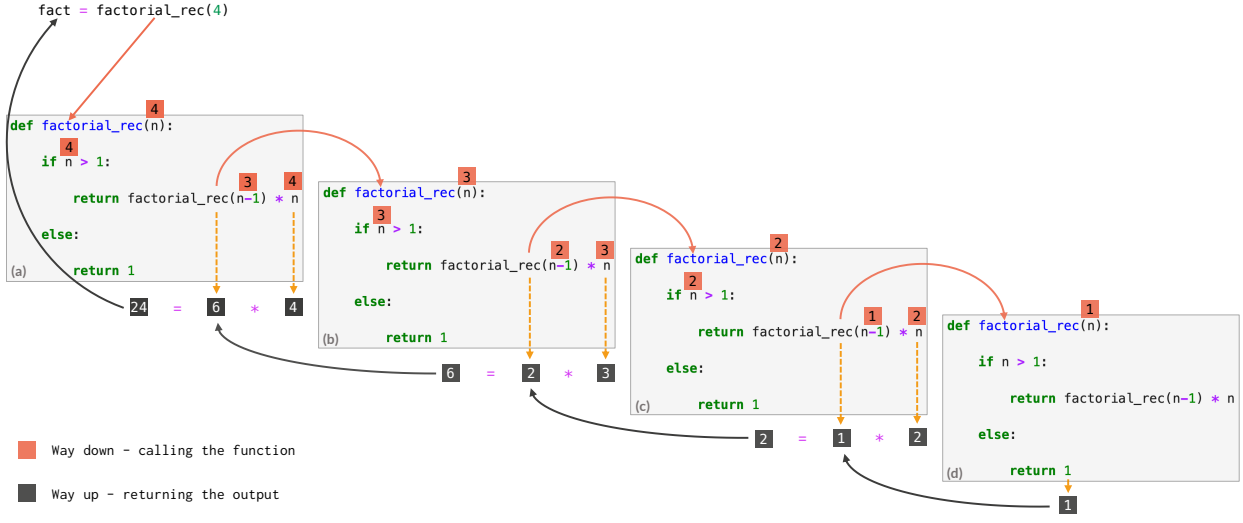


Figure 31.1. The mechanism of a recursive function.

There are four major components. First, there is the initial function call `fact = factorial_rec(4)` (corresponding to line 25 in cell 2) at the top left of Figure 31.1. Second, there are four simplified represen-

tations of `factorial_rec()` in cascade, each of them contained in a gray rectangle and indicated as (a), (b), (c), and (d), respectively. Third, there are orange arrows and numerical squares representing the "way down", that is, the consecutive openings of several `fact = factorial_rec()` with their current inputs. And fourth, there are black arrows and numerical squares constituting the "way up", that is, the return of the outputs and the closure of the functions. Now, let's see how these components interact with each other. When we **call** `fact = factorial_rec(4)`, we **begin the "way down"**—as indicated by the first, straight orange arrow—and open a cascade of functions as follows:

- In (a), `n` is `4`—that is, the initial input—thus the header of the function is `def factorial_rec(4)`. The `if` condition is `if 4>1`, which is `True`, so we move to the following statement containing the call `factorial_rec(3)`—`3` is calculated from `n-1`, that is, `4-1`. From here, we follow the orange arrow and move to (b), leaving the function open in (a) and temporarily disregarding all its remaining code.
- In (b), `n` is `3`, so the header is `def factorial_rec(3)`. The `if` condition is now `if 3>1`, which is `True` again, so we move to the following statement where we call `factorial_rec(2)`. Thus, we follow the orange arrow and move to (c), leaving the function open in (b) and temporarily disregarding all its remaining code.
- In (c), `n` is `2`, thus the header is `def factorial_rec(2)`. The `if` condition is `if 2>1`, which is `True` once more, so we move to the following statement where we call `factorial_rec(1)`. Again, we follow the orange arrow and move to (d), leaving the function open in (c) and temporarily disregarding all its remaining code.
- In (d), `n` is `1`, thus the header is `def factorial_rec(1)`. The `if` condition is `if 1>1`, which is `False`! Therefore, we skip the statement under the if and we directly go to the statement under the `else`, which says: `return 1`.

We have reached the so-called **base case**. At this point, we **start the "way up"**. Let's go through the black numerical squares and arrows to collect the returned values and close the functions:

- In (d), the return is `1` and we pass it to the function call in (c), as indicated by the black arrow. The function in (d) is terminated.
- In (c), we complete the return statement under the `if` condition—that is, `return factorial_rec(1)*2`. Thus, we multiply the output of `factorial_rec(1)`,—which is `1` from (d)—by `2`, obtaining 2, which we pass to the function call in (b), as indicated by the black arrow. The function in (c) is terminated.
- In (b), we complete again the return statement under the `if` condition. Thus, we multiply the output of `factorial_rec(2)`,—which is `2` from (c)—by `3` and we obtain 6, which we pass to the function call in (a), as indicated by the black arrow. The function in (b) is terminated.
- Finally, in (a), we complete the return statement under the `if` condition for the last time. We multiply the output of `factorial_rec(3)`,—which is `6` from (b)—by `4`, obtaining 24, which we pass to the output variable `fact` in the initial call, as indicated by the last black arrow. The function in (a) is terminated, as well as the whole recursion.

Now that the functioning mechanism is clear, let's briefly formalize the **syntax** of recursive functions. They typically contain an **`if/else` construct** where statements return or print a value. One of the two statements is called **base case** because it ensures that the **recursion will stop**—in our example, `return 1` (line 22). The other statement is called **recursive case** because it **contains a call to the function itself**—in our example, `return factorial_rec(n-1)*n` (line 18).

Let's conclude with some advantages and disadvantages of recursive functions. On the one hand, recursive functions contain **compact code** and are appropriate when solving **intrinsically recursive** problems—see the *In more depth* section at the end of this chapter. On the other hand, they are **compu-**

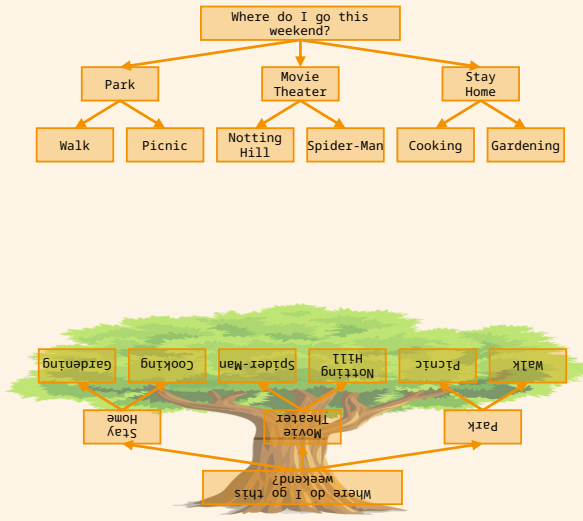
tationally expensive because each call occupies space in memory, which is released only when closing the functions during the "way up". Finally, recursive functions can be **challenging to debug**.

Recap

- Iterative functions contain a loop to repeat some code.
- Recursive functions call themselves to repeat some code.
- Recursive functions typically contain an `if/else` construct, where one statement is the base case, and the other is the recursive case.
- Recursive functions contain compact code and are appropriate for intrinsically recursive problems. However, they use a large amount of computational memory and can be harder to debug.

When do we use recursive functions?

Recursive functions are helpful to solve **intrinsically recursive** problems, that is, when **repeated patterns** are present. Examples include the calculation of factorials—as you learned in this chapter; computations of Fibonacci numbers—see the coding exercise below; and algorithms to




```python
def browse_activities(activity, activities, indent=0):
    """
    Recursively browse the activities dictionary and print its content

    Parameters
    ----------
    activity : string
        The current activity to be printed and explored
    activities : dictionary
        A dictionary where keys are activity names and values are lists
        of sub-activities
    indent : integer
        The number of spaces used for indentation when printing (default
        is 0)
    """

    # print the current activity
    print(" " * indent + activity)

    # if the activity has sub-activities (the value is not an empty list)
    if activities[activity] != []:
        # for each sub-activity
        for sub_activity in activities[activity]:
            # recall the function
            browse_activities(sub_activity, activities, indent + 2)
    # otherwise
    else:
        # stop recursion (base case)
        return


# input decision tree as a dictionary
activities = {
    "Weekend activity": ["Park", "Movie Theater", "Stay Home"],
    "Park": ["Walk", "Picnic"],
    "Movie Theater": ["Notting Hill", "Spider-Man"],
    "Stay Home": ["Cooking", "Gardening"],
    "Walk": [],
    "Picnic": [],
    "Notting Hill": [],
    "Spider-Man": [],
    "Cooking": [],
    "Gardening": []}

# call the recursive function
browse_activities("Weekend activity", activities)
```

```
Weekend activity
  Park
    Walk
    Picnic
  Movie Theater
    Notting Hill
    Spider-Man
  Stay Home
    Cooking
    Gardening
```

Figure 31.2. Example of decision tree (top left); upside-down version of the decision tree that illustrates the similarity to a natural tree (bottom left); and simplified code to traverse the decision tree (right).

search characters in a string—behind methods like `.find()` there are usually recursive functions. Another common recursive problem is traversing **decision trees**, which are sort of flowcharts used to make consecutive decisions among defined alternatives. Nowadays, they are very popular as they are one of the most valid algorithms in machine learning. As an example, let's have a look at Figure 31.2. In this decision tree (top left), we have to decide where to go this weekend, and we must choose among three options: `Park`, `Movie Theater`, or `Stay Home`. After this first choice—for example, `Park`—we must make another more detailed choice—that is, between `Walk` and `Picnic`. At each step of the tree, we repeat the same recursive action—that is, taking a choice—that can be conveniently represented by a recursive function, as you can see in the simplified example in Figure 31.2 (right). Why are decision trees called as such? Because if we turn them upside down, the starting point is like the roots of a tree, the paths through intermediate options are like branches, and the final options are like leaves—see Figure 31.2 (bottom left).

Let's code!

1. *Fibonacci numbers.* You might remember that the Fibonacci sequence is an infinite sequence of numbers where each number is the sum of the two previous numbers.
   a. Modify the code you implemented in Exercise 6 of Chapter 14 into an iterative function that, given a positive integer, returns the Fibonacci number in that position—and not the whole sequence! Make sure to check the input type.
   (Example input: 15. Example output: 610).
   b. Re-implement the function above into a recursive function. *Hint*: this implementation has 2 base cases!

32. How can I reuse functions?

Working with Jupyter Notebooks and Python Modules

In Part 8, we have learned how to write functions, organize them in projects, handle their inputs and outputs, and implement recursion. We also mentioned that functions are a useful way to reuse code because by calling them we avoid copy/pasting long sequences of commands multiple times. But what if we need the same function in more than one notebook? We obviously don't want to copy/paste it across notebooks! The solution is to use Python modules. In this chapter, you will learn how to create a module, how to import it into a notebook and use its functions, and how to modify it. In other words, you will learn a new way of working! As an example, we will reuse the functions to create username, password, and a database that we implemented in Chapter 29. Follow along with Notebook 32! On the website, you will also find the whole module that we will build in this chapter.

1. Creating a module

Do you remember what a module is? In Chapter 15, while learning about `random`, we defined a module as a **unit** containing **functions** for a **specific task**. Let's now refine this definition to be more precise:

> A **module** is a **file** containing **functions** for a **specific task** and whose extension is **`.py`**

Let's jump right in and create a module! The first thing to do is to open an integrated development environment (IDE)[1]. As we mentioned in the *Introduction* of this book, there are several IDEs, all equally valid. For convenience, we will use Spyder. To **access Spyder**:
1. *From Anaconda.* If you installed Anaconda (see the *Getting ready* Part at the beginning of this book), open Anaconda and double-click the *Launch* button in the Spyder tile—box 1 in Figure 32.1.
2. *Install the software.* If you didn't install Anaconda, download Spyder from `www.spyder-ide.org/` and install it like any other software—click *Next* or *Continue* when requested and accept the license agreement. Once installed, open Spyder by clicking on its icon.

When Spyder opens, you should see something very similar to the graphical user interface (GUI) represented in Figure 32.1 (right). On the left side of the GUI, there is a **panel to browse and edit modules** (box 2). In this case, there's only one open module called `untitled0.py` or `temp.py` (box 3).

When using modules, we can perform standard **file actions** using mouse clicks or keyboard shortcuts. To get familiar with some file actions, try to:

- *Close a module*: Click on the *x* on the left side of the file name (box 3); or click on *File* (box 4), and then *Close*.
- *Create a new module*: Click on the *New file* icon (box 5); or click on *File* (box 4), and then *New file*; or press *Command* if you are on Mac—*Control* if you are on Windows—and the letter N simultaneously on the keyboard.
- *Save a module*: Click on the *Save file* icon (box 6); or click on *File* (box 4), and then *Save*; or press *Command* if you are on Mac—*Control* if you are on Windows—and the letter S simultaneously on the keyboard.

[1]Modules can also be opened in JupyterLab by clicking on *File*, *New*, and *Python File*. However, IDEs are usually preferred when working with modules

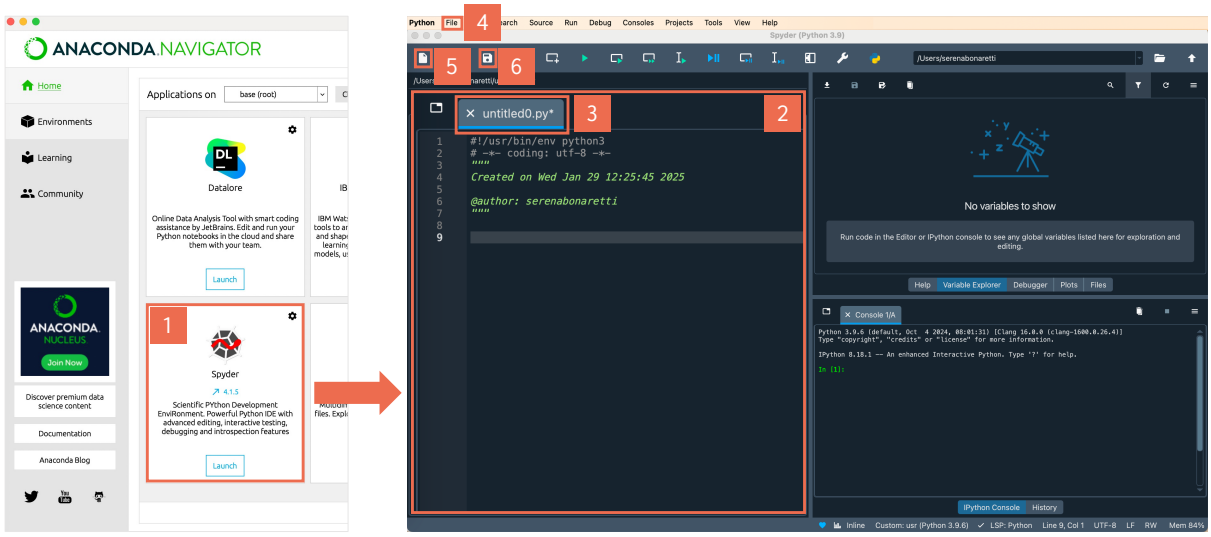


Figure 32.1. Spyder launcher in Anaconda (left) and Spyder graphical user interface (right).

Now, let's **create a module** for the setup of a database of usernames and passwords. Create a new module file—using mouse clicks or keyboard presses, as you have just learned. Save the module as `setup_database.py` in the same folder as Notebook 32. **The rules for module naming are the same as for variable naming**—lowercase and word separation by underscore. You should get something similar to what is shown in box 1 in Figure 32.2.

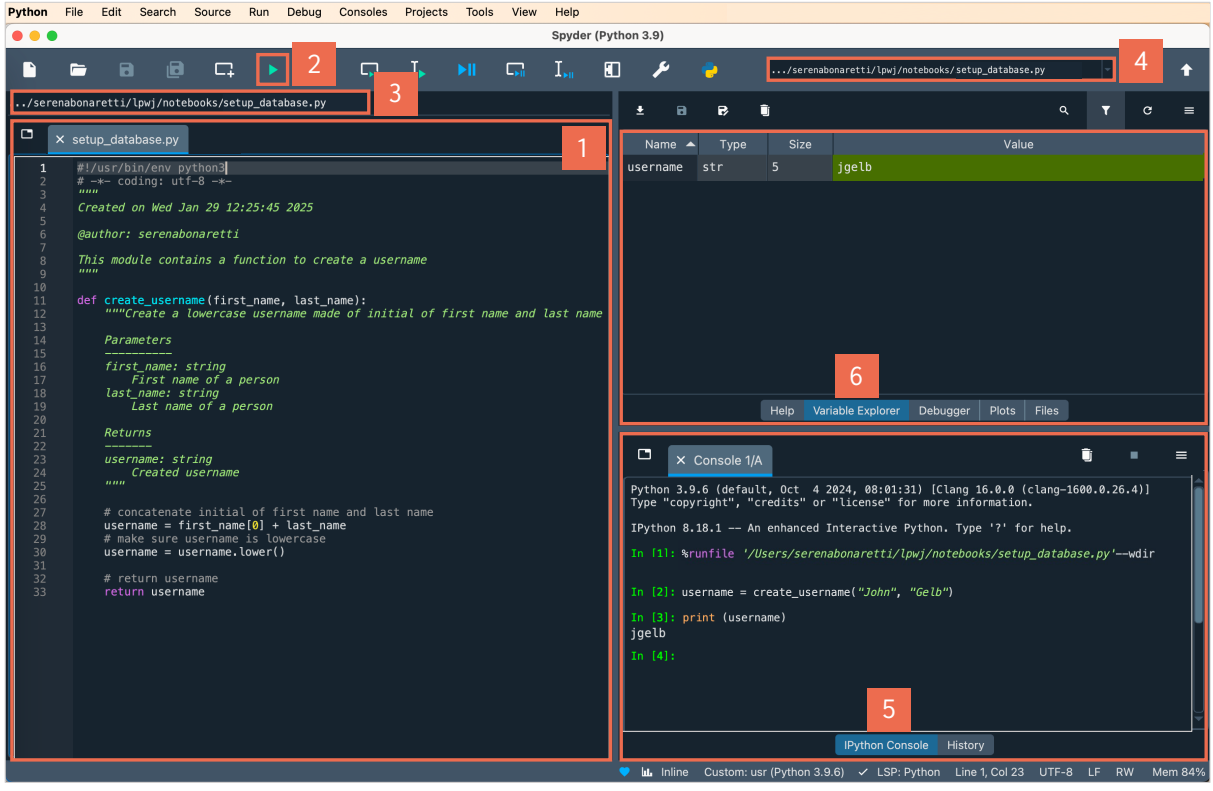


Figure 32.2. Function call from a module in Spyder.

The module starts with two default comments containing information for our computer (lines 1 and 2 in Figure 32.2, box 1)—you can keep or delete them, as you wish. Then, there are the **docstrings containing the module documentation** (lines 3–9). Like for functions, module docstrings are enclosed in double quotes repeated three times. Spyder automatically creates docstrings, and you can edit them as you wish. In their default version, they contain the date of the file creation (line 4) and the name of the author (line 6). We add a simple description of the module content for completeness (line 8). Paste the function `create_username()` from Chapter 29 below the docstrings (lines 11-33). Congratulations, you have just created your first module!

Can we execute `create_username()` directly in Spyder? Yes! First, we need to click the *Run file* button (box 2)—this is one of several convenient methods. By doing so, the folder where the file is saved (box 3) and the **working folder**—that is, where the code is executed—(box 4) become the same, making the content of the module available. Then, we move to the **IPython console** (box 5)—or simply console. There, we **call the function** by typing `username = create_username("John", "Gelb")` and we press *Enter* on the keyboard to execute the code—differently from Jupyter Notebook where we press *Shift* and *Enter*. In the following line, we print the function output to check for correctness. So, we type `print(username)`, press *Enter*, and see the username `jgelb` printed to screen. As you might have noticed, each coding line starts with `In []:`, where `In` stands for input and the square brackets `[]` contain the order of execution of the current command, as if it was a Jupyter Notebook cell. This is because **IPython is a precursor of Jupyter Notebook**—and is still its basic engine. Obviously, in the console you can write **any Python command** as you would in a Jupyter Notebook! Finally, in Spyder, we can keep **control of the created variables** in the **Variable Explorer** (box 6). In our case, we have only one variable whose name is `username`, type is `str`, size is `5`—that is, it is composed of five characters—and value is `jgelb`. Try creating new variables in the console and then take a look at how they are represented in the Variable Explorer!

2. Importing a module and running a function

Let's go back to our notebook and call `create_username()` from there! We need to (1) **import the module** and (2) **call the function**. There are four ways to do it. Let's go through them!

- Import the module as is, then call the function:

```
[1]:
1  import setup_database
2
3  username = setup_database.create_username(
   "John", "Gelb")
4  print(username)
jgelb
```

```
import setup database

username is assigned setup database dot
create username John Gelb
print username
```

We import the module using (1) the **keyword `import`** followed by (2) the **module name without the extension `.py`** (line 1). By doing this, we essentially instruct Python to locate the file `setup_database.py` and make its content available in the current notebook. Now, we can call the function. The syntax is: (1) **module name**, (2) **dot**, and (3) **function name** —as in `setup_database.create_username()` (line 3). We complete the function call by adding its arguments—`"John"` and `"Gelb"`—and we assign the output to the variable `username`, which we print to check for correctness (line 4). Does this syntax remind you of anything? These are the rules that we learned for the module random (Chapter 15), which—now, you know!—is simply a file called `random.py` hosted in a folder somewhere on your computer. To import

the random module, we write `import random`, and to call its functions we type `random.randint()` or `random.choice()`, as we do for any other module. As you can see, we are unveiling more and more secrets of Python!

- Import a module and create an alias, then call the function:

```
[2]: 1 import setup_database as sdb
     2
     3 username = sdb.create_username("John", "Gelb")
     4 print(username)
jgelb
```

```
import setup database as sdb

username is assigned sdb dot create username John Gelb
print username
```

When the module name is long, it can be tedious to retype it every time we call a function. To overcome this, we can use **an alias**—usually an **an abbreviation**. In our example, we import the module `setup_database` and we rename it as `sdb` (line 1). The syntax is: (1) keyword `import`, (2) module name, (3) **keyword `as`**, and (4) **alias**. Then, we call the function using: (1) **alias**—instead of the full name—(2) dot, and (3) function name—that is, `sdb.create_username()` (line 3). Like before, we add the arguments—`"John"` and `"Gelb"`—and we assign the output to `username`, which we print to check for correctness (line 4). Note that the use of aliases is especially common when using **Python packages**—you might have seen `import pandas as pd` or `import numpy as np`. Packages are just **collections of modules**—that is, of files ending in `.py`—and whose imports and function calls work the same way as for modules.

- Import one single function from a module, then call the function:

```
[3]: 1 from setup_database import create_username
     2
     3 username = create_username("John", "Gelb")
     4 print(username)
jgelb
```

```
from setup database import create username

username is assigned create username John Gelb
print username
```

We consider the case where we need **only one function from a module** that contains several functions—we will see how to create such modules in a bit. Importing the entire module could occupy excessive space in memory that instead we want to use for computations. Thus, we import only one single function by writing: (1) **keyword `from`**, (2) module name, (3) keyword `import`, and (4) function name without round brackets (line 1). To call the function, we **directly use the function name** without the module name (line 3). Finally, we print to check the function output (line 4).

- Import the function from a module and create an alias, then call the function:

```
[4]: 1 from setup_database import create_username as cu
     2
     3 username = cu("John", "Gelb")
     4 print(username)
jgelb
```

```
from setup database import create username as cu

username is assigned cu John Gelb
print username
```

In this last case, we import one single function whose name is very long, and thus, we want to rename it with an alias. To import the function, the syntax is: (1) keyword `from`, (2) module name, (3) keyword `import`, (4) function name, (5) keyword `as`, and (6) alias (line 1). To call the function, we directly use the alias—that is, `cu`—(line 3). Finally, we print the output (line 4). Note that this solution is rarely ideal

because it can compromise code readability.

Let's conclude this section with two important remarks about imports. First, in a notebook, **imports should be grouped in the very first cell**, before any computation—here, they are spread across different cells for sake of explanation. Second, we can **import modules into other modules**. For example, we can import the module `random` in `setup_database`, if needed. Like in notebooks, all imports in modules should be grouped **at the very beginning of the file** to improve code readability.

3. Importing a module from a different folder

What if we want to use a module that is in a different folder than the notebook? As you can guess, we do not want to have multiple copies of the same module! As an example, the module `setup_database` is in the folder `"/Users/serenabonaretti/lpwj/notebooks"` on my computer (box 3 in Figure 32.2.). Let's say that I need to move it to `"/Users/serenabonaretti/lpwj/code"`. How can Python know where the module is now? We tell Python **where to find a module** using the following commands.

- Restart the kernel, then add the module folder and import the module:

```
[1]:
1  import sys
2  sys.path.append("/Users/serenabonaretti/lpwj/code")

3  import setup_database
```

```
import sys
sys dot path dot append Users
serenabonaretti lpwj code
import setup database
```

We restart the kernel to clear the previous imports and make sure that we are correctly testing the code in this section. Then, we import a **built-in module** called **sys** (line 1), which manages file locations—among other things. From `sys`, we call the command `sys.path.append()` (line 2), which takes a string containing the module **path**—that is, its location—as an argument—in this example, `"/Users/serenabonaretti/lpwj/code"`. Note that we have to insert the whole module path, which starts with `/Users` in macOs, and with `C:\`—or another letter of the alphabet—in Windows. Now that Python knows where `setup_database` is, we can import it (line 3). Let's call the function `create_username()` using the same commands as in cell 1 of section 2.

- Call the function:

```
[2]:
1  username = setup_database.create_username("John", "Gelb")

2  print(username)
```

```
jgelb
```

```
username is assigned
setup database dot create
username John Gelb
print username
```

Why don't you test these new commands? Move your module `setup_database` to another folder, then add the new file location to `sys.path.append("")`. Go back to your notebook, import the module, and call the function!

For the next section, let's bring the module back to the same folder as the notebook for convenience.

4. Changing a function in a module and calling it in a notebook

What if we need to make a change to the function that we call in the notebook? Let's call `create_username()` before and after a change and see what happens!

- Restart the kernel, then call the function before the change:

```
[1]:
1  import setup_database
2
3  username = setup_database.create_username("John", "Gelb")
4  print(username)
jgelb
```

```
import setup database

username is assigned
setup database dot create
username John Gelb
print username
```

We restart the kernel to clear all previous imports and variables from memory. Then, we use the same code as we used in cell 1 to import the module (line 1), call the function (line 3), and print the output (line 4).

Now, let's go to the module in Spyder and make a simple change to the function. For example, let's add `print(first_name, last_name)` at the beginning of the function body, as you can see at lines 27–28 in Figure 32.3.

```
11  def create_username(first_name, last_name):
12      """Create a lowercase username made of initial of first name and last name
13
14      Parameters
15      ----------
16      first_name: string
17          First name of a person
18      last_name: string
19          Last name of a person
20
21      Returns
22      -------
23      username: string
24          Created username
25      """
26
27      # print inputs
28      print(first_name, last_name):
29
30      # concatenate initial of first name and last name
31      username = first_name[0] + last_name
32      # make sure username is lowercase
33      username = username.lower()
34
35      # return username
36      return username
```

Figure 32.3. Module function modified at line 28.

After a change, **always remember to save the file!** *Tip*: when the module is not saved, you will see an **asterisk** in the name tab in the module panel. The asterisk will disappear after you save the file (see Figure 32.4).

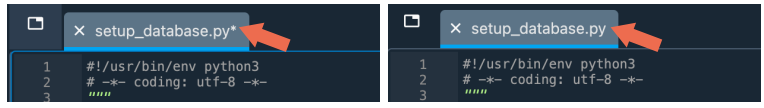


Figure 32.4. Module tab with an asterisk before saving (left) and without an asterisk after saving (right).

Time to call the function again!

- Call the function after the change:

```
[2]:
1  username = setup_database.create_username("John", "Gelb")
2  print(username)
jgelb
```

```
username is assigned
setup database dot create
username John Gelb
print username
```

We call `create_username()` (line 1) and print the output (line 2). We get only one print, that is, the username from line 2, although we also expected to see a print containing first name and last name from line 28 of the module. Why does this happen? Because the module and the notebook are not automatically synchronized! When we import a module, the notebook memorizes it as is and does not automatically detect any change that we make afterwards. To **synchronize notebook and module**, we need to use `autoreload`, an IPython extension that **automatically re-imports all modules every time we execute a cell**—as you will see in a bit. What is an **IPython extension**? It is a **module that adds extra functionalities to IPython**—the core of Jupyter Notebook. Extension commands often **start with a percentage symbol**—they are called **magic commands**!—and can be used only in Jupyter Notebooks—not in Python. Let's see `autoreload` in action!

- Restart the kernel, then run `autoreload`:

```
[1]:
1  %load_ext autoreload
2  %autoreload 2
```

```
percentage load ext autoreload
percentage autoreload 2
```

Similarly to above, we restart the kernel to have a clear memory. Then, we load `autoreload` (line 1), and we run `autoreload` with the **parameter 2**, which indicates that we will **reload all the imported modules** every time we run a cell (line 2). Note that these two lines of code must always be located **in the very first cell of the notebook**—that is, even before the imports—to be effective. Now, let's call the function before the change—comment out line 28 in the module!—and after the change to test `autoreload`.

- Call the function before the change:

```
[2]:
1  import setup_database
2
3  username = setup_database.create_username("John", "Gelb")
4  print(username)
```

```
jgelb
```

```
import setup database

username is assigned
setup database dot create
username John Gelb
print username
```

As expected, we get the print `jgelb` from line 4. Now, let's make the change in the module again by re-adding `print(first_name, last_name)` at line 28 of the module. **Don't forget to save the file!** It is a **common mistake** not to see the changes reflected in the output because we forgot to save the file! Let's call the function once more, and finally see the effect of `autoreload`.

- Call the function after the change:

```
[3]:
1  username = setup_database.create_username("John", "Gelb")
2  print(username)
```

```
John Gelb
jgelb
```

```
username is assigned
setup database dot create
username John Gelb
print username
```

Finally, we get the print from line 28 of the module—`John Gelb`. This is because `autoreload` automatically—and secretly!—imported the latest version of `setup_database` before executing the Python commands.

5. Adding functions to a module

As you can imagine, we can add as many functions as we want to a module. However, we should add only the **functions that solve a specific problem or serve a specific task**. For example, in the module `setup_database`, it's meaningful to add `create_password()` and the main function `create_database()` from Chapter 29—see Figure 32.5 It would be inappropriate to add a function that implements—let's say—a calculator because it would be off topic—for that, we would create a separate module. When adding a new function, it is always important to remember three aspects: first, to **update the description** of the module content—see line 8 in Figure 32.5; second, add all **the imports at the very beginning** of the module—see `import random` at line 11 in Figure 32.5; and third, **position the functions in a logically meaningful place** within the file—in our example, first the satellite functions `create_username()` (lines 14—36) and `create_password()` (lines 39—52), and then the main function `create_database()` (lines 55-90).

Copy `create_password()` and `create_database()` from Chapter 29 to the module `setup_database`. Then, let's move to the notebook and call each function to check that they correctly work.

- Call `create_password()`:

```
[4]: 1  password = setup_database.create_password()
                                                        password is assigned setup database
                                                        dot create password
     2  print(password)                                  print password
     4056
```

- Call `create_database()`:

```
[5]: 1  customers = [["Julia", "Smith"], ["Mohammed",   customers is assigned Julia Smith
       "Seid"], ["Maria", "Lopez"]]                     Mohammed Seid Maria Lopez
     2  db, n = setup_database.create_database(customers) db n is assigned setup database
                                                        dot create database customers
     3  print("Database", db)                           print Database db
     4  print("Number of customers:", n)                print Number of customers n
     Database: 'jsmith': '5092', 'mseid': '3543', 'mlopez': '4476',
     Number of customers: 3
```

As you might have guessed, the call of **any** function from a module follows the syntax that we learned—that is, (1) module name, (2) dot, and (3) function name (line 1 in cell 4 and line 2 in cell 5). Note that we did not need to restart the kernel because we previously ran `autoreload`, and thus the addition of `create_password()` and `create_database()` are automatically detected.

In this chapter, we learned a **new way of working** that allows us to reuse functions across various projects. In general, we use **Jupyter Notebooks** to **draft code, test it quickly, and integrate narrative** for context—and to learn Python! On the other side, we use **Python modules** to **store functions** that can be called in various **execution environments**, including **Spyder**, **Jupyter Notebook**, and the **terminal**—as you will see in the *In more depth* section at the end of this chapter. Play around with these tools and find your most comfortable way of coding!

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Wed Jan 29 12:25:45 2025

@author: serenabonaretti

This module contains functions to create usernames and passwords, and add them to a database
"""

import random


def create_username(first_name, last_name):
    """Create a lowercase username made of initial of first name and last name

    Parameters
    ----------
    first_name: string
        First name of a person
    last_name: string
        Last name of a person

    Returns
    -------
    username: string
        Created username
    """

    # concatenate initial of first name and last name
    username = first_name[0] + last_name
    # make sure username is lowercase
    username = username.lower()

    # return username
    return username


def create_password():
    """Create a password composed of four random integers

    Returns
    -------
    password: string
        Created password
    """

    # create a random number with four digits
    password = str(random.randint(1000,9999))

    # return password
    return password


def create_database(customers):
    """Creates a database as a dictionary with usernames as keys and passwords as values

    Parameters
    ----------
    customers : list of lists
        Each sublist contains first name and last name of a customer

    Returns
    -------
    db : dictionary
        Created database (shorted as db)
    n_customers : int
        Number of customers in the database
    """

    # initialize dictionary (i.e. database)
    db = {}

    # for each customer
    for customer in customers:

        # create username
        username = create_username(customer[0], customer[1])

        # create password
        password = create_password()

        # add username and password to db
        db[username] = password

    # compute number of customers
    n_customers = len(db)

    # return dictionary and its length
    return db, n_customers
```

Figure 32.5. The module `setup_database`.

✎ Insert into the right column

Group the new syntax and concepts from this chapter in the following table for a structured overview.

`from, Spyder, random, autoreload, Terminal, import, as, Jupyter Notebook, sys`

Python keywords	Python built-in modules	Jupyter Notebook extension	Execution environments

Recap

- A module is a file containing functions for a specific task and whose extension is `.py`.
- In Spyder, we can create and manage modules, and call their functions.
- To import a module we can use four different syntaxes resulting from the combinations of the keywords `import`, `from`, and `as`.
- A package is a collection of modules whose imports and function calls works the same way as for modules.
- To import a module from a separate folder, we use the module `sys` and its command `sys.path.append()`.
- An IPython extension is a module that adds extra functionalities to IPython, which is the core of Jupyter Notebook. Extension commands often start with a percentage symbol and are called magic commands.
- To synchronize a module and a notebook, we use the IPython extension `autoreload`.
- A module can contain as many functions as needed to solve a task. It is good practice to structure a module with (1) documentation, (2) imports, and (3) functions in a logically meaningful order.
- Jupyter Notebook, Spyder, and the terminal are complementary tools to write code.

What is: if __name__ == “__main__”?

In this chapter, we learned to call functions from a module. However, modules can also be **executed directly**, that is, without manually running cells or calling specific functions. Have you ever seen the command: `if __name__ == "__main__"`? That's what we need! Let's go back to the module `setup_database` that we created in this chapter and add some extra lines of code **at the end of the file** (Figure 32.6 (top)). First, we write the predefined condition **`if __name__ == "__main__"`**, which tells the computer something like “if you run this file as a module, execute the following commands” (line 93). Then, we write the code to execute. In our case, we create the variable `customers` containing a list of strings with customer names (line 96), call

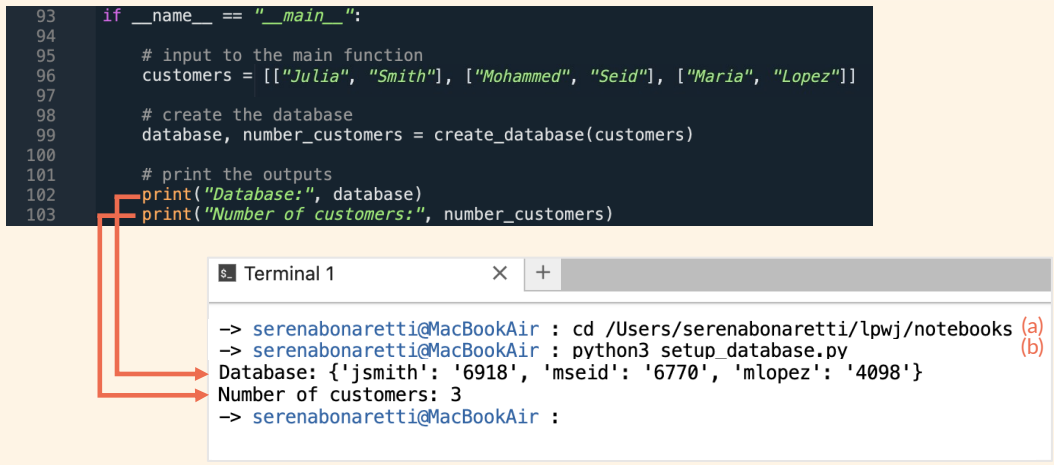


Figure 32.6. Code to execute a module (top) and to call it in a terminal (bottom).

`create_database()` (line 99), and print its outputs (lines 102–103). Modules can be run in Spyder by simply pressing the *Run* command (box 2 in Figure 32.2). However, the most common way is to use a **terminal**, which is usually preferred **for larger projects** because it runs code in a **fast and automated** way (Figure 32.6 (bottom)). To open a terminal, go to JupyterLab, and click on *File*, *New*, and *Terminal*[a]. Note that in a terminal, we can use only **specific command lines**—not Python! To execute a module, we need two commands. First, we tell the terminal where the module is. To do so, we type **`cd`**—abbreviation of change directory—followed by the **whole path** of the folder containing the module—in this example, `/Users/serenabonaretti/lpwj/notebooks` (line (a)). We press *Enter* to execute the command. Then, we execute the module by typing (1) the command `python3` (or simply `python` in some terminals), and (2) module name with the extension `.py` (line (b)). We press *Enter* to execute the command. When the functions are executed, the prints from lines 102 and 103 of the module appear on the terminal, as indicated by the orange arrows in Figure 32.6. Go back to `setup_database`, add the `if __name__ == "__main__"` construct, and run it from Spyder and from terminal. Which way do you prefer?

[a]Alternative ways to open a terminal include: if you are on a macOS, type *terminal.app* in the *Spotlight Search* bar and then press *Enter*; if you are on a Windows, press the *Windows key*, type *cmd*, and press *Enter*. If you use JupyterLite, you might have to open a terminal using one of these two ways.

Let's code!

1. A *module for Bill Gates!* Create a module containing the main and satellite functions you wrote for the coding exercise 1 in Chapter 29 *What does Bill Gates tweet about?* Import the module in a notebook and call the main function. Then, execute the module in a terminal.
2. A *module for hangman!* Do the same as in the previous exercise for the functions of the coding exercise 1 in Chapter 30 *Let's play hangman!*

PART 9

LAST BITS OF BASIC PYTHON

In this part, you will learn how to read and write files and some final types, keywords, built-in functions, and modules. Enjoy it!

33. Birthday presents

Reading and writing .txt files

One important aspect in coding is to know how to read and write files. Let's learn the basics in this chapter! Below is the task we want to solve—follow along with Notebook 33!

- Three of your friends celebrated their birthday this month, and you bought them presents online. Now, it's time to review what you spent and keep a record of it. The purchase amounts are in the file `33_purchases.txt`.

How are we going to solve this task? Outline the steps you would take before jumping into the solution below.

To solve the task, we will divide (and conquer!) the problem in three steps. First, we will read the `.txt` file and store its content into a list. Second, we will analyze the purchases by calculating the minimum, maximum, and total of the amounts in the list. And third, we will save the analysis into a new `.txt file`. Let's start!

1. Reading a .txt file

Before we start writing the code, it's important to see **what the input file looks like**. Download and save `33_purchases.txt` in the same folder as the notebook and open it by double clicking on it either in the File Browser—i.e., the left panel—in JupyterLab or in the folder where it is located. You will see something similar to Figure 33.1.

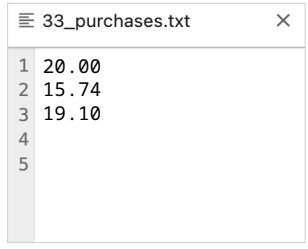


Figure 33.1. Input file.

The file contains three numbers, each on a separate row, representing purchase amounts. Let's read the file and store the three numbers into a list.

- Write a function that reads a `.txt` file containing one number per row and stores the numbers into a list:

[1]:
```
def read_txt(file_name_in):
    """Reads a .txt file with one number
    per row and returns them as a list

    Parameters
    ----------
    file_name_in : string
        Name of the file to read
```

```
def read txt file name in
Reads a .txt file with one number
per row and returns them as a list

Parameters

file name in : string
Name of the file to read
```

```
8
9       Returns                                    Returns
10      -------
11      numbers : list                             numbers : list
12          File content in a list of numbers      File content in a list of numbers
13      """
14
15      # initialize output                        initialize output
16      numbers = []                               numbers is assigned empty list
17
18      # open the file to read                    open the file to read
19      with open(file_name_in, "r") as file:      with open file name in r as file
20
21          # read the file                        read the file
22          for line in file:                      for line in file
23              print("line as read:", line)       print line as read line
24
25              # remove "\n" from line            remove backslash n from line
26              line = line.rstrip("\n")           line is assigned line dot rstrip
                                                   backslash n
27              print("line after stripping:", line)  print line after stripping line
28              print("-----")                     print dashes
29
30              # get only the non-empty lines     get only the non-empty lines
31              if line != "":                     if line not equal to empty string
32
33                  # transform the number to float  transform the number to float
34                  number = float(line)           number is assigned float line
35
36                  # add to the output list       add to the output list
37                  numbers.append(number)         numbers dot append number
38
39      # return the output                        return the output
40      return numbers                             return numbers
41
42  # call the function and print the output       call the function and print the
                                                   output
43  purchases = read_txt("33_purchases.txt")       purchases is assigned read txt 33
                                                   purchases dot txt
44  print("purchases:", purchases)                 print purchases purchases
```

```
(a)  line as read: 20.00
(b)
(c)  line after stripping: 20.00
(d)  -----
(e)  line as read: 15.74
(f)
(g)  line after stripping: 15.74
(h)  -----
(i)  line as read: 19.10
(j)
(k)  line after stripping: 19.10                  ↳
```

```
(l)   -----
(m)   line as read:
(n)
(o)   line after stripping:
(p)   -----
(q)   purchases: [20.0, 15.74, 9.1]
```

What happens in the function? Get some hints by completing the following exercise!

✎ True or false?

1. The variable `file_name_in` is a string. T F
2. `with` and `open()` are a built-in function and a keyword, respectively. T F
3. In the `for` loop, we read the file content one line at the time. T F
4. A file cannot contain empty rows. T F

Computational thinking and syntax

Let's analyse the function `read_txt()`. It reads a text file containing one number per row and returns the numbers as a list (line 2). The input is the string `file_name_in`—that is, the name of the file to read (lines 4—7)—and the output is the list `numbers` (lines 9—12). In the function body, first we initialize the output `numbers` to an empty list (line 16). Then, we open the file and store its content into a variable by using a standard command composed of four elements (line 19): (1) the keyword **`with`**, which supports proper **file opening and automatic closing**; (2) the built-in function **`open()`**, which **opens and reads the file** by taking two arguments—the file name and the string `"r"` for reading—and returns the file content; (3) the keyword **`as`**; and (4) a variable commonly named **`file`**—or `file_object`, or its abbreviation `fo`—representing the file with its content. If we print the variable `file`, we get something like `<_io.TextIOWrapper name='33_purchases.txt' mode='r' encoding='UTF-8'>`—you can try this on your notebook! To actually extract the file content, we have to write an indented `for` loop through values that, at each iteration, stores the current file row into the variable `line` (line 22). When we print `line` for a check (line 23), we get paired prints: (a) and (b), (e) and (f), (i) and (j), and (m), and (n). Why does this happen?

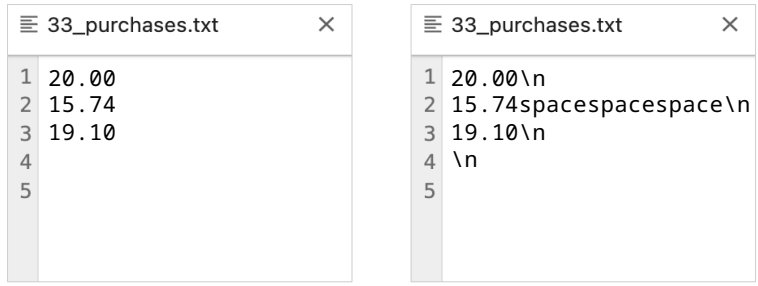


Figure 33.2. How we see a file content (left) vs. how a computer sees a file content (right).

When we **humans** read the file `33_purchases.txt`, we **see three numbers**, each of them in a separate row (Figure 33.2, left). When a **computer** reads the file, it **sees four strings**, each containing some digits and some hidden characters, such as `\n` or space (Figure 33.2, right). Thus, at the first iteration

of the `for` loop, the variable `line` contains the string `20.00\n` (line 22). As a consequence, the print is `20.00` (print (a)) followed by the empty line due to `\n` (print (b)). At the second iteration, `line` contains the string `15.74spacespacespace\n`, and thus the prints are `15.74` followed by three spaces (print (e))—which we humans do not see!—and an empty line (print (f)). Similarly, at the third iteration, `line` contains the string `19.10\n`, and thus the prints are `19.10` (print (i)) followed by empty line (print (j)). Why is there a fourth iteration? Most likely, who created the file accidentally added a new line before saving the file! Thus, we see no content (print (m)) followed by an empty line (print (n)). Now that we know how a computer reads the file, we have to do some cleaning to obtain the list of numbers! First, we **remove the empty lines** by using the string method **`.rstrip()`**, which removes its argument `"\n"` from the right side of each string `line` (line 26)—in `.rstrip()`, `r` stands for right. To test for correctness, we print `line` after stripping (line 27), followed by five dashes to ease visualization (line 28). This time, we obtain the line content—prints (c), (g), (k), (o)—without subsequent empty lines. Then, to **remove the empty line at the end of the file**—whose presence was revealed by the print (n)—we create an `if` condition that excludes empty lines from further processing (line 31). Finally, we **transform the number** in `line` **into a float** using the built-in function **`float()`**, which **automatically ignores spaces** while converting a string into a number (line 34)—like in the case of `15.74spacespacespace\n`. As the last step, we assign the obtained `number` to the list `numbers` (line 37) and we return it (line 40). Finally, to execute the function, we call it with the file name `"33_purchases.txt"` as an input (line 43)—if the file is not in the same folder as the notebook, enter the whole file path! The output is assigned to the variable `purchases` (line 43), which we print to check for correctness (line 44).

Now that we are aware of the details to consider when reading a file, can we write a more compact code? Yes! The whole function body (lines 15–40) can be shortened in the following three lines!

```
15      # open the file
16      with open(file_name_in, "r") as file:
17
18          # read the numbers and transform them
            into floats
19          numbers = [float(number) for number
            in file.read().split()]
20
21      # return the output
22      return numbers
```

```
open the file
with open file_name_in r as file

read the numbers and transform them
into floats
numbers is assigned float number
for number in file dot read dot
split

return the output
return numbers
```

We open the file (line 16) using the same command that we learned above (cell 1, line 19). Then, we read the file, clean its content, and create the output list in one single command containing a list comprehension (line 19). To untangle it, let's read the command starting from the `for` loop: `for numbers in file.read().split()` assign `float(number)` to `numbers`. The method **`.read()`** of the variable `file` **transforms the file content into a long string**—that is, `'20.00\n15.74    \n19.10\n\n'`. Then, the string method `.split()` divides the obtained string at white spaces or new lines and transform it into a list of strings—that is, `['20.00', '15.74', '19.10']`. Finally, at each loop iteration, each list element is transformed into a float using `float(number)` and automatically added to the list `numbers`. Note that since we are using list comprehension, we do not need to initialize the outcome variable `numbers`. At the end of the function, we return the output `numbers` (line 22)—as we did in cell 1, line 40.

Now that the three numbers are in the numerical list, let's do some simple analysis on them!

2. Analyzing the numbers

- Write a function that takes a list of numbers as an input and returns the minimum, maximum, and sum as separate variables.

[2]:

```
def calculate_stats(numbers):
    """Returning minimum, maximum, and sum
    of a list of numbers

    Parameters
    ----------
    numbers : list
        Contains numbers

    Returns
    -------
    minimum : float
        Minimum of the list
    maximum : float
        Maximum of the list
    total : float
        Sum of the list numbers
    """

    # calculate the minimum
    minimum = min(numbers)

    # calculate the maximum
    maximum = max(numbers)

    # calculate the sum
    total = sum(numbers)

    # return the stats
    return minimum, maximum, total

# call the function
mn, mx, tot = calculate_stats(purchases)

print("minimum:", mn)
print("maximum:", mx)
print("total:", tot)
```

```
minimum: 15.74
maximum: 20.0
total: 54.84
```

```
def calculate stats numbers
Returning minimum, maximum, and sum of a
list of numbers

Parameters

numbers : list
Contains numbers

Returns

minimum : float
Minimum of the list
maximum : float
Maximum of the list
total : float
Sum of the list numbers


calculate the minimum
minimum is assigned min numbers

calculate the maximum
maximum is assigned max numbers

calculate the sum
total is assigned sum numbers

return the stats
return minimum maximum total

call the function
mn mx tot is assigned calculate stats
purchases
print minimum mn
print maximum mx
print total tot
```

Computational thinking and syntax

The function `calculate_stats()` takes a list of numbers as an input (lines 4—7) and returns three variables containing minimum, maximum, and total sum of the list numbers, respectively (lines 9–16). To

calculate the minimum, we use the built-in function **`min()`**, which takes a list as an input and **returns the minimum number of the list** (line 20). Similarly, to calculate the maximum, we use the built-in function **`max()`**, which given a list as an input, **returns the maximum number of the list** (line 23). Finally, to calculate the total amount, we use the built-in function **`sum()`** that, given a list as an input, **returns the sum of the numbers in the list** (line 26). We store the three outputs into the variables `minimum`, `maximum`, and `total` making sure that **we do not use the function names**—see the *In more depth* section *Don't name variables with reserved words!* in Chapter 14. At the end of the function, we return the three variables separately (line 29). Finally, we call the function (line 32) and print the three variables `mn`, `mx`, and `tot` to test for correctness (lines 33–35).

As the last step of our task, let's save the results of the analysis into a new text file!

3. Saving the analysis

- Create a function that given the minimum, maximum, and total, writes them to a file on three consecutive lines, specifying what they represent:

[3]:

```
1  def write_txt(file_name_out, minimum,
   maximum, total):
2      """Writing minimum, maximum, and sum
       to a file
3
4      Parameters
5      ----------
6      file_name_out : string
7          Name of the file to write
8      minimum : float
9          Minimum of the list
10     maximum : float
11         Maximum of the list
12     total : float
13         Sum of the numbers in the list
14     """
15
16     # open the file to write
17     with open(file_name_out, "w") as file:
18
19         # write the file content
20         file.write("minimum: " + str(minimum)
           + "\n")
21         file.write("maximum: " + str(maximum)
           + "\n")
22         file.write("total: " + str(total))
23
24 # call the function
25 write_txt("33_purchases_stats.txt", mn, mx,
   tot)
```

```
def write txt file name out minimum
maximum total
Writing minimum, maximum, and sum to a
file

Parameters

file name out : string
Name of the file to write
minimum : float
Minimum of the list
maximum : float
Maximum of the list
total : float
Sum of the numbers in the list


open the file to write
with open file name out w as file

write the file content
file dot write minimum: concatenated
with str minimum concatenated with
backslash n
file dot write maximum: concatenated
with str maximum concatenated with
backslash n
file dot write total: concatenated
with str total

call the function
write txt 33 purchase stats dot txt mn
mx tot
```

What's new in this function? Discover it by completing the following exercise.

✎ True or false?

1. The argument "w" in open() defines that we want to close a file. T F
2. .write() is a method of the variable file. T F
3. When writing a file content, we use string concatenation. T F

Computational thinking and syntax

The function `write_txt()` takes four inputs, that is, the name of the file to write and the calculated minimum, maximum, and total (lines 4—13), and it has no return. We begin the function body with a command that **simultaneously creates and opens a new file** (line 17) and has the same components as the command we used to open and read a file (cell 1, line 19). The difference is the inputs of `open()`, which now are the name of the file to write and the string **`"w"`** for writing—instead of `"r"` for reading. Then, indented with respect to the previous line of code, we add the commands to write the file content. Each command is composed of the variable `file` followed by its method **`.write()`**, which takes **the string to be written into the file as an argument**. Thus, to write the minimum, the argument is `"minimum: "` concatenated with the minimum converted into a string—`str(minimum)`—and the escape character `"\n"` to send the following content to a new row (line 20). Similarly, to write the maximum, the argument is `"maximum: "` concatenated with `str(maximum)` and the escape character `"\n"` (line 21). Finally, to write the total, the argument is `"total: "` concatenated with the string of `total`. In this last case, we do not add `"\n"` because we do not want to add an empty line at the end of the file (line 22). To conclude, we call the function (line 25), and we obtain the new file `33_purchases_stats.txt`. Check the file by opening it in the File Browser in JupyterLab or in the folder. You will see something similar to Figure 33.3.

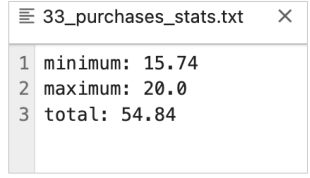


Figure 33.3. Output file.

In this chapter, we learned how to read a file, analyze its content, and write a new file. Did you also notice how the various **functions interact** among each other? We organized `read_txt()`, `calculate_stats()` and `write_txt()` into a **pipeline** where **the output of each function is the input of the following function** (see Figure 33.4). In general, pipelines are a convenient way to solve problems with **linear solutions** and are an alternative to the organization in main and satellite functions—which we learned in Chapter 29. Depending on the nature of the solution, we can use different function configurations to divide and conquer our computational problem!

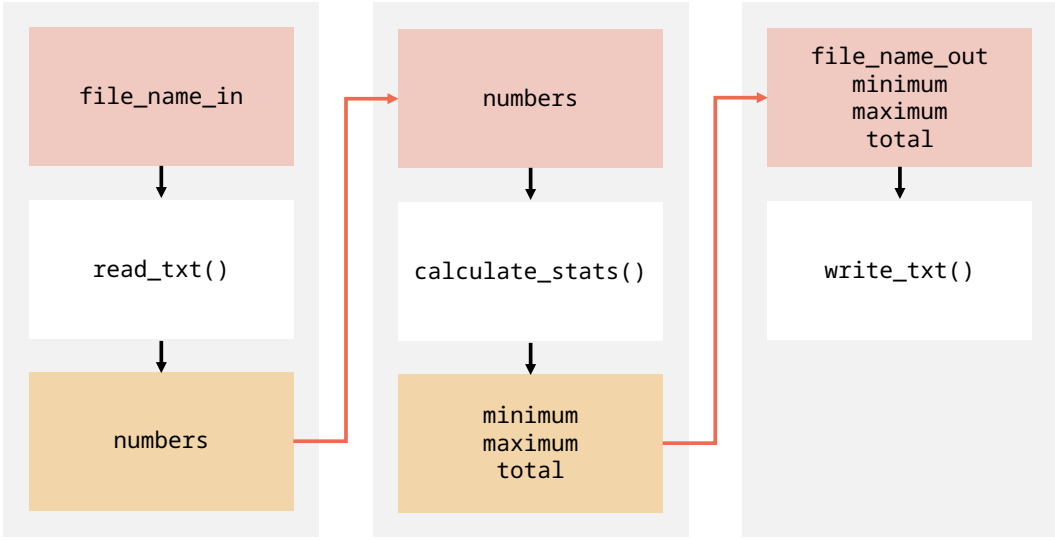


Figure 33.4. Pipeline of functions, where the output of a function (yellow rectangle) is the input of the following function (orange rectangle), as shown by the orange arrow.

Recap

- To open–and–read or create–and–save a file, we use the command `with open() as file`, where:
  - `with` is a keyword that supports opening and closing a file;
  - `open()` is a built-in function whose arguments are the name of the file to read or write and the strings `"r"` when reading or `"w"` when writing the file;
  - `as` is a keyword to rename the variable representing the file;
  - `file` (or `file_object` or `fo`) is a conventional file name for the variable representing the file.
- To read a file content, we use either a `for` loop that goes through each line of the file content, or the method `.read()` that returns the file content as a long string. To clean the file content, we use string methods such as `.rstrip()` or `.split()`.
- To write content to a file, we use the method `.write()`, which takes a string (or a concatenation of strings) as an argument.
- To calculate minimum, maximum, and sum of a list, we use the built-in functions `min()`, `max()`, and `sum()`.
- To solve problems with linear solutions, we organize functions in pipelines where the output of the previous function is the input of the following function.

How do I organize folders and files?

When coding, we deal with several kinds of files, including modules, notebooks, and data. As projects become larger and more complex, it is important to organize folders and files properly to avoid confusion. A common way is to have a hierarchical structure that starts with a **project folder** enclosing the whole project and containing sub-folders for the sub-projects. Within each sub-folder, it is common to have one folder for **code**—which can contain two sub-folders, one for

notebooks and one for modules—one folder for **data**—with one or more datasets—and one folder for **documentation**—such as a website, a readme file, or any other document (Figure 33.5, left). This general structure can be adapted to the material of a specific project. For this book, we can create a project folder called `learn_python_with_jupyter`, organized in sub-folders corresponding to each chapter and structured according to their content. For example, the folder `32_modules` contains just a subfolder `code` with the module `setup_database.py` and the notebook `32_modules_.ipynb`. The folder `33_read_write_file` contains two folders, that is, `code` with the notebook `33_rw_txt_file.ipynb`, and `data` with the input file `33_purchases.txt` and the output file `33_purchases_stats.txt`. Finally in the project directory, there is the folder `documentation` containing the `.pdf` of the book (Figure 33.5, right). Every time we approach a new project, let's start by organizing folders and files in a structured way. It is the first step to divide and conquer with clarity!

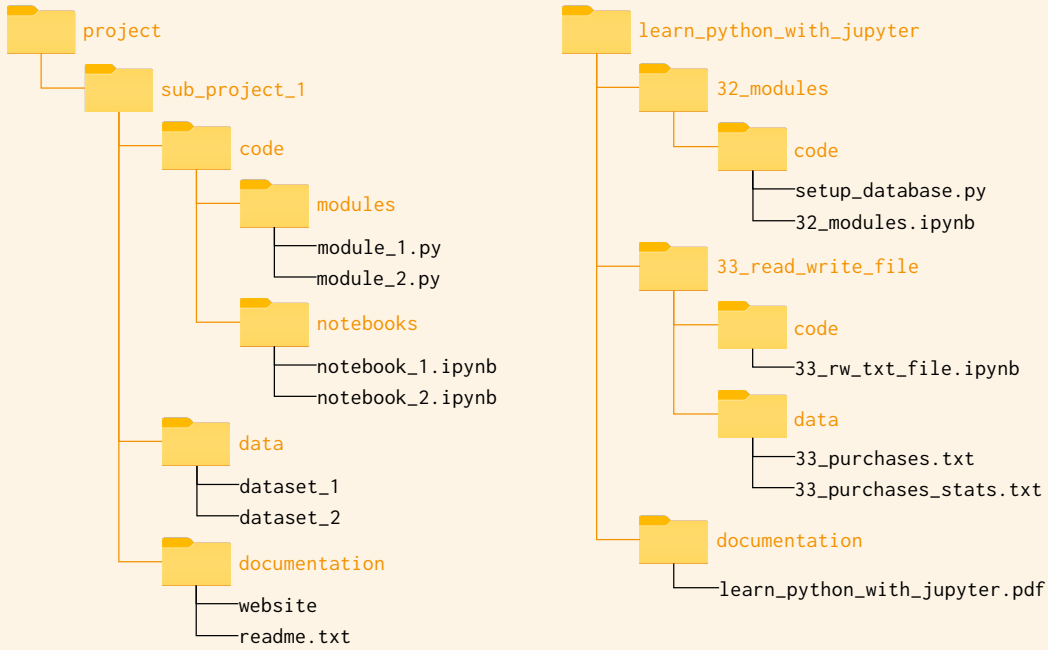


Figure 33.5. Schematics of a common folder and file structure for computational projects (left) and example for the material of Chapters 32 and 33 of this book (right).

Let's code!

1. *Login database for Hollywood actors.* You work at a famous Hollywood agency and need to create usernames and passwords for the actors' accounts.
   a. *Reading input file.* Create a function that reads the file `actors.txt`—download it from the website—and returns a list of lists where each sub-list contains first name and last name of an actor. *Hint*: What string method can you use to separate first names and last names?
   b. *Creating usernames and passwords.* Write a function that reads a list of lists where each sub-list contains first name and last name of an actor, then computes username and password for each actor, and finally returns a list of lists where each sub-list contains a username and a password. Call the functions from the module `setup_database` (Chapter 32).

c. *Writing usernames and passwords to file.* Create a function that, given a list of lists where each sub-list contains a username and a password, writes the list to a `.txt` file where each row contains username and password of an actor. Make sure not to create an empty line at the bottom of the file!

34. What's more in Python?

Additional types, keywords, built-in functions, modules

We are getting to the end of our journey in developing computational thinking while learning Python. Before getting to the last part of this book, where we will learn object-oriented programming, let's take a moment to analyze a few more data types, keywords, built-in functions, and modules that are very useful when coding in Python. For the remaining ones, you will be referred to reliable webpages. Follow along with Notebook 34!

1. Data types

We extensively learned about strings, lists, integers and floats, Booleans, and dictionaries. Let's now have a look at the main characteristics of tuples and sets.

1.1 Tuples

We previously learned about tuples as the data type of the output of the built-in function `enumerate()` (Chapter 22), of the dictionary method `.item()` (Chapter 24), and of any function returning more than one variable (Chapter 29). As you might remember, tuples are a sequence of elements enclosed in between **round brackets** and separated by commas. They are **immutable**—that is, their elements cannot be replaced, added, or removed—thus, they are the ideal data type for **variables that do not change throughout the code**. In addition, tuples have only two methods, that is, **`.count()`**, which returns the **number of times an element is present in a tuple**, and **`.index()`**, which returns the **position of an element in a tuple**. Let's have a look at a simple example of a tuple and its two methods.

- Given the following tuple:

```
[1]:
1  image_size = (256, 256, 3)                image size is assigned 256 256 3
2  print(image_size)                          print image size
(256, 256, 3)
```

The tuple `image_size` contains the dimensions of an RGB image (line 1), where the first element `256` corresponds the number of rows, the second element `256` to the number of columns, and `3` to the color channels—that is, red, green, and blue, as you might remember from the *In more depth* section *Digital images* in Chapter 23. In the following line, we print the tuple to check for correctness (line 2).

- Calculate how many times `256` is present:

```
[2]:
1  print(image_size.count(256))               print image size dot count 256
2
```

We use the method `.count()` that takes the number whose presence we want to count—that is, `256`—as an argument and returns the number of times that number is present—that is, `2`—which we directly print (line 1).

- Compute the position of `3`:

```
[3]:
1  print(image_size.index(3))                 print image size dot index three
2
```

We use the method `.index()` that takes the number whose position we want to know—that is, `3`—as an input and returns its position—that is, `2`—which we directly print (line 1).

1.2 Sets

A **set** is a data type whose elements are enclosed in between **curly brackets** and **separated by commas**. Sets have three main characteristics: (1) they are **immutable**—like tuples; (2) they are **unordered**, that is, their elements do not have a fixed position—thus, sets **cannot be sliced**; and (3) they contain **unique** elements, therefore no duplicates are allowed. Let's have a look at a set.

- Given the following set, print it:

```
[4]:
1  cities = {"Buenos Aires", "Prague", "Delhi", "Delhi"}
2  print(cities)
3  print("The number of elements is:", len(cities))
```

```
{'Delhi', 'Buenos Aires', 'Prague'}
The number of elements is: 3
```

```
cities is assigned Buenos Aires, Prague, Delhi, Delhi
print cities
print The number of elements is: len cities
```

We create a set called `cities` containing 4 elements that are strings (line 1) and we print it (line 2). From the print, we can see that the resulting set is different from what we defined at line 1. Because sets are unordered—characteristic (2) above—the elements are internally organized differently than in their definition, and they might be printed in a different order every time. Second, because sets contain only unique elements—characteristic (3) above—`"Delhi"` is present only once. Thus, `cities` actually contains only 3 elements, as we can see when printing its length (line 3).

Given their characteristics, sets are very convenient **intermediates for list operations**. We can use set properties to remove duplicates from a list and two set methods—`.union()` and `.intersection()`—to respectively merge two lists and find their common elements—sets have 15 more methods, which you can easily explore on the internet[1]. Let's look at some examples.

- Given the following list:

```
[5]:
1  cities = ["San Francisco", "Melbourne",
   "San Francisco", "Milan"]
```

```
cities is assigned San Francisco, Melbourne, San Francisco, Milan
```

We start with the list `cities` containing four strings, two of which are the same—`"San Francisco"` is both in positions `0` and `2`.

- Remove the duplicates:

```
[6]:
1  cities = list(set(cities))
2  print(cities)
```

```
['Milan', 'Melbourne', 'San Francisco']
```

```
cities is assigned list set cities
print cities
```

We use the built-in function **`set()`** to **transform the list into a set**, and thus automatically **remove duplicate elements**. Then, we use the built-in function **`list()`** to transform the obtained set **back to a list** (line 1). From the print, we see that `"San Francisco"` is now present only once. Note that the order of the list elements is now different from the original list (cell 5) because sets are unordered. Thus, this trick is useful only when the element order is not important! Let's look into the next examples.

[1] Example: `www.w3schools.com/python/python_ref_set.asp`

- Given the following lists:

```
[7]: 1 cities_1 = ["Santiago", "Bangkok",
     "Cairo", "Santiago"]
     2 cities_2 = ["Cairo", "Cape Town"]
```

```
cities one is assigned Santiago, Bangkok,
Cairo, Santiago
cities two is assigned Cairo, Cape Town
```

We start with two lists called `cities_1` and `cities_2` containing 4 and 2 strings respectively.

- Create a new list that contains unique elements from both lists:

```
[8]: 1 all_cities = list(set(cities_1).union(
     set(cities_2)))
     2 print(all_cities)
['Bangkok', 'Cape Town', 'Santiago', 'Cairo']
```

```
all cities is assigned list set cities
one dot union set cities two
print all cities
```

Let's understand the nested code at line 1 with the help of Figure 34.1. We transform the two lists `cities_1` and `cities_2`—represented by the orange and yellow rectangles at the top of the figure—to the corresponding sets—orange and yellow ellipses in the middle of the figure—using the built-in function `set()`. Then, we apply the method **`.union()`** to `set(cities_1)` with `set(cities_2)` as an argument to **merge the elements that are present in both sets**—we could have also applied `.union()` to `set(cities_2)` with `set(cities_1)` as an argument. To the output set, we directly apply the built-in function `list()` and obtain `all_cities`—green rectangle at the bottom of the figure—that is, a list containing the four unique elements of the two initial lists. At line 2 of the code, we print `all_cities` to test for correctness.

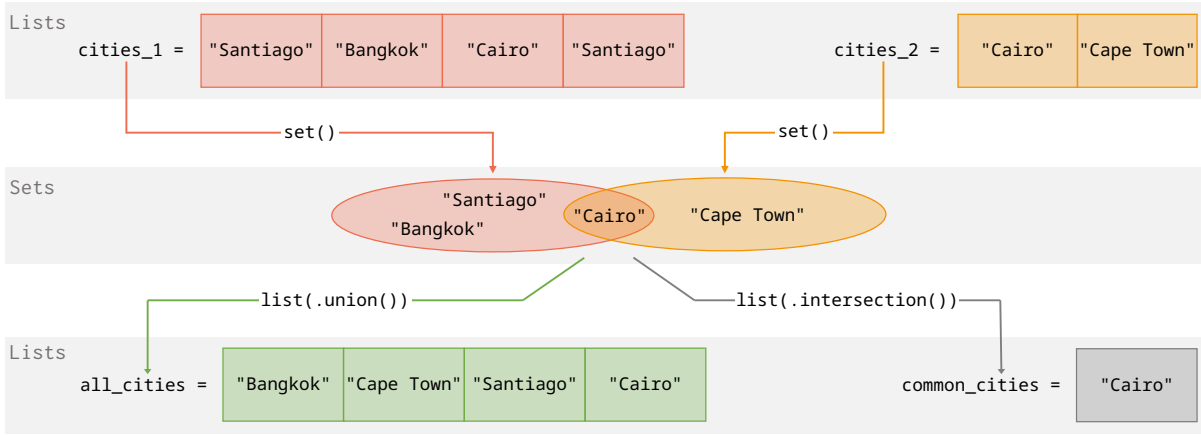


Figure 34.1. Using the sets methods `.union()` and `.intersection()` to merge two lists or find their common elements.

- Create a new list that contains the elements common to both lists:

```
[9]: 1 common_cities = list(set(cities_1).
     intersection(set(cities_2)))
     2 print(common_cities)
['Cairo']
```

```
common cities is assigned list set cities
one dot intersection set cities two
print common cities
```

We perform the same steps as above but we use the set method **`.intersection()`** to **extract the common elements in the sets**. We obtain the list `common_cities`—gray rectangle in Figure 34.1—containing only the common element `"Cairo"`. Finally, we print to check for correctness (line 2).

2. Keywords

You already know many Python keywords—including `for`, `del`, `def`, `if`, etc. You can find the complete list of keywords on several webpages[2]. In this section, we will focus on `lambda`.

2.1 lambda

The keyword `lambda` is used to create **one line functions containing one single operation**. These compact functions are called **anonymous functions**—because they do not have a name!—or **lambda functions**, because they are created using the keyword `lambda`. To understand how they work, let's compare a simple regular function that calculates the double of a number with its corresponding lambda function.

- Here is the regular function:

```
[10]:
1  def double_number(number):
2      """Returns the double of a number
3
4      Parameters
5      ----------
6      number : float
7          The input number
8
9      Returns
10     -------
11     float
12         The double of the input number
13     """
14
15     return number * 2
16
17 print(double_number(5))
10
```

```
def double number number
Returns the double of a number

Parameters

number : float
The input number

Returns

float
The double of the input number


return number times two

print double number five
```

In the function `double_number`, the input is a float (lines 1 and 4–7) and the output is the calculated double (lines 9-12 and 15). We call the function for the number 5 and directly print the output (line 17).

- Here is the corresponding lambda function:

```
[11]:
1  double_number = lambda number: number * 2

2  print(double_number(5))
10
```

```
double number is assigned lambda number
number times two
print double number five
```

The typical syntax of a lambda function is represented at line 1 after the assignment operator. It includes: (1) the keyword **`lambda`**; (2) the function **input**—that is, `number`; (3) **colon**; and (4) the **operation that computes the output**—that is, `number * 2`. It is common to **assign a lambda function to a variable**—in our case, `double_number`—to call it. To better understand the syntax of a lambda function, let's compare it with the syntax of the corresponding regular function with the help of Figure 34.2. The input `number` (orange rectangle) is in between round brackets in the header of a regular function (line

[2] Example: `www.w3schools.com/python/python_ref_keywords.asp`

(a)), whereas it is positioned right after the keyword `lambda` in a lambda function (line (c)). The operation that computes the output `number * 2` (yellow rectangle) is after the keyword `return` in a regular function (line (b)), whereas it is after the colon in a lambda function (line (c)). Finally, the regular function has a name—e.g., `double_number` (gray rectangle at line (a))—whereas a lambda function is often **assigned to a variable** (gray rectangle at line (c)).

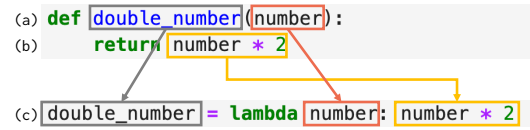


Figure 34.2. Regular function (a–b) and corresponding anonymous function (c).

To **call a lambda function**, we write the **name of the function variable** followed by the **input in round brackets** (cell 11, line 2). For convenience, we directly print the output to check for correctness. Lambda functions are particularly convenient when we want to apply simple operations to list elements. To do so, we use the built-in function `map()`, as you will learn in the coming section.

3. Built-in functions

In the past chapters, you learned several Python built-in functions, including `print()`, `input()`, `len()`, `sum()`, etc. Python has a total of 71 built-in functions, which you can discover in the official Python documentation[3]. In this chapter, we briefly explore `map()`.

3.1 map()

The built-in function `map()` is commonly used to **apply a lambda function to each element of a list**. It somehow acts like a `for` loop that extracts one list element at the time and applies a wanted function to it. Let's see how `map()` works with the following example.

- Double each list element using a lambda function:

```
[12]: 1 numbers = [3, 5, 7]
      2 doubles = list(map(double_number, numbers))
      3 print(doubles)
      [6, 10, 14]
```

```
numbers is assigned three, five, seven
doubles is assigned list map double
number numbers
print doubles
```

We create a list containing three integers and we name it `numbers` (line 1). Then, we use `map()` with two inputs: the lambda function `double_number`—from cell 11—and the list we want to apply the lambda function to, that is, `numbers` from line 1—note that it's also common to write the lambda function directly into the `map()` command: `map(lambda number: number * 2, numbers)`. Because `map()` returns its own type, we transform the output to a list using the built-in function `list()` (line 2). Finally, we print the output to check for correctness (line 3).

[3] `docs.python.org/3/library/functions.html`

4. Modules

Python provides numerous built-in modules that can be found on the official website[4]. In this section, we will learn one more function of the module `random` and introduce the module `time`.

4.1 random

We already know two functions of the module `random`, that is, `.randint()` to generate random numbers within a range and `.choice()` to randomly pick an element from a list. When using these functions, it can be hard to **debug code** or **verify the correctness of results** because the generated random output is **different at each execution**. For example, when we generate a random number twice, we obtain two different results, as we can see in the example below.

- Generate a random integer between 1 and 10 twice:

[13]:
```
1 import random
2
3 n = random.randint(1, 10)
4 print("n:", n)
5 n = random.randint(1, 10)
6 print("n:", n)
```
```
n: 7
n: 9
```
```
import random

n is assigned random dot randint one ten
print n n
n is assigned random dot randint one ten
print n n
```

We begin by importing the module random (line 1). We use the command `.randint()` to generate a random integer between `1` and `10` (line 3), and we print it (line 4). The generated number is `7`. Then, we rewrite the same commands as above (lines 3–4) to generate another random integer between `1` and `10` (lines 5–6). This time, we obtain `9`. As expected, the two generated numbers are different. But how are these numbers created? When we call `.randint()`, we execute a process similar to the one shown in Figure 34.3. The core is an **algorithm** consisting of a set of complicated but **deterministic**—that is, well defined—operations that transform an input number into an output number. The input number is called **seed number** and is usually determined from the combination of the current time and date on our computer—and thus it is unique because time always progresses. The output number is a **pseudorandom number**—not completely random!—because it looks random but is actually generated by a deterministic process.

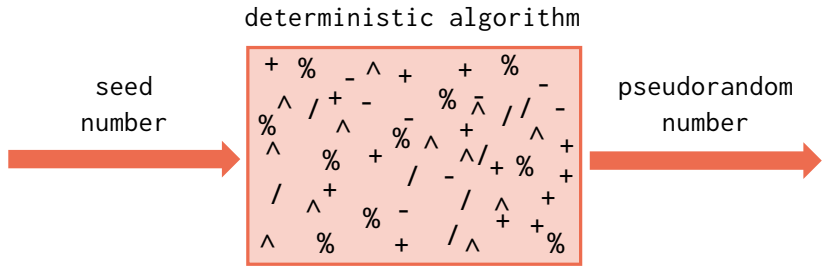


Figure 34.3. Scheme of a pseudorandom number generator.

Can we generate the **same random number**—or better, pseudorandom number—every time we run the code? Yes, when we specify the **seed number**! Let's see how it works in the following example.

[4] docs.python.org/3/library/

- Generate a random integer between 1 and 10 twice, using a seed number:

[14]:
```
1 random.seed(18)
2 n = random.randint(1, 10)
3 print("n:", n)
4
5 random.seed(18)
6 n = random.randint(1, 10)
7 print("n:", n)
```
```
n: 3
n: 3
```
```
random dot seed 18
n is assigned random dot randint one ten
print n n

random dot seed 18
n is assigned random dot randint one ten
print n n
```

From the module random, we call the function **`.seed()`**, which **takes a seed number as an input** (line 1)—in this example, `18`. We generate a random number between `1` and `10` using the same commands as above—that is, lines 2–3 are the same as lines 3–4 from cell 13. We obtain the number `3`. Then, we reuse the same commands to provide the same seed number and generate a new random number (lines 5–7). As expected, we obtain `3` again. Thus, when using a seed number, the generation of a random number is **reproducible** and we can debug code and verify results much easier. Finally, what number should we pick as a seed number? Any number we want! Our choice of a seed number is the only real random factor when generating random numbers!

4.2 time

Knowing the **computational time**—that is, how long a computation takes—is very important, especially when running large projects. In Python, we can use the module `time`. Let's see how it works with the following example.

- Compare the time it takes to create a list with ten, a hundred, and a thousand zeros, using a `for` loop vs. list replication. What do you think the time difference will be?

[15]:
```
1  import time
2
3  # list lengths
4  n_of_elements = [10, 100, 1000]
5
6  # for each length
7  for n in n_of_elements:
8
9      print("N. of zeros:", n)
10
11     # create the list using the for loop
12     start = time.time()
13     numbers = []
14     for _ in range(n):
15         numbers.append(0)
16     end = time.time()
17     print("For loop: {:.6f} sec".format(end -
       start))
18
```
```
import time

list lengths
n of elements is assigned 10, 100,
1000

for each length
for n in n of elements

print N. of zeros: n

create the list using the for loop
start is assigned time dot time
numbers is assigned empty list
for underscore in range n
numbers dot append zero
end is assigned time dot time
print For loop: column dot six f
sec dot format end minus start
```

```
19      # create the list using list replication
20      start = time.time()
21      numbers = [0] * n
22      end = time.time()
23      print("List repl: {:.6f} sec".format(end -
        start))
```

```
create the list using list
replication
start is assigned time dot time
numbers is assigned zero
replicated by n
end is assigned time dot time
print List repl: column dot six f
sec dot format end minus start
```

```
N. of zeros: 10
For loop: 0.004142 sec
List repl: 0.000001 sec

N. of zeros: 100
For loop: 0.000024 sec
List repl: 0.000002 sec

N. of zeros: 1000
For loop: 0.000147 sec
List repl: 0.000004 sec
```

Let's first analyze the code to create the lists. We begin with the list `n_of_elements`, which contains the lengths of the lists that we are going to create (line 4). Then, we write a `for` loop that spans the list lengths (line 7). For each length, first, we print the number of zeros that will be contained in the list (line 9). Then, we create a list using a `for` loop (lines 11–17), that is, we initialize the empty list (line 13), create a `for` loop through indices that goes from `0` (omitted) to the current length (line 14), and we append `0` to the list `numbers` at each iteration (line 15). Finally, we create a list using list replication (line 19–23), where we replicate the list `[0]` by the number of zeros `n` (line 21). How do we calculate the time of these computations? First, we import the module `time` (line 1). Then, before each list creation, we call the function **`.time()`** from the module `time` and we assign the output to the variable `start` (lines 12 and 20). This command corresponds to **pressing start on a stopwatch**. After each list creation, we call again the function `.time()` from `time`, and we assign it to the variable `stop` (lines 16 and 22)—like we pressed **stop on a stopwatch**. Finally, we print the **difference between `stop` and `start`** to know the computational time in **seconds** (lines 17 and 23). To obtain a clear print, we use the string method `.format()` with six digits. As you can see, creating a list with a `for` loop is much slower than with list replication, especially for long lists! Keep in mind that the computation time also depends on the number of processes currently running on the computer—for example, emails or opened documents—as well as on the computer's own characteristics. For this reason, it can vary at different times or across different machines.

5. Swapping variables

Let's conclude with a nice Python trick: the ability to swap variables in a single line—a command not available in many programming languages! Let's see how it works in the example below:

```
[16]:
1  v_1 = "a"                                  v one is assigned a
2  v_2 = "b"                                  v two is assigned b
3  print("v_1:", v_1)                         print v one v one
4  print("v_2:", v_2, "\n------")             print v two v two backslash n dashes
5
6  v_1, v_2 = v_2, v_1                        v one v two is assigned v two v one
7
8  print("v_1:", v_1)                         print v one v one
9  print("v_2:", v_2)                         print v two v two

v_1: a
v_2: b
------
v_1: b
v_2: a
```

We create two variables, that is, `v_1` to which we assign the string `"a"` (line 1) and `v_2` to which we assign the string `"b"` (line 2). We print them for a check (lines 3 and 4). Then, we swap the variables by writing: (1) the two variables separated by a comma, (2) the assignment operator, and (3) the two variables separated by a comma in inverted order (line 6). When we reprint the variables (lines 8 and 9), we can see that the values are swapped because `v_1` now contains `"b"` and `v_2` now contains `"a"`!

Recap

- Tuples are a data type containing immutable elements and have two methods: `.count()` and `.index()`.
- Sets are a data type whose elements are immutable, unordered, and unique. They have 17 methods, including `.union()` and `.intersection()`. Sets can be useful intermediators for list operations such as removing duplicates, merging two lists, or finding the common elements in two lists.
- The keyword `lambda` allows the creation of anonymous functions, which are compact functions composed of (1) `lambda`, (2) input, (3) colon, and (4) operation generating the output.
- The built-in function `map()` is useful when applying a lambda function to each element of a list.
- The function `.seed()` from the module `random` allows us to create reproducible random—or more technically, pseudorandom—numbers.
- To calculate computational time, we use the module `time`. Its function `.time()` acts like the start or stop of a stopwatch.

What is pip install?

Have you ever used the command **`pip install`**? It is a **terminal command to install Python packages**. A **package** is simply a **collection of modules**—that is, `.py` files containing functions for a specific task, as you might recall from Chapter 32. Some of the most popular packages include NumPy for numerical computing, Pandas for data manipulation and analysis, Matplotlib and Seaborn for data visualization, Scikit-learn for machine learning and Tensorflow and Pytorch for deep learning. To share a package with others, developers upload it to PyPI (Python Package Index; `www.pypi.org`), a repository hosting hundreds of thousands of Python packages. When we want to use a package in our code (e.g. Pandas), we typically follow 2 simple steps:

1. Install the package. To do that, we open the terminal and we write `pip install` and the name of the package—e.g., `pip install pandas`. This command downloads the package from PyPi and installs it on the computer.
2. Import the package in our Python code. We import packages exactly like we import modules, that is by writing the command `import` followed by the name of the package—e.g., `import pandas`.

How to create a package is beyond the scope of this book; however, you can find a detailed guide in the official documentation at `https://packaging.python.org`.

Let's code!

1. *How long does it take to square numbers?* Create a regular function that given a number returns its square. Then, write the corresponding lambda function. Finally, apply both functions to a list of 10 numbers. How long do all these operations take?
2. *How long does it take to concatenate lists?* Create two lists, one containing integers from 0 to 10000, and another containing integers from 10001 to 20000. *Hint*: use `range()`. Then, merge the two lists once by adding one element at a time from the second list to the first, and once by using list concatenation. Which method is faster? By how much?

PART 10

OBJECT-ORIENTED PROGRAMMING

In this last part, you will learn how to code using classes and objects, with their attributes and methods. Ready to expand your horizon?

35. Let's build an online store!

Classes and objects, attributes and methods

Up to this point, you've learned Python's grammar, including data types, operators, loops, and functions. You've also learned how to code following the principles of **procedural programming**, that is, creating **step-by-step instructions** that are **executed sequentially** to get to the solution of a task. In this last part of the book, we will switch gears and learn the basics of **object-oriented programming**—commonly abbreviated as OOP. It is a different way of thinking about code, based on the **representation of the world in classes and objects, with their attributes and methods**. Let's get familiar with these new concepts while solving the following task. Follow along with Notebook 35!

- You are building a new online store where you will sell various types of products. Each product will have a similar webpage containing characteristics such as name, original price, and discount. In addition, there will also be two possible actions: apply coupon (to add the value of a coupon to the discount) and calculate price (to calculate an item price).
  1. Create a template for the webpage and then customize it for a T-shirt and a lamp with the following characteristics:
     - T-shirt: name: `Feel good`, original price: `30` coins, launch discount: `4` coins;
     - Lamp: name: `Lux`, original price: `40` coins, no launch discount.
  2. Then, execute these actions for each product:
     - Calculate the price after the discount;
     - Calculate the price after adding the coupon `SAVE4` worth `4` coins to the T-shirt and the coupon `SUMMER10` worth `10` coins to the lamp.

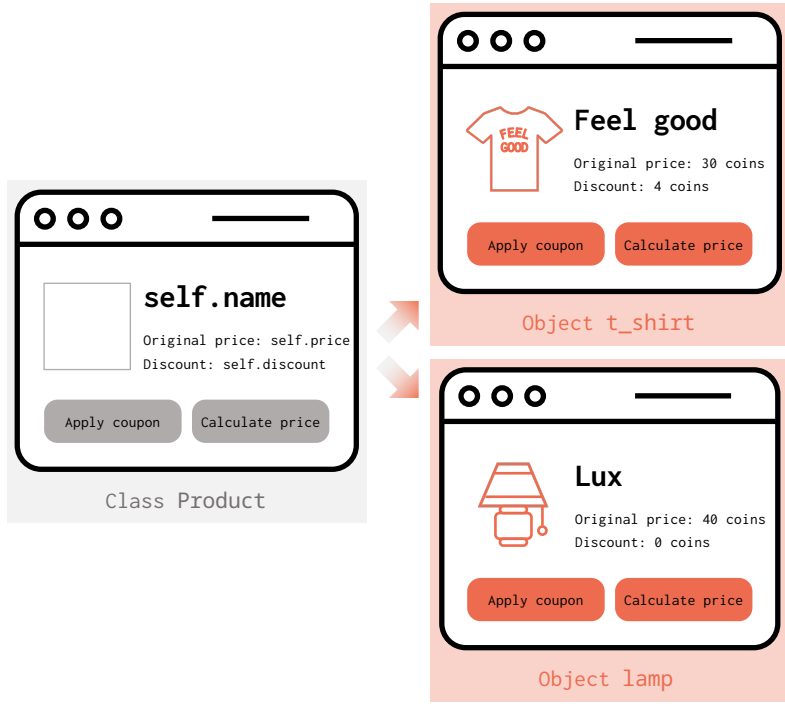


Figure 35.1. Representation of the class `Product` (left) and its objects `t_shirt` and `lamp` (right).

How do we solve this task? We will create a **class** (in grey in Figure 35.1) for the webpage template and two **objects**, one for the T-shirt, and one for the lamp (in orange). For the product **characteristics**—name, price, and discount—we will define **attributes**, whereas for the **actions**—apply coupon and calculate price—we will implement **methods**. For now, just focus on understanding the syntax of the code and its functionality. By the end of the chapter, concepts and definitions will become clear. In the next pages, the code will be presented without pronunciation guides, and functions will have reduced documentation because of the large number of commands. However, it is always recommended to read code carefully and write complete function documentation for better understanding and reuse. Enough talk, let's jump into the code!

1. Class, objects, and attributes

Let's start from the first point of the task. Here we have a class representing the webpage template and an object representing the T-shirt—we will create the object representing the lamp at the end of this first section. Read the code below and try to understand its syntax and functionality.

- Here is the class representing a product and its attributes:

[1]:
```
class Product:
    """Class representing a product"""

    # --- CONSTRUCTOR -------------------
    def __init__(self, name):
        """Constructor"""
        self.name     = name
        self.price    = 0
        self.discount = 0
```

- Here is the object `t_shirt` with its attributes at instantiation and after customization:

[2]:
```
t_shirt = Product("Feel good")

print("-> Attributes of the object t_shirt at instantiation:")
print("Name:", t_shirt.name)
print("Price:", t_shirt.price, "coins")
print("Discount:", t_shirt.discount, "coins")

print("-> Attributes of the object t_shirt after customization:")
t_shirt.price = 30
print("Price:", t_shirt.price, "coins")
t_shirt.discount = 4
print("Discount:", t_shirt.discount, "coins")
```

(a)
```
-> Attributes of the object t_shirt at instantiation:
```
(b)
```
Name: Feel good
```
(c)
```
Price: 0 coins
```
(d)
```
Discount: 0 coins
```
(e)
```
-> Attributes of the object t_shirt after customization:
```
(f)
```
Price: 30 coins
```
(g)
```
Discount: 4 coins
```

What happens in the code above? Get some hints by solving the following exercise.

✎ True or false?

1. To create a class we use the keyword class. T F
2. Product is an object and t_shirt is a class. T F
3. A class contains a constructor. T F
4. We use the variable self both in classes and objects. T F
5. An attribute is a variable whose value can change. T F

Computational thinking and syntax

In cell 1, we create a **class** by writing: (1) the keyword **`class`**, (2) the **class name**, and (3) **colon** (line 1). Unlike variable names, class names are **capitalized**, and if they consist of multiple words, each word begins with a capital letter—such as in `CustomerOrder`, for example. In our code, the class name is `Product`. All the code that follows this first line is **indented**. We write a **one-line documentation** describing the purpose of the class (line 2). Then, there is the **constructor** (lines 4–9), which is a method—that is, a class function, as we will see in the next section—that is always present in a class and is used to create an object, as we will learn in a bit. The constructor has the predefined name **`__init__()`** and one or more parameters in between round brackets (line 5). The **first parameter** is a variable commonly called **`self`**. For now, just consider `self` as part of the syntax. We will clarify its meaning in the *In more depth* section in the next chapter. The second parameter is `name`. Below the header, there is a one-line documentation specifying that this is the constructor (line 6). In the body, there are three variables, which we call **attributes** (lines 7–9). Each is composed of: (1) **`self`**, (2) **dot**, (3) **attribute name**, (4) **assignment operator**, and (5) **a variable or an initializing value**. In our example, the first attribute is `self.name` (line 7), and it is assigned the variable `name`—that is, the second parameter of the constructor (line 5). The second attribute is `self.price` and is initialized with `0` (line 8). Similarly, the third attribute is `self.discount` and is initialized with `0` (line 9). Note that **an input variable and the corresponding attribute usually have the same name**—such as `name` and `self.name`—to improve readability and indicate their connection. Also, it is common to align the assignment operators vertically for a clearer readability of the attributes. Finally, all the attributes are in blue in Jupyter Notebook. When we run a cell containing a class, the class is saved in memory but no code is executed—similarly to when we run a cell containing a function without its call. To execute the code of a class, we need an object.

To **instantiate**—that is, create—an object, we write: (1) **object name**, (2) **assignment operator**, (3) **class name**, and (4) **arguments**—excluding `self`!—in between round brackets. In our example, we instantiate `t_shirt` as an object of the class `Product` with the argument `"Feel good"` (cell 2, line 1). What happens behind the scenes when we instantiate an object? We **call the constructor** of the class! In our case, we call `__init__()` in `Product` and we pass the argument `"Feel good"` to the variable `name` (orange arrow in Figure 35.2). There, the value in `name`—that is, `"Feel good"`—is assigned to the attribute `self.name` (line 7 in cell 1, and grey arrow in Figure 35.2, left). At this point, the object `t_shirt` is created and it **has all the attributes specified in the class** (black dashed arrows in Figure 35.2). The syntax for an **attribute** in an **object** is: (1) **object name**, (2) **dot**, and (3) **attribute name**. Thus, the attribute `self.name` in the class `Product` is `t_shirt.name` in the object `t_shirt`. Similarly, the attribute `self.price` in `Product` is `t_shirt.price` in `t_shirt`, and `self.discount` is `t_shirt.discount`.

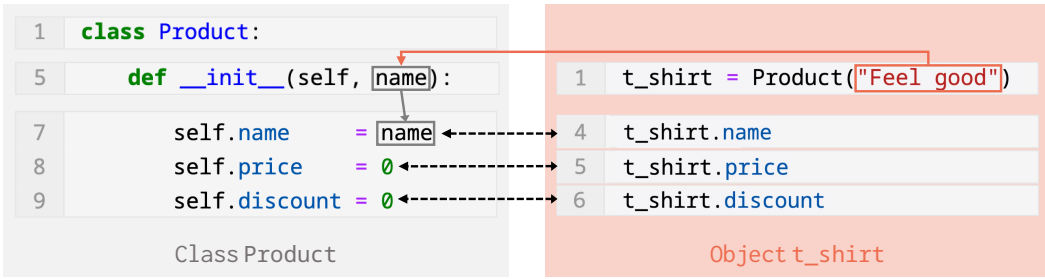


Figure 35.2. Class `Product` (left) and its object `t_shirt` (right). At instantiation, `"Feel good"` is passed to the parameter `name` and `t_shirt` gets the attributes of `Product`.

What are the **initial values** of an object's attributes? **At instantiation**, the object's attributes have the **values defined in the constructor**—remember, the class is a template! Thus, when we print `t_shirt.name`, we obtain `Feel good` (cell 2, line 4; print (b)), that is, the value in `name`, as defined in the class (cell 1, line 7). When we print `t_shirt.price`, we obtain `0` (cell 2, line 5; print (c)), as defined in the class (cell 1, line 8). And similarly, when we print `t_shirt.discount`, we obtain `0` (cell 2, line 6; print (d)), as defined in the class (cell 1, line 9). How can we **customize the values of an object's attribute**? Simply, by **assignment**. Thus, to set the original price of the T-shirt to 30 coins, we write `t_shirt.price = 30` (cell 2, line 9), and from the print we see that the attribute `t_shirt.price` now has value `30` (line 10; print (f)). Similarly, to customize the launch discount, we assign the value `4` to `t_shirt.discount` (line 11), and when we print it, we see that the value is now `4` coins (line 12; print (g)).

Let's create the object `lamp`, as requested by the first point of the task. How would you do it? Instantiate `lamp` and play with its attributes before looking at the solution below.

- Here is the object `lamp` with its attributes:

```
[3]:
1 lamp = Product("Lux")
2 lamp.price = 40
3 print("Name:", lamp.name)
4 print("Price:", lamp.price, "coins")
5 print("Discount:", lamp.discount, "coins")
```

```
(a) Name: Lux
(b) Price: 40 coins
(c) Discount: 0 coins
```

We instantiate `lamp` as an object of the class `Product` with the string `"Lux"` as the argument (line 1). Then, we change the lamp price from the default `0`—in the class, `self.price` is initialized to `0` (cell 1, line 8)—to `40` with an assignment (cell 3, line 2). Because the lamp has no discount, `lamp.discount` has the default value `0` (cell 1, line 9), and thus we do not need to write any assignment command. When we print the attributes, we see that `lamp.name` is now `"Lux"` (cell3, line 3; print (a)), `lamp.price` is `40` (line 4; print (b)), and `lamp.discount` is `0` (line 5; print (c)).

2. Methods

Let's tackle the second point of the task! We need to implement two actions, that is, applying a coupon and calculating an item price. To do so, we copy the class `Product` and we add two methods called `apply_coupon()` and `calculate_price()` (cell 4, lines 12–27)—in the code below you will find also the

built-in method `__str__()` (lines 31–33) that we will discuss at the end of the chapter. Because the class is changed, we need to re-instantiate the object `t_shirt` to add the new functionalities (cell 5). Read the following code and try to understand what it does!

- Here is the complete class with its attributes and methods:

[4]:
```
1  class Product:
2      """Class representing a product"""
3
4      # --- CONSTRUCTOR -------------------
5      def __init__(self, name):
6          """Constructor"""
7          self.name     = name
8          self.price    = 0
9          self.discount = 0
10
11
12     # --- METHODS -----------------------
13     def apply_coupon(self, coupon):              # added
14         """Updates discount based on a coupon"""
15         if coupon == "SAVE4":
16             self.discount = self.discount + 4
17             print("Coupon SAVE4 applied!")
18         elif coupon == "SUMMER10":
19             self.discount = self.discount + 10
20             print("Coupon SUMMER10 applied!")
21         else:
22             print("Your coupon is not valid")
23
24     def calculate_price(self):                   # added
25         """Calculates price after discount"""
26         updated_price = self.price - self.discount
27         return updated_price
28
29
30     # --- BUILT-IN METHOD --------------
31     def __str__(self):                           # added
32         """Prints the object characteristics"""
33         return "Name: " + self.name
```

- Instantiate the object `t_shirt` with its attributes. Then, calculate the price after discount, and the price after discount and coupon:

[5]:
```
1  # instantiating the object with its attributes
2  t_shirt = Product("Feel good")
3  t_shirt.price = 30
4  t_shirt.discount = 4
5  print("Name:", t_shirt.name, "| original price:", t_shirt.price,
   "coins | launch discount:", t_shirt.discount, "coins")
6                                                                    ↳
```

```
7  # calculating the price after launch discount
8  print("-> Price after launch discount")
9  t_shirt_price = t_shirt.calculate_price()
10 print("Price:", t_shirt_price, "coins")
11
12 # applying the coupon
13 print("-> Applying the coupon")
14 t_shirt.apply_coupon("SAVE4")
15 print("Discount:", t_shirt.discount, "coins")
16
17 # calculating the price after launch discount and coupon
18 print("-> Price after launch discount and coupon")
19 t_shirt_price = t_shirt.calculate_price()
20 print("Price:", t_shirt_price, "coins")
```

```
(a) Name: Feel good | original price: 30 coins | launch discount: 4 coins
(b) -> Price after launch discount
(c) Price: 26 coins
(d) -> Applying the coupon
(e) Coupon SAVE4 applied!
(f) Discount: 8 coins
(g) -> Price after launch discount and coupon
(h) Price: 22 coins
```

What are methods and how do they work? Get some hints by solving the following exercise!

✎ True or false?

1. Methods are functions in a class. T F
2. An object cannot call a class method. T F
3. The method apply_coupon() modifies the attribute t_shirt.discount. T F
4. The method calculate_price() adds the discount to the original price. T F

Computational thinking and syntax

Methods are simply **functions in a class**. Their **first parameter** is usually **`self`**, as we can see in the headers of `apply_coupon()` (cell 4, line 13) and `calculate_price()` (line 24). In case of additional parameters, they are placed after `self`—such as `coupon`, which is in second position in the `apply_coupon()` header (line 13). Because `self` is a parameter, **attributes can be accessed and modified directly within methods**. We will explain why in the *In more depth* section in the next chapter. Thus, in `apply_coupon()` we can modify `self.discount` without passing it as a parameter (lines 16 and 19). Similarly, in `calculate_price()` we can directly use `self.price` and `self.discount` (line 26). What do `apply_coupon()` and `calculate_price()` do? The method `apply_coupon()` contains an `if/elif/else` construct specifying the update of `self.discount` based on a coupon. If `coupon` is `"SAVE4"` (line 15), we add `4` coins to `self.discount` (line 16) and we print a message accordingly (line 17); otherwise if `coupon` is `"SUMMER10"` (line 18), we add `10` coins to `self.discount` (line 19) and we print a message accordingly (line 20); otherwise (line 21), we print that the coupon is not valid (line 22). The method `calculate_price()` calculates and returns the updated price as the difference between the value of `self.price` and the value of `self.discount` (lines 26–27). In general, methods can be written **in any**

order in the class and they can be **called in any order and multiple times by an object**. Let's see how.

The syntax to **call a method from an object** is: (1) **object name**, (2) **dot**, (3) **method name**, and (4) **arguments**—if any—in between round brackets, **excluding `self`**. The method's **output** can be assigned to a **variable**, of course. In our example, first, we re-instantiate the object customizing its attributes, and we print them for a check (cell 5, lines 2–5; print (a)). Then, we calculate the price after launch discount by writing `t_shirt.calculate_price()` (line 9), and we assign the output to `t_shirt_price`, which we print to obtain `26` (line 10; print (c)). What happens exactly when we call the method? As you can see in Figure 35.3, `t_shirt` calls `calculate_price()` in the class (orange arrow), which is **executed using the values of the object's attributes**. Thus, `self.price - self.discount` is `30 - 4` (line 26). The resulting value `26` is assigned to `updated_price`, which is returned (line 27) and passed to the variable `t_shirt_price` (gray arrow). Finally, the price is printed for a check (line 10; print (c)).

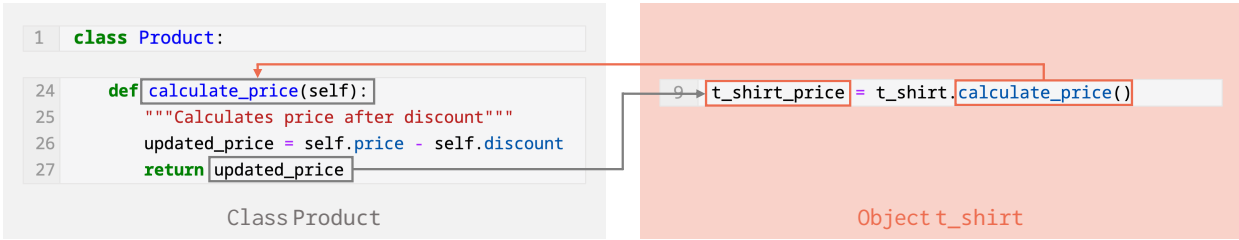


Figure 35.3. Method `calculate_price()` defined in the class `Product` (left) and called by the object `t_shirt` (right).

Let's apply the coupon and calculate the final price. The object `t_shirt` calls the method `apply_coupon()` with the argument `"SAVE4"` (line 14). Similarly to above, `apply_coupon()` is executed using the values of the object attributes. Because the `if` condition in the method is `True` (cell 4, line 15), we execute `self.discount + 4`—that is, `4 + 4`—and we update `self.discount` with the new value `8` (line 16). Then, we print the message that the coupon has been applied (line 17; print (e)). To check for correctness, we print the object's attribute `t_shirt.discount`, and we see the updated value `8` (cell 5, line 15; print (f)). Finally, we calculate the T-shirt price with the updated discount that includes launch discount and coupon. To do so, we simply recall the method `calculate_price()` (line 19). Within the method, we subtract `8`—value in `self.discount`— from `30`—value in `self.price`— to obtain `22` (cell 4, line 26), which is returned (line 27) to the variable `t_shirt_price` (cell 5, line 19) and eventually printed (line 20; print (h)).

How would you calculate the price of the object `lamp` before and after applying the coupon `"SUMMER10"`? Give it a try before looking into the solution below.

- Instantiate the object `lamp` with its attributes and calculate its price before and after applying the coupon `SUMMER10`:

[6]:
```
# instantiating the product lamp with its attributes
lamp = Product("Lux")
lamp.price = 40

# calculating the original price
print("-> Original price")
lamp_price = lamp.calculate_price()
print("Price:", lamp_price, "coins")
```

```
9
10 # calculating the price after coupon
11 print("-> Price after coupon")
12 lamp.apply_coupon("SUMMER10")
13 lamp_price = lamp.calculate_price()
14 print("Price:", lamp_price, "coins")
```

```
(a) -> Original price
(b) Price: 40 coins
(c) -> Price after coupon
(d) Coupon SUMMER10 applied!
(e) Price: 30 coins
```

Similarly to cell 3, we instantiate `lamp` as an object of the class `Product` with the argument `"Lux"` (line 2), and we customize its price to `40` coins (line 3). Because the lamp has no launch discount, we do not need to modify its attribute `lamp.discount`, which has the default value of `0` coins. We calculate the original price by calling `lamp.calculate_price()` (line 7). In the method, `self.price - self.discount` is `40 - 0` (cell 5, line 26), that is `40`, which is returned (line 27) to the variable `lamp_price` (cell 6, line 7) and printed (line 8; print (b)). Then, we apply the coupon `"SUMMER10"` using the method `apply_coupon()` (line 12). This time, the `True` condition is `elif coupon == "SUMMER10"` (cell 4, line 18). Thus the discount is updated from the original value `0` to the new value `10` (line 19). Moreover, the feedback message stating that the coupon was applied is printed (line 20; print (d)). Finally, we recalculate the price using the method `calculate_price()` (cell 6, line 13). In the corresponding method, `self.price - self.discount` is `40 - 10` (cell 4, line 26), that is `30`, which is returned (line 27) to the variable `lamp_price` (cell 6, line 13), and eventually printed (line 14; print (e)).

As we mentioned at the beginning of this section, classes can also have **built-in methods**—that is, **predefined methods**—whose names start and end with double underscore. Also for built-in methods, the first argument is commonly `self` (cell 4, lines 5 and 31). The most common built-in method is the **constructor** `__init__()` (cells 1 and 4, line 5), which is mandatory in a class to allow for object instantiation. Another commonly used built-in method is **`__str__()`**, which defines **the print of an object**. In our example (cell 4, lines 31–33), we return (not print!) the string `"Name: "` concatenated with the attribute `self.name`. Let's see what happens when we print the two objects we created in this chapter.

- Print the two objects:

```
[7]: 1 print(t_shirt)
     2 print(lamp)
```

```
Name: Feel good
Name: Lux
```

When we print the object `t_shirt` (cell 7, line 1), we obtain: `Name: Feel good`. Similarly, when we print `lamp` (line 2), we obtain: `Name: Lux`. With the built-in method `__str__()` we somehow overwrote the built-in function `print()`. Without `__str__()`, we would get something like `<__main__.Product object at 0x102d8cbb0>` when executing `print(t_shirt)` or `print(lamp)`. In the following chapters, we will not focus further on built-in methods; however, you can find a complete list of built-in methods in the official Python documentation[1].

At this point, you might ask: When do I use object-oriented programming instead of procedural pro-

[1] https://docs.python.org/3/library/operator.html

gramming? Usually object-oriented programming is used in **projects that involve objects with similar characteristics and behaviors**. For example, on a social media platform, every person is an object with attributes such as name, posts, number of likes, and is able to perform actions such as follow somebody, like a post, and send a message. Similarly, in a school registry, each student has a name, an age, a curriculum, and can change a course, receive a grade, or participate in a school event. In a customer registry, each customer has a name, an email address, a phone number, and can send an email, schedule a meeting, generate an invoice. As you can see, a lot of software that we use in everyday life is based on object-oriented programming. You will also have the opportunity to explore more examples in the coding exercises at the end of these last few chapters.

Let's conclude this chapter with formal definitions of the concepts we learned:

A **class** is a template containing properties and actions
An **object** is an instance of a class
An **attribute** is a variable that represents a property of an object
A **method** is a function that represents an action that an object can perform

Before moving to the next chapters where we will expand the online store and discover new interesting properties of object-oriented programming, complete the exercise below to summarize and memorize the concepts we have discussed so far.

✎ Insert into the right column

Summarize the differences between classes and objects by completing the table with the following phrases:

`object_name.attribute_name`, present, capitalized with no spaces, `object_name.method_name()`, a template, absent, `self.attribute_name`, an instantiation of a class, lowercase separated by underscore, `def method_name()`

	Class	Object
What it is		
Name style		
`self`		
Attribute syntax		
Method syntax (creation in class / call in object)		

Recap

- In object-oriented programming, we represent the world in classes and objects with attributes and methods.
- A class is a template and is introduced by the keyword `class`.

- An object is an instance of a class.
- Attributes are variables representing an object's properties.
- Methods are functions in a class representing actions that an object can perform. Their first parameter is usually `self`.
- Classes can have built-in methods, such as the constructor `__init__()` and `__str__()`, which defines an object's print.

Python data types are classes!

Since the beginning of our journey, we have used **methods** for lists—e.g., `.append()`, `.remove()`—for strings—e.g., `.lower()`, `.upper()`— for dictionaries—e.g. `.keys()`, `.values()`— and for other data types, including tuples and sets. Does this ring any bell? In Python, **data types are classes** and the **variables** we create **are objects**! And we are familiar with many of their methods! Let's look into this simple code:

```
[1]:
1 greeting = "hello"
2 print(greeting.upper())
3 print(type(greeting))
```

```
HELLO
<class 'str'>
```

```
greeting is assigned hello
print greeting dot upper
print type greeting
```

At line 1, we create the variable `greeting`, which is an object of the class `str`. To be consistent with the syntax, we should write `greeting = str("hello")`, but Python provides us a shortcut—that is, the quotes—to simplify the instantiation. Then, we call the method `.upper()` to change its content to uppercase and we print it (line 2). You can imagine the class defining a string as containing 47 methods, including `.upper()`, `.lower()`, etc., exactly like `calculate_price()` or `apply_coupon()` in this chapter. Finally, when we print the type of `greeting` (line 3), we see that it is of class `str`—that is, string. Similarly, let's have a quick look into a list:

```
[2]:
1 numbers = [1, 3, 5]
2 print(numbers.index(3))
3 print(type(numbers))
```

```
1
<class 'list'>
```

```
numbers is assigned one, three, five
print numbers dot index three
print type numbers
```

We instantiate the object `numbers` from the class `List` using a provided shortcut—that is, the square brackets—(line 1). Then, we call the method `.index()` with argument `3` and we directly print the output `1` (line 2). As for strings, we can imagine the code defining the list class as containing a sequence of 11 methods—the ones we summarized in Chapter 21. Finally, when printing the type, we see that it is of class list (line 3).

Let's code!

1. *More items in the online store!* Create two more objects from the class `Product` with the following characteristics:
   a. A beach ball called *Giant ball*, original price: 10 coins, launch discount: 0.50 coins. Calculate its price before and after applying the coupon `SAVE4`.
   b. A diary called *My adventures*, costing 15 coins, launch discount: 3 coins. Calculate its price before and after applying the coupon `SPRINGSALES30`.

   What do you get when you print each object?
2. *Sportswear testimonials.* You work for a famous sportswear company who hires champions for advertising.
   a. Create a class named `Athlete` that represents sports champions. The class contains 4 attributes: first name and last name (which are also the parameters of the constructor), sports type, and earnings. Then, create an object for each of the following athletes:
      - Yannick Sinner, tennis, 10000 coins;
      - Cristiano Ronaldo, soccer, 20000 coins;
      - Serena Williams, tennis, 15000 coins.

      Make sure that when you print an object, you print first and last name.
   b. Last year, soccer clothing sales had a small increase, so you give Ronaldo an earning increase. To do so, modify the class by adding a method called `increase_earning()`, which increases the earnings by a given percentage. Then, call the method `increase_earning()` for Ronaldo with a 1% bonus increase. How much is Ronaldo earning now?
   c. Similarly, tennis clothing sales in the past year had a relevant increase, so you decide to give a 5% earning increase to the athletes playing tennis. To do so, create a list called `athletes` whose elements are the created objects. Add a 5% raise to the athletes playing tennis using a `for` loop and an `if` condition. How much are they earning now?

36. Securing the online store

Encapsulation

In this chapter, we will build on the online store example to learn encapsulation, that is, how to define the access to attributes and methods by making them public or private. To learn what it means, have a look at the task and its solution below. You already know most of the code. Search for the differences between this implementation and the one in the previous chapter, and try to understand what they imply. Follow along with Notebook 36!

- While filling out the online store, you realize that you want to minimize errors that could compromise revenue. Thus, you make the attributes representing price and discount private and create get and set methods to access them. In addition, you realize that you need to include the tax amount in the calculation of the final price. Therefore, you implement a private method that calculates the tax amount and modify `calculate_price()` accordingly:

[1]:
```python
class Product:
    """Class representing a product"""

    # --- CONSTRUCTOR -------------------
    def __init__(self, name):
        """Constructor"""
        self.name       = name
        self.__price    = 0           # modified
        self.__discount = 0           # modified
        self.__tax_rate = 0.02        # added


    # --- GET/SET METHODS ---------------
    def get_price(self):              # added
        """Gets the price value"""
        return self.__price

    def set_price(self, price):       # added
        """Sets the price value"""
        if isinstance(price, (int, float)) and price > 0:
            self.__price = price
        else:
            raise ValueError("Price must be a number greater than 0")

    def get_discount(self):           # added
        """Gets the discount value"""
        return self.__discount

    def set_discount(self, discount): # added
        """Sets the discount value"""
        if isinstance(discount, (int, float)) and 0 < discount < self.__price:
            self.__discount = discount
```

```
        else:
            raise ValueError("Discount must be a number greater than 0 and less than
            the product's price")


    # --- METHODS -----------------------
    def apply_coupon(self, coupon):
        """Updates discount based on a coupon"""
        if coupon == "SAVE4":
            self.__discount = self.__discount + 4     # modified
            print("Coupon SAVE4 applied!")
        elif coupon == "SUMMER10":
            self.__discount = self.__discount + 10    # modified
            print("Coupon SUMMER10 applied!")
        else:
            print("Your coupon is not valid")

    def __calculate_tax(self, price):                   # added
        """Calculates tax on price"""
        tax = round(price * self.__tax_rate, 2)
        print("Tax amount on", price, "coins:", tax, "coins")
        return tax

    def calculate_price(self):
        """Calculates price after discount and tax"""
        # calculate the discounted price
        discounted_price = self.__price - self.__discount # modified
        # calculate tax on the discounted price
        tax = self.__calculate_tax(discounted_price)      # added
        # add tax to the discounted price
        taxed_price = discounted_price + tax              # added
        return taxed_price


    # --- BUILT-IN METHODS --------------
    def __str__(self):
        """Prints the object characteristics"""
        return "Product: " + self.name
```

- To test the new code, you create again the T-shirt object with its details, that is, name: `Feel good`; original price: `30` coins; and launch discount: `4` coins. Then, you calculate the T-shirt price before and after applying the coupon `SAVE4`:

[2]:

```
# creating the object
t_shirt = Product("Feel good")
print("t_shirt name:", t_shirt.name)

# providing and retrieving the original price
print("-> Original price")
t_shirt.set_price(30)
print("Price:", t_shirt.get_price(), "coins")

```

```
# providing and retrieving the discount
print("-> Launch discount")
t_shirt.set_discount(4)
print("Launch discount:", t_shirt.get_discount(), "coins")

# calculating the price after launch discount and tax
print("-> Price after launch discount and tax")
t_shirt_price = t_shirt.calculate_price()
print("Price:", t_shirt_price, "coins")

# applying the coupon and calculating the price
print("-> Price after launch discount, coupon, and tax")
t_shirt.apply_coupon("SAVE4")
t_shirt_price = t_shirt.calculate_price()
print("Price:", t_shirt_price, "coins")
```

```
(a)  Name: Feel good
(b)  -> Original price
(c)  Price: 30 coins
(d)  -> Launch discount
(e)  Launch discount: 4 coins
(f)  -> Price after launch discount and tax
(g)  Tax amount on 26 coins: 0.52 coins
(h)  Price: 26.52 coins
(i)  -> Price after launch discount, coupon, and tax
(j)  Coupon SAVE4 applied!
(k)  Total discount: 8 coins
(l)  Tax amount on 22 coins: 0.44 coins
(m)  Price: 22.44 coins
```

What are the novelties in the code above and what are their effects? Get some hints by solving the following exercise.

✎ True or false?

1.	To make an attribute or a method private, we add one underscore at the beginning of its name.	T	F
2.	self.__price, self.__discount, and self.__tax_rate are private attributes.	T	F
3.	An object can set the value of a private attribute by a set method and get its value by a get method.	T	F
4.	An object can directly call a private method.	T	F
5.	The final price is calculated by subtracting the discount from the original price and adding the tax.	T	F

Computational thinking and syntax

In the code in cell 1, we kept some attributes and methods public and converted others to private. This change affects how we access them, and it is the core of encapsulation. As we mentioned at the beginning of the chapter:

Encapsulation enables us to **define the access to attributes and methods** by making them **public or private**

What does this mean? Let's start by analyzing the **attributes** in the constructor (cell 1, lines 7–10). The attribute `self.name` (line 7) is the same as in Chapter 35 (cell 4, line 7). We call this kind of attributes **public**—you are already familiar with them. On the other side, the attributes `self.__price` and `self.__discount` (lines 8 and 9) are now **private**. To make an attribute private, we add a **double underscore** at the beginning of their names. Also the new attribute `self.__tax_rate` (line 10) is private because its name starts with a double underscore—we will use `self.__tax_rate` in the method `__calculate_tax()` (line 51). What is the main difference between public and private attributes? It's about the way we **access** them, that is, how we **assign a value** to an attribute or **retrieve the value** from an attribute. Let's refresh how to access public attributes.

Public attributes can be **directly** accessed both within a **class** and by an **object**

As you already know, within a **class**, we access a public attribute by using the syntax **`self.attribute_name`**. For example, within the class `Product`, we directly access the public attribute `self.name` in the built-in method `__str__()` (cell 1, line 69). Similarly, from an **object**, we access a public attribute by using **`object_name.attribute_name`**. For example, from `t_shirt`—which we instantiate in cell 2, line 2—we directly access the public attribute `t_shirt.name` (line 3) to print it (print (a)). What about private attributes?

Private attributes can be **directly** accessed only within a **class**; they can be accessed by an **object** only through **get/set methods**

In a **class**, we access a private attribute by using the syntax **`self.__attribute_name`**, similarly to what we would do for a public attribute. For example, in `calculate_price()` we directly access the private attributes by writing `self.__price` and `self.__discount` (line 58). On the other side, if an **object** tries to directly access a private attribute, we get an **`AttributeError`** saying that the attribute does not exist:

```
1 print("t_shirt price:", t_shirt.__price, "coins")
```

```
---------------------------------------
AttributeError       Traceback (most recent call last)
Cell In[3], line 1
----> 1 print("t_shirt price:", t_shirt.__price, "coins")

AttributeError: 'Product' object has no attribute '__price'
```

To avoid this error, we use an attribute's get and set methods (lines 13–34), which have a typical structure:

- A **get method**—also called **getter**—**returns a private attribute**. For example, `get_price()` returns `self.__price` (line 16), and `get_discount()` returns `self.__discount` (line 27). The name of a getter is usually composed of: (1) `get`, (2) underscore, and (3) the attribute name without the double underscore. The input is usually `self`.
- A **set method**—also called **setter**—**assigns the value of a parameter to a private attribute**. For example, in `set_price()`, we assign the input `price` to the private attribute `self.__price` (line 21), and

in `set_discount()`, we assign the input `discount` to the private attribute `self.__discount` (line 32). The assignment is often conditional on a **check of the parameter type and value**. We commonly check the type using `isinstance()`—in our case with a tuple as a second parameter containing the possible numerical types—and the value as a numerical range, as we learned in Chapter 30. In our example, the parameters `price` and `discount` must be an integer or a float greater than `0` (lines 20 and 31), and `discount` must also be less than the product price (line 31). If the conditions are not met (lines 22 and 33), we raise a `ValueError` with a message that conveniently combines the requirements for both type and value (lines 23 and 34)—it is also possible to raise `ValueError` and `TypeError` separately. The name of a setter is usually composed of: (1) `set`, (2) underscore, and (3) the attribute name without the double underscore. The name of the setter parameter is usually the same as the name of the attribute—without a double underscore—to show the correspondence (e.g., `price` and `self.__price`). Finally, setters usually take two inputs—that is, `self` and the parameter that will be assigned to the private attribute—and have no returns.

How does an **object** use get and set methods? By calling them with the usual syntax **`object_name.method_name`**. Let's have a look at `t_shirt` (cell 2). We provide the price of `30` coins by calling the setter with the command `t_shirt.set_price()` (line 7). When executing the code, the argument `30` goes through the `set_price()` method in the class (cell 1, lines 18–23). Because `30` satisfies the type and value checks—it is an integer greater than `0`—it will be assigned to the attribute `self.__price` (line 21). We can retrieve the price amount by calling the getter with the command `t_shirt.get_price()`, which returns the value of the attribute `self.__price` (cell 1, line 16), that is, `30` coins, which we print (cell 2, line 8; print (c)). Similarly, we provide the discount of `4` coins by calling the setter with the command `t_shirt.set_discount()` (cell 2, line 12). In the setter (cell 1, lines 29–34), the value of `discount` satisfies the type and value conditions—it is an integer between `0` and `30`—(line 31) and thus is assigned to the attribute `self.__discount` (line 32). Finally, we can retrieve and print the discount amount by calling the getter with the command `t_shirt.get_discount()` (cell 2, line 13; print (e)).

Now that syntax and functionality are clear, you might wonder: **why and when should I make an attribute public or private**? It's a matter of **protection of the integrity code** and the judgment often depends on the context and the coder's interpretation of the whole project. In general, we make an attribute **public** when it can be freely accessed **without risks** of unintended changes or bugs. In our example, the attribute `self.name` is public to allow easy change of the product name, because it would not cause any economic damage to the online store. On the other side, we make an attribute **private** when we want some **control** on how it is used and modified. In our code, `self.__price` and `self.__discount` are private because we want to check their types and values before assignments to avoid compromising revenues. What about `self.__tax_rate` (cell 1, line 10)? Why doesn't it have a getter and a setter? Because the tax rate is the same for all items, we do not want individual objects to be able to modify it.

Let's now look into encapsulation for **methods**. It works as follows:

Public methods can be **directly** accessed both within a **class** and by an **object**.
Private methods can be accessed only within a **class**—**not** by an **object**

Let's analyze the three methods in the `Product` class (cell 1, lines 37–63). The first method is `apply_coupon()` (lines 38–47), which is public, that is, its accessibility works the same way as in Chap-

ter 35. However, in the method's code the attribute `self.__discount` is now private (lines 41 and 44). The second method is the newly added `__calculate_tax()` (lines 49–53), which is private. Similarly to attributes, we make a method private by adding a **double underscore at the beginning of its name**. Within `__calculate_tax()`, we calculate the tax amount as the product of the input price and the tax rate—defined by the private attribute `self.__tax_rate`—and we round it to the first two decimals for convenience (line 51). Then, we print the tax amount (line 52), and we return it (line 53). Finally, the third method is `calculate_price()` (lines 55–63), which remains public. However, its code is modified with respect to Chapter 35. We calculate the discounted price by subtracting `self.__discount` from `self.__price`—both of them are now private attributes—and we store the result in `discounted_price` (line 58). Then, we calculate the tax amount by calling the private method `__calculate_tax()` (line 60). To **call a public or private method** within a class, we use the syntax **`self.method_name()`**. The method `__calculate_tax()` takes `discounted_price` as an input and returns the tax amount as an output. Finally, the tax amount is summed to the discounted price and stored in `taxed_price` (line 62), which is finally returned (line 63).

How are the methods used by an **object**? As defined above, objects can **call only public methods** using the syntax `object_name.method_name()`, as we learned in the previous chapter. They can **not call private methods**. In our example, we call the methods in the same sequence as in Chapter 35 (cell 5, lines 7–20), that is, first `calculate_price()` to calculate the price, then `apply_coupon()` to apply a coupon, and finally `calculate_price()` again to recalculate the price (cell 2, lines 15–24). However, the outcome is different because the price calculation now includes a tax. Let's briefly go through the code. First, we calculate the price with launch discount and tax by using the public method `calculate_price()` (cell 2, line 17). Within the method, we subtract the discount of `4` coins from the original price of `30` coins to obtain the discounted price of `26` coins (cell 1, line 58). Then, we calculate the tax by calling the private method `__calculate_tax()` (line 60), which receives the discounted price of `26` coins as an input, multiplies it by the tax rate `0.02` (defined at line 10) and rounds the result to two digits (line 51). The obtained amount of `0.52` coins is printed (line 52; print (g)) and returned (line 53) to the variable `tax` in `calculate_price()` (line 60). Finally, the value of `tax` is summed to the value of `discounted_price` to obtain `26.52` coins (line 62). This value is returned (line 63) and assigned to the variable `t_shirt_price` (cell 2, line 17), which is then printed (line 18; print (h)). Second, we apply the coupon `SAVE4`. We call the public method `apply_coupon()` (line 22), which adds `4` coins to the private attribute `self.__discount` (cell 1, line 41) and prints a message (line 42; print (j)). Finally, we recalculate the T-shirt price after launch discount, coupon, and tax. We call `calculate_price()` (cell 2, line 23), which executes the same steps as above. Because the total discount is now `8` coins, the tax amount is calculated on `22` coins, resulting in `0.44` coins (print (l)). The final amount of `22.44` coins is returned and assigned to the variable `t_shirt_price` (line 23), which is finally printed (line 24; print (m)).

What happens if an object tries to call a private method? Similarly to attributes, we get an error:

```
1 t_shirt.__calculate_tax(price)
```

```
---------------------------------------
AttributeError      Traceback (most recent call last)
Cell In[4], line 1
----> 1 t_shirt.__calculate_tax(price)

AttributeError: 'Product' object has no attribute 'calculate_tax'
```

Even if we tried to access a private method, we obtain an `AttributeError`, as we can see in the last line of the message. This can be misleading. However, the presence of the round brackets after the

method name in the line indicated by the arrow will help us detect that we are dealing with a method, and not an attribute.

Once more, you might ask: **when do we make a method public or private**? Similarly to attributes, we make a method public when it can be used by an object without compromising the code, whereas we make a method private when we do not want the object to access it or when it supports some other methods, like `__calculate_tax()`.

To conclude, find a summary of the syntax of accessibility to public and private attributes and methods within the same class and by an object in Table 36.1. It is important to mention that in the majority of programming languages, encapsulation also includes a third category of attributes and methods called **protected**. However, this goes beyond the scope of exploring the basics of object-oriented programming of this book.

Class attributes and methods	Accessibility within the class	Accessibility by an object
Public Attribute `.name`	Direct `self.name`	Direct `t_shirt.name`

Private Attribute
`.__price`
	Direct
	`self.__price`
		Using getters and setters
		`t_shirt.get_price()`
		`t_shirt.set_price()`

Public Method `calculate_price()`	Direct `self.calculate_price()`	Direct `t_shirt.calculate_price()`
Private Method `__calculate_tax()`	Direct `self.__calculate_tax()`	Not possible

Table 36.1. Definition of a class's public and private attributes and methods (left column) and their accessibility within the same class (central column), and from an object instantiated from the class (right column).

Recap

- In encapsulation, we define public or private access to attributes and methods.
- The name of a private attribute or a method starts with a double underscore.
- A public attribute or method can be accessed both within the class and by the object, whereas a private attribute or method can be accessed only within the class.
- An object can call a get method—or getter—to retrieve the value of a private attribute and a set method—or setter—to assign a value to a private attribute.

The self is a little crab!

In the past two chapters, we used `self` extensively within the class, considering it as a syntax particle for convenience. But what is `self`? A fun way is to think of it like a little crab whose left legs represent the attributes and right legs represent the methods (Figure 36.1, left). Because `self` is the first input parameter of the methods of a class—including the constructor!—every method has access to all the legs of the crab, that is, all the attributes and methods. For example, the getter `get_price()` (cell 1, lines 14–16) receives `self`—the whole crab!—as an input and

returns `self.__price`—the second left leg! The method `calculate_price()` (lines 55–63) receives `self` as an input and uses not only the attributes `self.__price` and `self.__discount` (line 58)—the second and third left legs—but also the method `__calculate_tax()` (line 60)—the second right leg (Figure 36.1, right). The role of `self` is to tie together all the attributes and methods and make them accessible to the other attributes and methods. This accessibility is made possible using **dot notation**, that is, using the syntax: (1) `self`, (2) dot, and (3) name of attribute or method. When operating with an object, `self` represents that object within the class, because it carries the attribute values of that object. For example, when calling `calculate_price()` for the object `t_shirt`, the operation `self.__price-self.__discount` is `30-4`. In summary:

> `self` represent the **current instance** of the class and
> is used to **access** its attributes and methods within the class

Finally, the name `self` is a convention. `self` is not a reserved word in Python, and it is colored black in Jupyter Notebook, like any other variable. However, it is recommended to use the name `self` to ensure code readability.

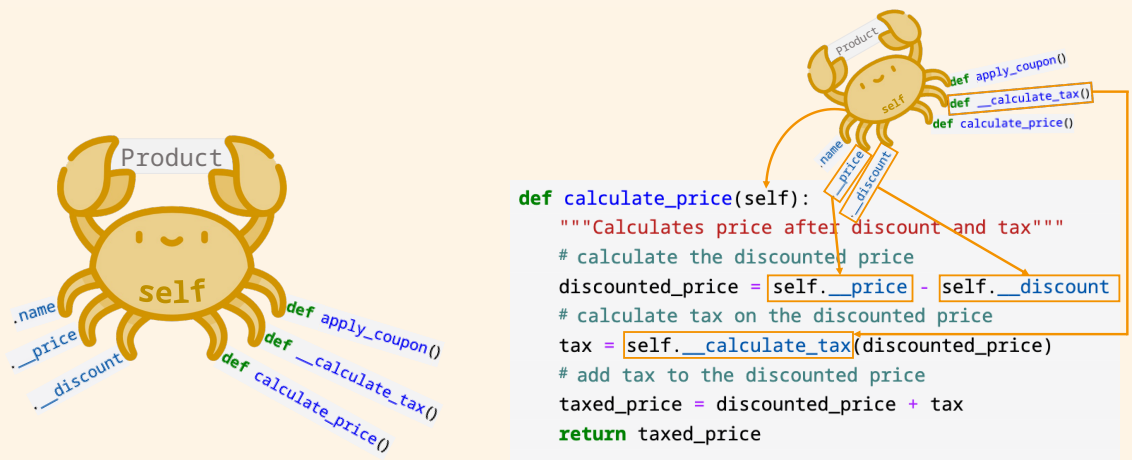


Figure 36.1. `self` as a crab, with the left legs representing the attributes and the right legs representing the methods (left), and its role in a class method (right).

Let's code!

1. *Let's add more items to the online store!* From the updated class `Product`, instantiate the following objects from the previous chapter:
   - A lamp called *Lux*, original price: 40 coins, no launch discount. Calculate its price before and after applying the coupon `SUMMER10`.
   - A beach ball called *Giant ball*, original price: 10 coins, launch discount: 0.50 coins. Calculate its price before and after applying the coupon `SAVE4`.
   - A diary called *My adventures*, costing 15 coins, launch discount: 3 coins. Calculate its price before and after applying the coupon `SPRINGSALES30`.
2. *Running a used car garage.* You run a used car garage and have just received a new vehicle that needs to be added to the garage database. The car is a gray WJ from the LP brand. From the current report, you see that it traveled 30000 km in 2024 and 50000 km in 2025. It also has a dent on the front left door.

a. Create a class representing a car similar to the one above. What attributes will you create and of what types? Which attributes will be public and which ones private? For the private attributes, write getters and setters.
b. Now you go for the inspection. You notice that the car has a another dent on the left back door. How do you modify the class to add this new information?
c. Finally, you want to print a report on the car's status. The report must contain all the car characteristics, the amount of km the car traveled each year and their average, and the number of dents and their locations. Add at least two methods to perform this final task. Which methods will be public and which methods will be private?

37. How can I read a book sample?

Inheritance

So far, we have learned the basics of object-oriented programming. That is, we've learned the concepts of classes and objects, with their public and private attributes. In this chapter, we will extend the capabilities of classes and objects by using inheritance. Here, you will get familiar with the concepts of parent and children classes and their properties. Ready? Let's enhance the online store with some new features! Follow along with Notebook 37.

- It's time to add books to the online store. For their webpages, you need to add a Read Sample button so that customers can preview the books before buying. However, you have to make sure that this button does not appear on the pages of other products, such as clothing or furniture. How can you do it?

Have a look at Figure 37.1. Any ideas? Everything will be clearer in a bit!

Figure 37.1. The parent class `Product` (left), the child class `Book` with the additional functionality Read sample (middle), and the object `coding_book` (right).

Let's continue with the code!

- You keep the class `Product` as it is—see Chapter 36, cell 1 and Notebook 37, cell 1. That will be the parent class.
- You create a child class representing books that inherits all attributes and methods from the `Product` class. Then, you add a private attribute representing the book sample—with its get and set methods—and create a public method that prints the sample:

[2]:
```
class Book(Product):
    """Child class representing a book"""

    # --- CONSTRUCTOR -------------------
    def __init__(self, name):
        """Constructor"""
        super().__init__(name)
        self.__book_sample = ""

```

```
10
11      # --- GET/SET METHODS ---------------
12      def get_book_sample(self):
13          """Gets the book sample"""
14          return self.__book_sample
15
16      def set_book_sample(self, book_sample):
17          """Sets the book sample"""
18          if isinstance(book_sample, str):
19              self.__book_sample = book_sample
20          else:
21              raise TypeError("book_sample must be a string")
22
23
24      # --- METHODS -----------------------
25      def read_sample(self):
26          """Prints the book sample"""
27          if self.__book_sample != "":
28              print(self.__book_sample, "[...] - Enjoying the book? Buy it!")
29          else:
30              print("Book sample not available")
```

- To test the new code, you instantiate an object representing a coding book called *Let's code*, original price 20 coins, and launch discount 2 coins. Then, you print the book's characteristics, its price after applying the coupon `SUMMER10`, and its sample:

```
[3]:
1  # creating the object and setting the attributes
2  coding_book = Book("Let's code!")
3  coding_book.set_price(20)
4  coding_book.set_discount(2)
5
6  # printing the characteristics
7  print("Book name:", coding_book.name, "| original price:", coding_book.get_price(),
   "coins | launch discount:", coding_book.get_discount(), "coins")
8
9  # printing the price
10 print("-> Price after launch discount, coupon, and tax")
11 coding_book.apply_coupon("SUMMER10")
12 print("Price: ", coding_book.calculate_price(), "coins")
13
14 # reading the book sample
15 print("-> Reading book sample")
16 coding_book.set_book_sample("Coding is a lot about telling a computer what to do")
17 coding_book.read_sample()
```

```
(a)  Book name: Let's code! | original price: 20 coins | launch discount: 2 coins
(b)  -> Price after launch discount, coupon, and tax
(c)  Coupon SUMMER10 applied!
(d)  Tax amount on 8 coins: 0.16 coins
(e)  Price: 8.16 coins
(f)  -> Reading book sample
(g)  Coding is a lot about telling a computer what to do [...] - Enjoying the book? Buy it!
```

Get a few more hints about inheritance by solving the following exercise before reading the next section!

✎ True or false?

1. `Product` is the child class and `Book` is the parent class. T F
2. A child class can use attributes and methods of the parent class. T F
3. A child class cannot add its own attributes and methods. T F
4. The method `read_sample()` prints the value of `self.__book_sample`. T F

Computational thinking and syntax

Did you get an idea of what inheritance is in object-oriented programming? Consider the facial features of a child. Let's say a boy inherited many characteristics from his parents, such as his eye and hair color. But he has also some features that are unique to him, such as his smile. In object-oriented programming, inheritance is similar. A **child class**—or **subclass**—inherits its attributes and methods from its **parent class**—or **superclass**—and can add new ones that are unique to itself. In other words:

> **Inheritance** is a mechanism that enables a **child class** to
> **derive and extend the attributes and methods** of a **parent class**

Let's see how inheritance works by analyzing the code of the online store. Let's start with the syntax, and then we will look into the functionality. The class `Book` (cell 2) is the child class of the parent class `Product` (Chapter 36, cell 1). To create a child class, we write: (1) keyword `class`, (2) class name, and (3) **name of the parent class in between round brackets** (cell 2, line 1). With this line, **we pass all the attributes and methods of the parent class to the child class**—as if they were the eyes and hair color from the parents to their child. Right below, we write a one-line documentation describing the class (line 2). Then, we write the **constructor** (lines 5–8), which is needed **only when new attributes are added** with respect to the parent class—you will see an example of a child class without a constructor in the next chapter. In our example, the constructor takes `self` and `name` as parameters (line 5), like in the parent class (Chapter 36, cell 1, line 5). Within the constructor, we must call the parent class's constructor by using a typical command composed of: (1) built-in function **`super()`**, which further gives the child class access to all the parent class's attributes and methods; (2) a dot; and (3) the call to the constructor of the parent class. Then, we add a new private attribute called `self.__book_sample`, which we initialize as an empty string (line 8). This **addition** and the ones below are **unique to the child class**—as if they were the child's smile in the example above. The **parent class will not have access to the new attribute** of the child class—as parents cannot inherit their child's smile. Because `self.__book_sample` is private, we write a getter (lines 12–14) to return its value, and a setter (lines 16–21) to assign a value. In the setter, the assignment is conditional on the input being a string (line 18). Finally, we add a new method that prints the book sample (lines 25–30). Within the method, if the book sample is not an empty string (line 27), we print the value of `self.__book_sample` (line 28). Otherwise (line 29), we print that the book sample is not available (line 30). As with attributes, the **parent class will not have access to the new method** of the child class.

What happens on the **object** side? Let's look into cell 3. We instantiate the object `coding_book` from the child class `Book` (line 2). Then, we set the price by using the setter `set_price()` inherited from the parent class `Product` (line 3). Similarly, we set the discount by using `set_discount()` inherited from `Product` (line 4). Then, we print the book characteristics by directly accessing the attribute `coding_book.name` and using the getters `get_price()` and `get_discount()`, which are once more inherited from the parent class `Product` (line 7; print (a)). Similarly, we calculate the price after the initial discount, coupon, and tax by using the methods `apply_coupon()` (line 11; print (c)) and `calculate_price()` (line 12; prints (d) and (e)), inherited from `Product`. Finally, we use the new attribute and methods added in the child class `Book`. First, we set the book sample by using the setter `set_book_sample()` (line 16). Because the argument is a string (cell 2, line 18), it gets assigned to `self.__sample` within the class (line 19). Finally, we call the method `read_sample()` to print the sample (cell 2, line 17; print (g)).

In this chapter, we learned the concepts of parent class and child class and their properties. Can we create **more than one child class** from the same parent class? Yes! You will see an example in the next chapter. And can we still instantiate objects from the parent class? Yes, as you will see in the first coding exercise below. Before moving to the next sections, take some time to refresh the accessibility to public and private attributes and methods of the parent class by a child class and a child object with the help of Table 37.1.

Parent class attributes and methods	Accessibility by a child class	Accessibility by a child object
Public Attribute `.name`	Direct `self.name`	Direct `coding_book.name`

Private Attribute
`.__price`
	Using getters and setters
	`self.get_price()`
	`self.set_price()`
		Using getters and setters
		`coding_book.get_price()`
		`coding_book.set_price()`

Public Method `calculate_price()`	Direct `self.calculate_price()`	Direct `coding_book.calculate_price()`
Private Method `__calculate_tax()`	Not possible	Not possible

Table 37.1. Definition of public and private attributes of a parent class (left column) and their accessibility from a child class (central column) and from an object instantiated from the child class (right column).

Recap

- Inheritance consists in creating one or more child classes (or subclasses) from a parent class (or superclass).
- The child class inherits attributes and methods from the parent class and can add new ones.
- To create a child class, we use the keyword `class` followed by the child class name and the parent class name in between round brackets.
- A child class needs a constructor only when it adds new attributes. In this case, the first line of the constructor is composed of the built-in function `super()` followed by dot and the constructor of the parent class.

Attributes, methods, and round brackets

When using an attribute or calling a method, it can happen to mistakenly add or remove the round brackets at the end of their name. Here is how to recognize these errors and correct them. When we mistakenly add round brackets after an attribute, we obtain the following error:

```
1 print(coding_book.name())
```

```
---------------------------------------
IndexError                Traceback (most recent call last)
Cell In[4], line 1
----> 1 print(coding_book.name())
TypeError: 'str' object is not callable
```

It is a `TypeError` saying that a string is not callable, which—as you might remember—means that we cannot call a string as if it were a function. To correct the error, we simply remove the round brackets. On the other side, when we forget to add the round brackets after a method—this might happen especially if the method does not have any inputs—we do not obtain an error message, but something similar to the following:

```
1 print(coding_book.calculate_price)
```

```
<bound method Product.calculate_price of <__main__.Book object at 0x10783d850>>
```

It says that we are looking at the method `calculate_price()` in the computer memory. We are not calling it! To call and execute the method, we simply add a pair of round brackets, along with any required arguments, immediately after the method name.

Let's code!

1. *Testing the parent class.* Instantiate an object `t_shirt` from the parent class `Product` having the same characteristics as in the previous chapters, that is, name: *Feel good*, initial price: 30 coins, launch discount: 4 coins, coupon `SAVE4`. Try to set a book sample and call the method `read_sample()`. What happens and why?
2. *Buying an electric car.* You want to buy a new electric car, and these are the cars you like:
   - Model *E-Nature*, energy consumption: 15 kWh per 100 km, battery capacity: 75 kWh;
   - Model *E-Green*, energy consumption: 18 kWh per 100 km, battery capacity: 40 kWh.

   You want to calculate whether the battery capacity of these cars is sufficient for a trip of 300 km. To do so, you create:
   - A parent class called `Vehicle`. Its constructor contains the attributes representing model and energy consumption. The method calculates the energy needed as the energy consumption divided by 100 and multiplied by the distance to drive.
   - A child class called `ElectricCar`. It adds an attribute representing the battery capacity and a method that prints whether the trip can be completed with the calculated energy.

   Which car will you buy?
3. *School management system.* You have to develop a software program for a school that calculates the total number of hours each teacher works per week and the average grade of each student. The characteristics that teachers and students have in common are first name and last name. In addition to these, each teacher has a number of working hours per day and a number of working days per week, whereas each student has grades in math, history, and music. How would you represent this situation using object-oriented programming? Test your solution for the following two teachers: Tom Kind who works 5 hours a day, 4 days a week; and Ani Heart

who works 3 hours a day, 5 days a week. Then, for the following three students: Amber Brown has 8 in math, 9 in history, and 5 in music; Mark Fox has 5 in math, 8 in history, and 7 in music; and Sophia Baker has 9 in math, 7 in history, and 8 in music.

38. Customizing the coupon for electronics

Polymorphism

In this final chapter, you will learn about polymorphism, which literally means "many forms" in Ancient Greek. It is the property of a child class that allows us to overwrite one or more methods of the parent class. As a result, objects from the parent class and its child classes can use a method with the same name but perform different actions. Let's see what this means in practice by performing the last addition to the online store. Follow along with Notebook 38!

- To complete the online store, you add electronic products. When purchasing them, customers can only use the coupon `TECH100`, worth 100 coins, instead of the `SAVE4` and `SUMMER10` coupons, which can be used for other products. How would you change the apply coupon functionality to satisfy this new requirement?

The solution to our task is depicted in Figure 38.1. We will create a child class called `Electronics` that appears similar to its parent class `Products`. However, the method `apply_coupon()` will contain different code to allow the use of the `TECH100` coupon. We will test our code with a new object called `laptop`.

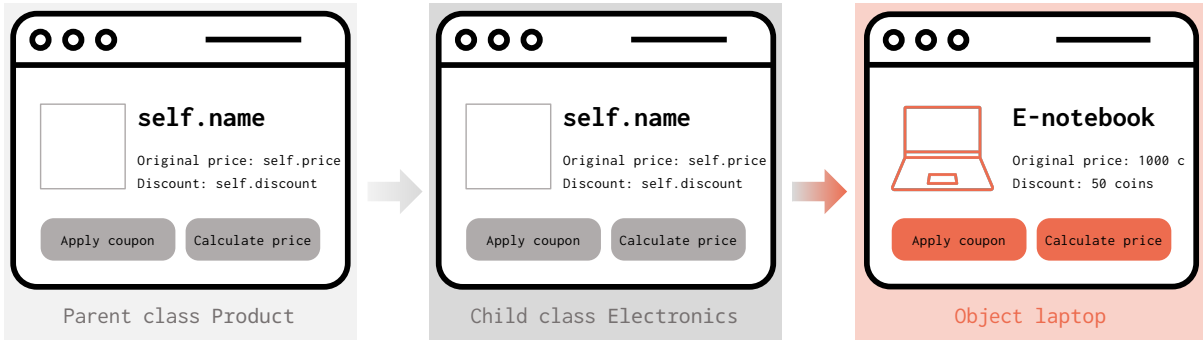


Figure 38.1. The parent class `Product` (left), the child class `Electronics` (middle), and the object `laptop` (right). Despite the similarity between parent and child class, the method `apply_coupon()` performs different functionalities.

Let's have a look at the code!

- You keep the same parent class `Product` as in Chapter 36, cell 1—find it also in Notebook 38, cell 1.

- You create the new child class `Electronics` that inherits from `Product` and overwrites its method `apply_coupon()`:

[2]:
```
class Electronics(Product):
    """Child class representing an electronic product"""

    # --- METHODS -----------------------
    def apply_coupon(self, coupon):
        """Updates discount based on a coupon - overwrites parent method"""
        if coupon == "TECH100":
            self.set_discount(self.get_discount() + 100)
            print("Coupon TECH100 applied!")
        else:
            print("Your coupon is not valid")
```

- To check the correctness of the code, you create a new object `laptop` named E-notebook, with the original price of 1000 coins, and launch discount of 50 coins. Then, you calculate the price before and after applying the coupon `TECH100`:

[3]:
```
# instantiating the object
laptop = Electronics("E-notebook")
laptop.set_price(1000)
laptop.set_discount(50)

# calculating price after launch discount
print("-> Price after launch discount")
laptop_price = laptop.calculate_price()
print("Price:", laptop_price, "coins")

# calculating price after coupon
print("-> Price after coupon")
laptop_price.apply_coupon("TECH100")
laptop_price = laptop.calculate_price()
print("Price:", laptop_price, "coins")
```

```
(a)  -> Price after launch discount
(b)  Tax amount on 950 coins: 19.0 coins
(c)  Price: 969.0 coins
(d)  -> Price after coupon
(e)  Coupon TECH100 applied!
(f)  Tax amount on 850 coins: 17.0 coins
(g)  Price: 867.0 coins
```

✎ True or false?

1. The child class Electronics inherits attributes and methods from the parent class Product. T F
2. The child class Electronics does not overwrite the method apply_coupon() of the parent class Product. T F
3. The objects of the child class Electronics can use only the coupon TECH100. T F

Computational thinking and syntax

Let's start by analyzing the class (cell 2). We create `Electronics`, a child class that inherits attributes and methods from the parent class `Product` (line 1). Because `Electronics` does not add new attributes beyond those of the parent class, we omit the constructor (see Chapter 37). The only change we make is to overwrite the method `apply_coupon()` (lines 5–11) from the parent class (Chapter 36, cell 1, lines 38–47). Overwriting methods is the core of polymorphism. Let's recall what we learned at the beginning of the chapter:

> **Polymorphism** is a mechanism that enables a **child class** to **overwrite a method** of the parent class while **keeping the same method name**

In other words, in the child class we write a method with the same name as in the parent class but containing different commands. In our example, we want to allow customers to exclusively use the `TECH100` coupon when buying electronics. Thus, in the method `apply_coupon()` of `Electronics` we implement an `if/else` construct, where if the coupon is `TECH100` (line 7), we add `100` coins to the discount (line 8) and we print that the coupon was successfully applied (line 9). Otherwise (line 10), we print that the coupon is not valid (line 11). As you might remember from the previous chapter, because `self.__discount` is a private variable in the parent class, we cannot access it directly from the child class, thus we use the get and set methods (see Table 37.1).

Let's now look at the object (cell 3). We instantiate `laptop` with the name `E-notebook` (line 2), and we call the setters `set_price()` with argument `1000` coins (line 3) and `set_discount()` with argument `50` coins (line 4). Then, we calculate the price by calling the method `calculate_price()` (line 8)—which includes the calculation of the tax amount (print (b))—and we print the price after tax (line 9; print (c)). Afterwards, we call the newly implemented `apply_coupon()` (line 13), and we get the message that the coupon was successfully applied (print (e)). Finally, we recalculate the price by calling `calculate_price()` once more (line 14)—obtaining the print of the tax amount (print (f))—and we print the final price (line 15; print (g)). As you can see, `laptop` called `apply_coupon()` from `Electronics`—not from `Product`!—and `calculate_price()` from `Product` because in `Electronics` we did not overwrite `calculate_price()`.

What happens if we apply a coupon that is valid for the parent class? Let's try with `SAVE4`:

```
1  laptop.apply_coupon("SAVE4")
```

```
Your coupon is not valid
```

The output message tells us that the coupon is not valid. This is because `laptop` is an instantiation of the child class `Electronics`, and thus it uses the method `apply_coupon()` in `Electronics`, not in `Product`, as we mentioned previously.

What if we use the coupons `SAVE4` and `TECH100` for the objects `t_shirt` of class `Product`?

```
1  t_shirt = Product("Feel good")
2  t_shirt.apply_coupon("SAVE4")
3  t_shirt.apply_coupon("TECH100")
```

```
(a)  Coupon SAVE4 applied!
(b)  Your coupon is not valid
```

Because it is of the class `Product`, `t_shirt` calls the implementation of `apply_coupon()` from the parent class (Chapter 36, cell 1, lines 38–47). There, the coupon `SAVE4` satisfies the `if` conditions (line 40), thus

it can be applied (cell above, line 2; print (a)). On the other side, the coupon `TECH100` does not satisfy either the `if` (Chapter 36, cell 1, line 40) or the `elif` (line 43) conditions; therefore, it is not considered valid (cell above, line 3; print (b)).

Similarly, what happens when we use the coupons `SAVE4` and `TECH100` for `coding_book` of class `Book`, which we introduced in the previous chapter?

```
1  coding_book = Book("Let's code!")
2  coding_book.apply_coupon("SAVE4")
3  coding_book.apply_coupon("TECH100")
```

```
(a)  Coupon SAVE4 applied!
(b)  Your coupon is not valid
```

Because the child class `Book` does not overwrite the method `apply_coupon()` of `Product`, the object `coding_book` uses the method `apply_coupon()` from `Product`. Therefore, similarly to the outcomes for `t_shirt`, the coupon `SAVE4` is applicable (cell above, line 2; print (a)), whereas the coupon `TECH100` is not (cell above, line 3; print (b)).

In summary, for `laptop`, `t_shirt`, and `coding_book`, we called the function `apply_coupon()` using the same syntax `object_name.method_name()` but we obtained different results—`TECH100` is valid for `laptop` but not for `t_shirt` and `coding_book`, whereas `SAVE4` is valid for `t_shirt` and `coding_book` but not for `laptop`. This is the strength of polymorphism: **when we instantiate an object from a class, the appropriate method is automatically selected based on the object's class, without requiring specific intervention**.

In the last four chapters, we learned the basics of object-oriented programming, a way to code based on the representation of the world with classes and objects, characterized by attributes—

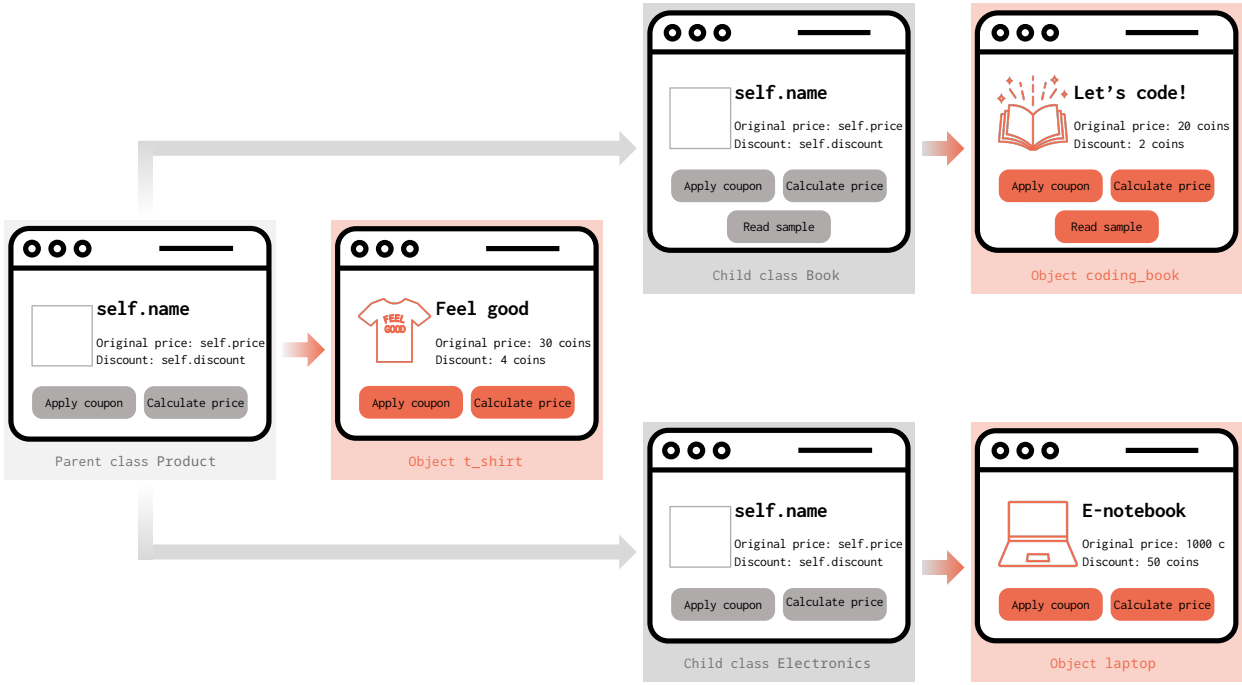


Figure 38.2. Graphical summary of the representation of an online store in object-oriented programming.

representing their properties—and methods—representing their actions. As an example, we implemented a simplified version of an online store (see Figure 38.2). From the parent class `Product`, we instantiated the object `t_shirt`—and the object `lamp`, not represented here for simplicity—(Chapter 35) and we created two child classes, `Book` and `Electronics`. `Book` inherits all attributes and methods from `Product` and adds the private attribute `self.__book_sample`—with its getter `get_book_sample()` and setter `set_book_sample()`—and the public method `read_sample()` (Chapter 37). `Electronics` inherits all attributes and methods without adding any of its own, but overwrites the method `apply_coupon()`, leveraging on polymorphism (Chapter 38). We can imagine the online store expanding with other child classes to represent beauty products, food, or games. In addition, child classes can have their own child classes too. `Electronics` can have child classes representing smartphones, with their specific features such as screen size and camera quality, and speakers, with features such as maximum output power and connectivity technology. Thanks to object-oriented programming, we can represent the real world through a **hierarchy of parent and child classes**, with code that is modular, reusable, and easily extensible.

Recap

- Polymorphism enables child classes to overwrite parent class methods while keeping the same method name.
- When it is called, the correct method is automatically selected based on the object's class.

How do I use artificial intelligence when coding?

In the last years, artificial intelligence and its large language models—such as ChatGPT, Claude Code, Copilot, etc.—have emerged as a support in coding. These tools are a fantastic resource when creating functions, writing code documentation, and brainstorming when encountering obstacles. However, we have to be aware of a few limitations. First, these models are mainly created from data from the internet, and they often provide **outputs that are average**, rather than original. Thus, whenever we need to code something common, their support is useful; but when we need to code something specific, we might still have to do the hard work. Second, the outputs that these models give are often **not reproducible**, which means that if we prompt the same requirements at different times, we might obtain different answers. Therefore, we always have to check and validate the model outputs because they might vary. Last, these models tend to be **verbose**, proposing code that is overcomplicated or adding commands that are not needed to solve the task. Therefore, we always need to read the code carefully and select the commands that do what we require. The **human judgment** of the output of a language models is still fundamental, and my hope is that with this book I could provide you the tools to better **understand the elements of coding** and **develop computational thinking** to analyze and solve computational tasks. Thank you for learning with me!

Let's code!

1. *Polygons.* Create a parent class representing a regular polygon. The class has two attributes, the number of sides and side length; and two methods, one to calculate the perimeter and one to calculate the area. Then, create a child class representing an equilateral triangle. The area of a triangle is calculated as $\frac{\sqrt{3}}{4} \cdot s^2$, where s is the side. In addition, implement a child class representing a square whose area is calculated as the square of the side. Finally, instantiate two objects, a triangle and a square of side 4, and calculate their perimeter and area
2. *Online food ordering.* You have to create a new software program for online food ordering. The client requested a basic ordering system with two customized options, one for pizza and one for burgers. The base order has an initial price of 10 coins. For pizza, the price increases by 20% if the size is medium, and by 50% if the size is large. For burgers, the price increases by 1.5 coins if the customer adds cheese and by 1 coin if they add onions. To test your code, create four orders: a small pizza, a medium pizza, and a large pizza, as well as a burger with both extra cheese and onions. What is the total cost?

Acknowledgments

This book would not exist without the Python language, the Jupyter environment, the LaTeX project for scientific writing, and the free images available online. I am gratefully to all the people who have created, extended, and maintain these open-source projects. I particularly thank Fernando Pérez, Chris Holdgraf, Ana Ruvalcaba, Jason Grout, and Sylvain Corlay from Project Jupyter, who have inspired and supported me.

I am thankful to the women who paved the way. Thanks to Ani Adhikari, Marta Baldocchi, and Diletta Torlasco, who inspire me to be a better teacher. And thanks to Gabriella Greison—thank you, Elena Tacca, for gifting me one of her books!—and Susan Holmes for writing scientific books. You gave me the courage of writing this book—if they did, I can too!

A sincere thanks to all the students who, over the years, have provided precious feedback on the Python course I taught, whose material is now collected in this book. Through our interactions, I could refine ideas, simplify concepts, and streamline the incremental progression of topics. I am particularly grateful to Andreas Hadjistyllis—for whom the material was first created—Aleksej Krilosov, Erika Pace, Evangelos Gargoulakis, Joël Guillaume, Marina Yvory, Nasim Sangani, Paul Wyman, and Thomas Drenik.

I deeply thank the early readers of the electronic version of this book, who patiently read the chapters as they were released, offered continuous support, and provided valuable suggestions for improvements. A special thanks to Arno de Graaf, who reviewed the exercise solutions.

This book has already inspired open-source derived work, which makes me extremely proud. A heartfelt thanks to Rodrigo Ernesto Álvares Aguilera, who created an open-source JupyterLab-based learning environment for middle school students using material from this book and translated the Jupyter Notebooks to Mexican Spanish. Thanks also to Nick Chomey for contributing to the translation.

I thank my family, friends, colleagues, and all those who supported and encouraged me throughout the years I worked to transform my "Learn Python with Jupyter" course into this book.

And finally, I thank you, dear Coder, for learning with me how to read and write a new language. I hope that this knowledge will help you better understand the world around us and empower you to contribute to its beauty.

References and sources

- A few examples and exercises in the book were inspired from online material or created using ChatGPT. The example in Chapter 17 "Do you want more candies?" was inspired from the book "Coding for Kids: Python: Learn to Code with 50 Awesome Games and Activities" by Adrienne Tacke. Rockridge Press. 2019
- Figure 1.1 is modified from the original by Cy21 and downloaded from `https://commons.wikimedia.org/wiki/File:QWERTY-home-keys-position.svg`
- Figure 16.1 is by the U.S. Naval Historical Center Online Library Photograph and downloaded from `https://en.wikipedia.org/wiki/Debugging#/media/File:First_Computer_Bug,_1945.jpg`
- The tree in Figure 31.2 is by OpenClipart-Vectors and downloaded from `https://pixabay.com/vectors/tree-nature-drawing-plant-branches-2027899`
- The icons in Figures 33.5, 35.1, 36.1, 37.1, 38.1, and 38.2 are downloaded from `www.freepik.com`