Learn Python with Jupyter

Serena Bonaretti

www.learnpythonwithjupyter.com

Dear coder,

Thanks for your interest in *Learn Python with Jupyter*! I hope it will help you learn computational thinking and coding in Python!

The writing of *Learn Python with Jupyter* is a **work in progress**. I release a new chapter every 4-6 weeks, as I write the book around working hours. That is why it is taking a bit of time.

Learn Python with Jupyter is **open and free** and it will remain open and free. Upon completion of the book, I might publish a printed copy. That would have a (low) cost to cover printing and distribution.

You can find some information about the **construction** of *Learn Python with Jupyter* in this Jupyter Blog post: `https://blog.jupyter.org/introducing-learn-python-with-jupyter-11214f152159`. I will write more extensively about linguistic, pedagogical, and psychological aspects behind *Learn Python with Jupyter* in a future post.

If you have any comments or questions, please **email me** at serena.bonaretti.research@gmail.com, and I will be happy to reply.

Thank you for learning with me,
Serena

Learn Python with Jupyter

Serena Bonaretti

www.learnpythonwithjupyter.com

Cover design by Federica Dias (www.behance.net/federicadias)
Editing and proofreading by John Batson

www.learnpythonwithjupyter.com

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 3. Introduction to the for loop			
Chapter	Syntax	Computational thinking	In more depth
8. My friends' favorite dishes *p. 61*	■ `for` loop ■ Built-in functions `range()` and `str()`	■ For loop to repeat commands ■ For loop to automatically slice a list	*Dealing with IndexError and IndentationError* *p. 66*
9. At the zoo *p. 69*	■ Comparison operator `==` ■ Built-in function `len()` ■ # for commands ■ Abbreviating `index` with `i`	■ Binary condition in command repetition ■ Code commenting	*Dealing with TypeError* *p. 73*
10. Where are my gloves? *p. 76*	■ Comparison operators `!=`, `>`, `>=`, `<`, `<=`	■ Searching an element in a list based on element length or position, by combining for loop and if/else construct ■ Using variables in place of hard-coded values	*Let's use keyboard shortcuts!* *p. 82*
11. Cleaning the mailing list *p. 85*	■ String methods `.lower()`, `.upper()`, `.title()`, `.capitalize()`	■ Changing list elements in a for loop with reassignment	*In what list am I changing the element?* *p. 88*
12. What a mess at the bookstore! *p. 91*	■ Special character `"\n"`	■ Creating lists in a for loop ■ String slicing ■ Multiple consecutive slicing	*Append or concatenate. Don't assign!* *p. 96*
Part 4. Numbers and algorithms			
Chapter	Syntax	Computational thinking	In more depth
13. Implementing a calculator *p. 101*	■ Arithmetic operators ■ Built-in functions `int()`, `float()`, `type()` ■ Keyword `elif`	■ Number variables as strings, integers, or floats ■ Testing multiple variable values using `elif` ■ Combining code in a code unit	*Solving arithmetic expressions* *p. 108*
14. Playing with numbers *p. 110*	(no new syntax)	■ Changing numbers based on conditions ■ Separating numbers based on conditions ■ Finding the maximum in a list of numbers	*Don't name variables with reserved words!* *p. 113*
15. Fortune cookies *p. 116*	■ Keyword `import` ■ `random` module functions `.randing(a,b)` and `.choice(list)`	■ Module as a unit containing specific functions ■ Importing a module ■ Randomness in coding	*What if I don't use the index in a for loop?* *p. 119*
16. Rock paper scissors *p. 121*	(no new syntax)	■ Testing, debugging, parallelism, divide and conquer, algorithm	*Why do we say Debugging, Divide and conquer, and Algorithms?* *p. 127*

31. Factorials *p. 271*	(no new syntax)	■ Iterative vs recursive functions ■ Recursive thinking ■ Base case and recursive case	*When do we use recursive functions?* *p. 275*

About this book

What will I learn in this book? 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 this book different? The topic progression in this book is designed according to computational thinking development while focusing on syntax and strategies, rather than listing disconnected language characteristics with isolated examples.

Is this book for me? If you have never coded before, if you are following online courses or videos but feel you can't quite grasp them, 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 Python/Jupyter environment from another programming language, or if you are a teacher looking for material, this book can be for you.

How is this book structured? The book is divided in 11 parts. The first part introduces the computational environment—that is, the Jupyter/Python environment. The following ten parts cover computational thinking and Python syntax. Each part contains two to five chapters, for a total of thirty-eight chapters.

How are chapters structured? Each chapter starts with one or more coding examples embedded in narrative and enriched with detailed explanations. In addition, each contains several theoretical and coding exercises. And they all finish with a recap to summarize the chapter's main concepts, and a *In more depth* section, with coding strategies or curiosities.

Why is code embedded in narratives? Stories provide context and allow long-term memorization. They are extensively used in learning foreign languages. And, in many respects, a programming language *is* a foreign language.

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 defined standard for code pronunciation. The pronunciation proposed in this book is the optimized result of hours of one-on-one interaction with students of various mother tongues.

What kinds of exercises are in the book? In this book, you will find both theory exercises and coding exercises. Theory exercises are meant to strengthen code comprehension and syntax precision. Coding exercises are meant to make you practice and thus learn by doing.

What is on the website? On `www.learnpythonwithjupyter.com`, you can find Jupyter Notebooks associated with each chapter, so you can test and experiment while learning. You can also find a community, with solutions to both theory and coding exercises. You can ask questions and propose alternative solutions, to deepen your knowledge.

How do I use this book? Start with the first part, *Getting ready*, to install and learn the computational environment. Then, proceed with the chapters. For each chapter, download the corresponding notebook at `www.learnpythonwithjupyter.com`. Make sure you understand the syntax, play with the

code, and do the theoretical exercises. Read the recap and the *In more depth* sections, which will give you useful hints. Finally, do the coding exercises and compare your solutions with the ones you find in the community portal. Obviously, looking at a solution before completing an exercise weakens your chance of learning. If you do not understand questions or solutions, ask in the community portal. Take your time to solve each exercise. Missing the understanding of one chapter might compromise your understanding of the chapters that follow it.

How is the language used in this book? The language is colloquial and simple—but precise. There are clear definitions and careful explanations. I directly talk to *you*, but I use *we* when explaining syntax. We are in this together! Also, I use the first person when I want to share some hints I learned along the way.

INTRODUCTION

In this part, we will briefly 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 coding?

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, made of *concise* and *logical* instructions. Practically speaking, we have to learn at least three things: a coding environment, language syntax, and computational thinking. Let's see what these are!

A **coding environment** is a **program where we can write and execute code**. There are several environments to code in Python. In this book, we will use the Jupyter/Python environment, which since its release in 2015 has become used increasingly both in industry and academia (Figure 1.1). It allows integrating code with narrative, and it is ideal for creating reports, draft code, and learning to code. Other very common coding environments are the integrated development environments (IDEs). For Python, popular IDEs are PyCharm, Visual Studio, and Spyder (Figure 1.2). IDEs typically embed various components, such as a script editor, a variable environment panel, and a console wherein code is tested and executed. In Chapter 32, you will get familiar with one of them, Spyder, which is commonly used for scientific coding. And finally, the most basic environment is the 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 1.3).

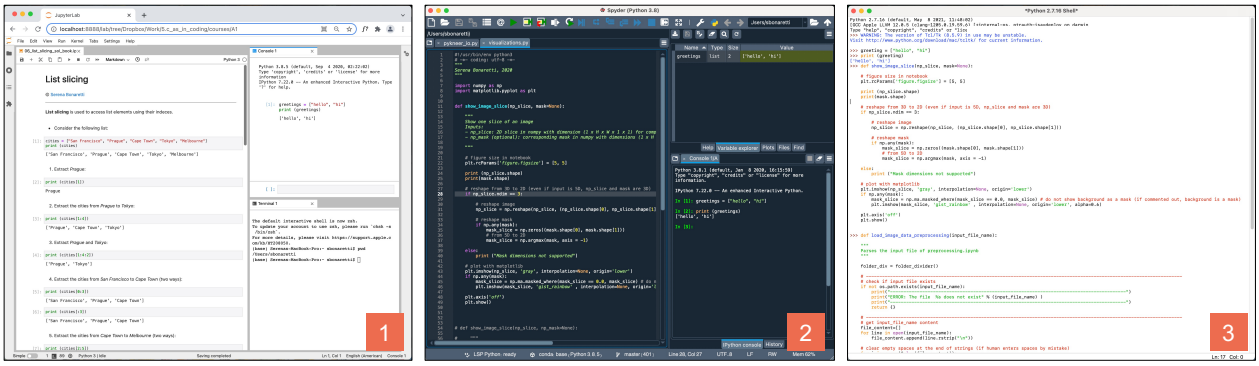


Figure 1. Three IDEs to code in Python: (1) the Jupyter environment, (2) Spyder, and (3) the Python IDLE.

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 have to know data types and operators, as well as how to arrange them in `if/else` constructs and loops to create functions and classes. In Table 1, you can see a schematic summary of elements and syntax 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 natural language syntax.

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 you will develop in this book are:

- *Creating algorithms*, which means conceiving and implementing a series of sequential instructions to solve a problem
- *Divide and conquer*, which consists of decomposing problems in sub-problems, and then combining the sub-problem solutions to obtain the main problem solution
- *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

As is the case for any subject, developing a way of thinking comes with studying and exercising. Thus, thinking computationally comes with learning syntax and practicing coding. We will start building these abilities in Chapter 1. In the next part, *Getting ready*, you will download, install, and learn how to use the Jupyter/Python environment.

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**, which is a web-based environment in which we can open, organize, and work on files of various types. In JupyterLab, there is **Jupyter Notebook**, which is 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 install the Jupyter/Python environment and see how it works!

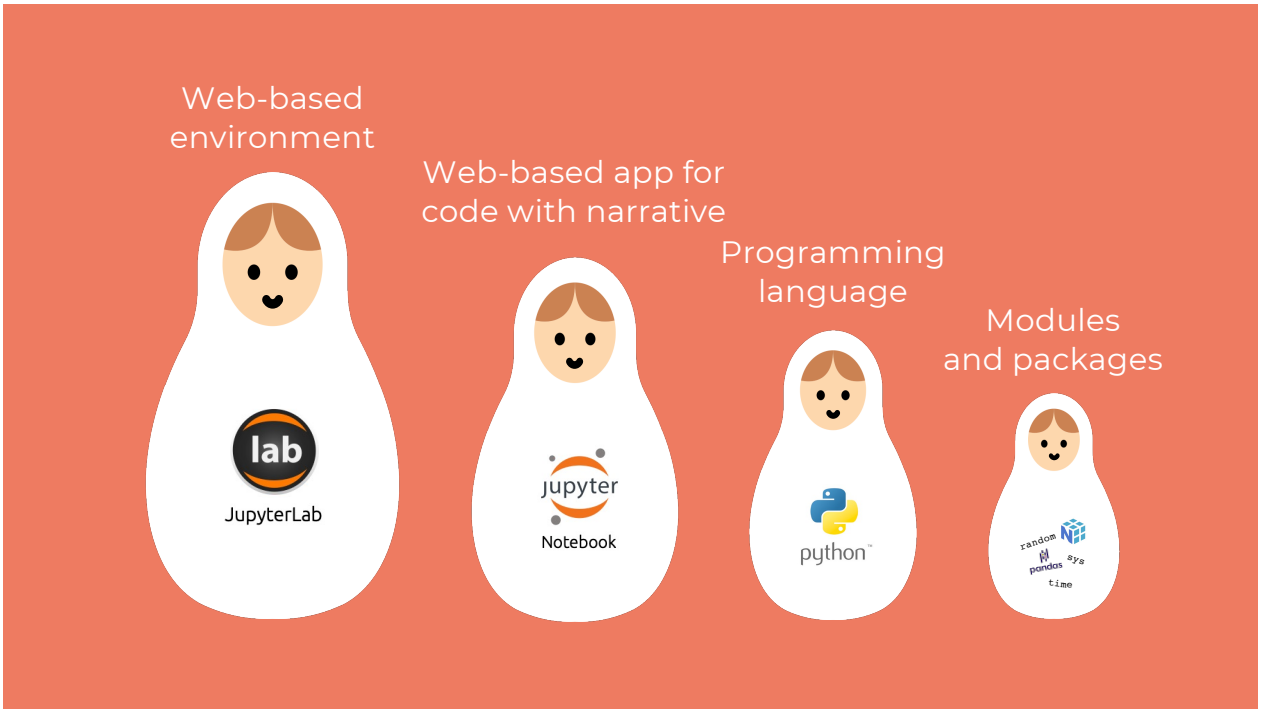


Figure 2. The Jupyter/Python environment represented as a Russian doll, where each component is included in the previous one.

Installing the Jupyter/Python environment

You can install JupyterLab, Jupyter Notebook, Python, and its scientific packages all at once through Anaconda, a commonly used distribution for scientific computing. Go to the Anaconda website, `https://www.anaconda.com/products/individual`, and click *download*. It might take a few minutes. 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 too. When Anaconda is installed, open the Anaconda Navigator by double-clicking its icon, which looks like the one in Figure 3, box 1. Once opened, you will see all the software contained in Anaconda, including JupyterLab (Figure 3, box 2), Jupyter Notebook (Figure 3, box 3), and Spyder (Figure 3, box 4). In this book, we will code in Python using JupyterLab as a working environment. So let's learn how

to use it!

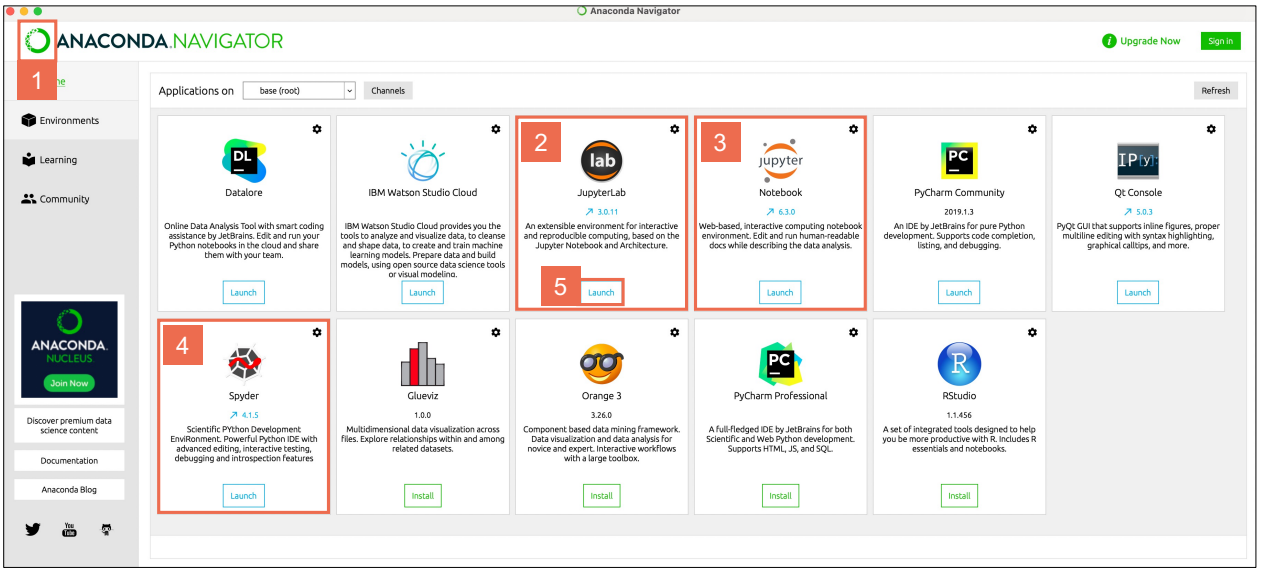


Figure 3. Anaconda interface: (1) icon, (2) JupyterLab, (3) Jupyter Notebook, (4) Spyder, and (5) JupyterLab launch button.

JupyterLab

JupyterLab is an environment where we can code in an organized and efficient way. Open it by clicking the *Launch* button in the JupyterLab tile in Anaconda (Figure 3, box 5). You will see something similar to Figure 4. Below are the most relevant features of JupyterLab and some suggestions on how best to use it.

- *JupyterLab is a web-based environment.* When you launch JupyterLab, the first thing you'll notice is that it starts in the browser. However, its address contains *localhost* (Figure 4, box 1), which means that you are actually working *locally*, that is, on your computer. In other words, you do *not* need to be connected to the internet to use JupyterLab.
- *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 go ahead and start exploring them! For now, just notice that when clicking some top bar buttons (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 open in the next section. Finally, a fun feature of JupyterLab is that you can set a dark theme. If you want that, go to *Settings*, then *JupyterLab themes*, and click on *JupyterLab Dark*.
- *Browsing and opening files.* On the left side of JupyterLab, you can find a panel with some vertical tabs (Figure 4, box 3). The first tab contains an icon representing a folder, and, for now, we will focus only on this one. The folder tab opens a panel on its right, which contains a few features. The first is a top bar (Figure 4, box 4), containing a symbol, +, which allows us to start a launcher (Figure 4, box 7); an icon representing a folder containing a +, to create a new folder; a vertical arrow pointing up, to upload a new file; and a circular arrow, to refresh the content of the current directory—in coding,

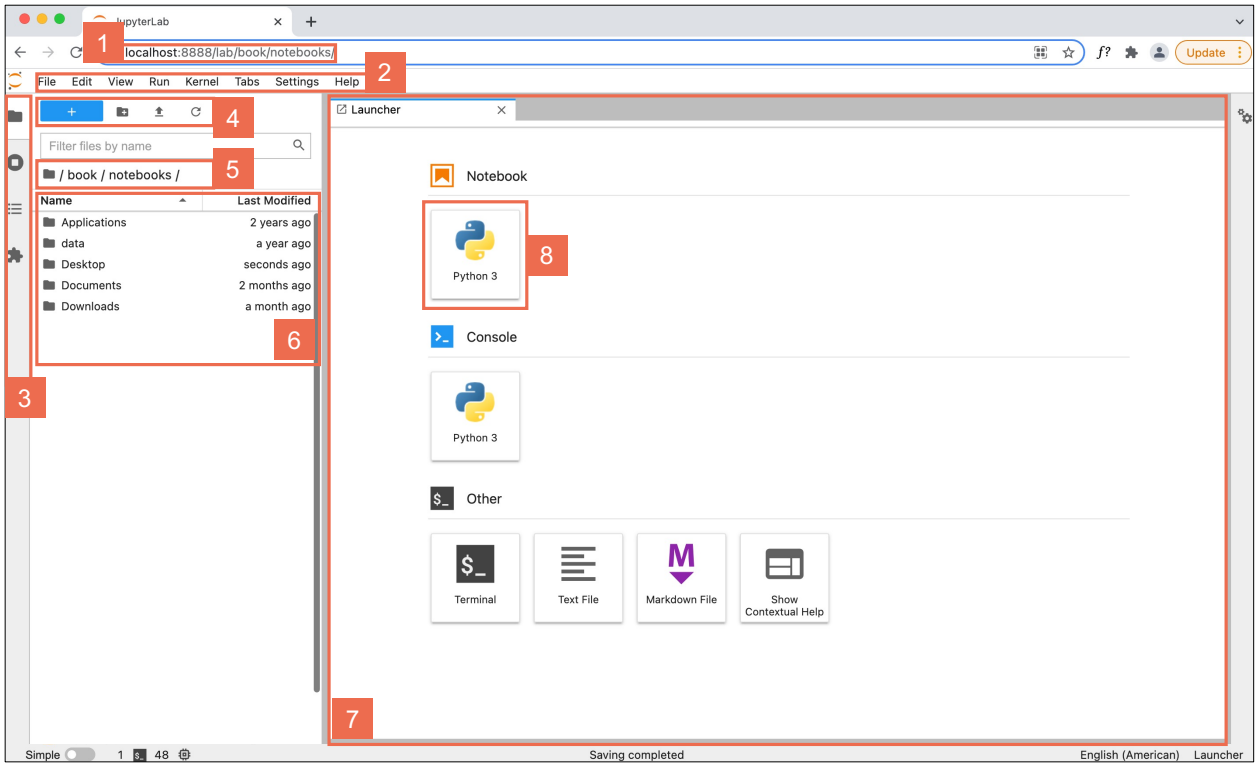


Figure 4. JupyterLab interface, containing: (1) local URL, (2) top bar, (3) lateral tabs, (4) folder browser top bar, (5) folder browser, (6) folder content, (7) launcher, and (8) Jupyter Notebook launch button.

we often say **directory** instead of *folder*. Right below, there is a box to search for files. Then, there is the path of the **working directory** (Figure 4, box 5)—that is, the folder where we are currently opening and saving files. And below, there is a list of the directory content (Figure 4, box 6). In JupyterLab, you can open an *existing* file only from this panel, and not by double-clicking the file in your computer folder. Therefore, you need to know how to navigate folders from JupyterLab. To go back to a previous folder, click on a folder name in Figure 4, box 5 (for example, to go back to the previous folder in this screenshot, you would click on *book*). To go into a sub-folder—a folder in the current folder—just double-click on the sub-folder listed in the folder panel (Figure 4, box 6). Last thing: when clicking on the folder icon (Figure 4, box 3), the whole file browser panel toggles out, meaning it disappears. When re-clicking, the whole panel toggles back in, so it reappears. Toggling out can be convenient if you have a small screen.

- *Launching tools.* The launcher is the place 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 on *File*, then *New*, and then selecting the file type you want. 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 Notebook opens in the launcher area (Figure 5, box 2), and it is visible as *Untitled.ipynb* in the browser panel (Figure 5, box 1). Notebooks have the extension *.ipynb*, which stands for interactive python notebook. To give the Notebook an appropriate name, right-click on *Untitled.ipynb* in the browser panel (Figure 5, box 1). Then, click *Rename*, and change it to 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 actions, such as delete, cut, copy, duplicate, and more.

Let's now focus on a Notebook content. 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 or 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! This might sound redundant, but it is conceived to help coders with different habits—some prefer using keyboard commands, others prefer clicking on the screen—conveniently perform the cell operations they need. If there are too many options for you, then just choose one way and stick to that! Below are some useful cell operations and some of the possible ways to perform them.

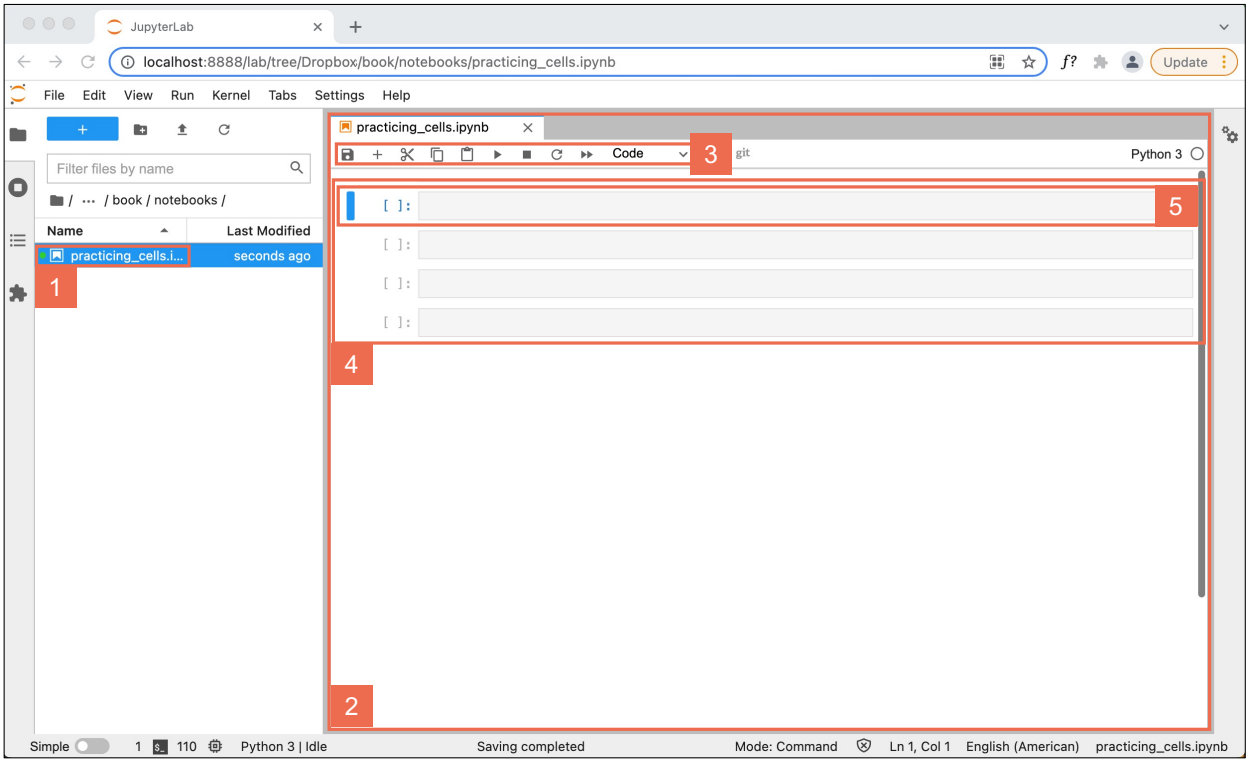


Figure 5. A Jupyter Notebook opened in JupyterLab. (1) Notebook in the folder browser, (2) Jupyter Notebook, (3) Jupyter Notebook top bar, (4) cells, (5) currently active cell.

- *Creating a cell*: To create a new 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 to copy, and then item 5 to paste (Figure 6).
- *Undoing or redoing cell operations*: To undo a cell operation (for example, if you have deleted a cell by mistake), press Z, or in JupyterLab top bar (Figure 4, box 2), go to *Edit*, and then *Undo cell operation*. Similarly, to redo a cell operation, simultaneously press *shift* and Z, or in JupyterLab top bar, go to *Edit* and then *Redo cell operation*.
- *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. When you reach the position you want to move the cell to, release. As an alternative, you can 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—you'll come to realize this starting in Chapter 1. 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, etc. by going to the JupyterLab top bar (Figure 4, box 2), and then see the options in *Edit*, or by right-clicking *in* a cell and browsing the options that appear. Just 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.

What about the remaining buttons in Figure 6? The first button representing a floppy disk—yes, once upon a time we saved data on floppy disks!—is to save the Notebook. The buttons 6 to 9 are used to execute code, and you will learn how to use them in Chapter 1 (button 6) and Chapter 7 (buttons 7 to 9).

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*

session in Chapter 22. And last but not least, cells can contain **code**. The remainder of the book will be about that! So, it's time to start coding, but before doing that, one last bit: you need to download the Jupyter Notebooks associated with this book.

Downloading 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 includes the corresponding chapter number. Each Notebook contains the examples discussed in the text so that you can practice and understand while reading. Download the Notebooks at `www.learnpythonwithjupyter.com`. I highly recommend that you save the Notebooks in a *new folder*—not in the *Download* folder—so that you don't mix them up with other files you download for other purposes. If you feel like going a step further, I really recommend that you create this folder in a *cloud service*, so that you do not lose your files in case your computer breaks or has issues (yes, computers are machines and they break!). As for cloud services, you can use Google Drive (`https://www.google.com/drive`), Dropbox (`https://www.dropbox.com`), or any others that you prefer. Using these tools is very easy. Download the program that installs the system on your computer. After the installation, you will see a new folder. Just create the folder that is going to contain the Notebooks in the newly created cloud folder, and all your files will always be automatically synchronized and saved.

Finally, in each chapter of the book, you will find coding exercises. I recommend that you create a separate folder called *Exercises*, or something similar, and inside this folder, create a Jupyter Notebook for the exercises of each book chapter. Creating Notebooks yourself will strengthen your organizational skills and will allow you to become even more familiar with the Jupyter/Python environment.

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

PART 1
CREATING THE BASICS

It's time to start coding! In this part, you will learn the basic elements that we will use throughout the whole book. You will learn about strings — that is, a data type that contains text — and the concatenation operation, used to combine strings. You will also learn how to ask questions and how to print out information. And most importantly, you will learn what a variable is. Let's get started!

1. Text, questions, and art

Strings, input(), and print()

Programming languages are written languages, and the core of written communication is *text*. How is *text* represented in Python? How can we ask a question to a person? And how can we provide information to a person? To answer these questions, let's open Jupyter Notebook 1 and start!

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, we see the code as it is in Jupyter Notebook 1. On the right side, we see how to pronounce the code. Let's 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 let's 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*. As we can see, a string is just some text in between quotes. 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. Quotes that start a string are called **opening** quotes, whereas quotes that end a string are called **closing** quotes. When writing a string in Python, we can use either double or single quotes; we just have to make sure we do not mix them up. In other words, if we start writing a string with an opening *double* quote, we must finish the string with a closing *double* quote. Similarly, if we start writing a string with an opening *single* quote, we must finish the string with a closing *single* quote. *Strings* are a Python **data type**, which means that they are one of the core parts of the Python language (see Table 1 at page 4). 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 written on any key, *enter* is the key on the right side of the keyboard depicting an angled arrow). As an alternative, you can click the *start* button in the Jupyter Notebook top bar (Figure 6, icon 6, at page 9).

This is how the first cell looks when we run 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   Everything you write between quotes
       string'                                     is a string
     'Everything you write between quotes is a string'
```

Two things occurred again. First, 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. Second, we can 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?")                    input what's your name?
     What's your name? |                |
```

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

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

And now press *return* or *enter* on the keyboard. You will see the following (you will see your name, of course!):

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

A few key things have happened here! First, 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 does not get executed, and in addition, we will not be able to run the following cells. Now, let's look at the code. We know that `"What's your name?"` is a string, because it is text in between quotes and it is colored red. What about **`input()`**? `input()` allows us **to ask a question to a user**. In Jupyter Notebook, `input()` creates a *text box* (a white rectangle) where we can insert some text. `input()` performs a specific task and is called a *built-in function*.

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, built-in functions are always followed by **parentheses ()**. In this book, instead of *parentheses*, we will call them **round brackets**, to differentiate from other types of brackets that we will encounter in the chapters that follow. In between the round brackets, we often write an **argument**, which for `input()` is a string containing the question we want to ask. *Built-in functions* are very useful, as they contain code written by the creators of a programming language to facilitate ease-of-use when coding.

Let's run the next cell:

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

Similarly to before, now enter your country of origin in the text box (I will type mine!):

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

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?")                    input where are you from?
     Where are you from? Italy
     'Italy'
```

What happened here is similar to the previous cell. Let's summarize it: the number on the left of the cell turned to 4 because this is the fourth cell we ran. The *built-in function* `input()` created a text box in Jupyter Notebook in which we could answer the question contained in the string we gave as an argument. Too concise? Let's try again: when we run the cell, the *built-in function* `input()` shows us the question, which we put in between the round brackets as a string, and it 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.

At this point we can ask ourselves: where do 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 it! For example, this is the case when we write our names to open a new account, 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 programmer and user! When writing code, we wear the *programmer hat*: we create code to perform a task, design code structure, and define user messages. When testing code, we wear the *user hat*: we check whether the code does what 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 a piece of information? We use the *built-in function* `print()`! There are several ways to learn about `print()`, and the following one is indeed a lot of fun. It involves a type of digital art called ASCII art, by which images can be created using the symbols on a keyboard. Let's have a look at the following cell:

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

What are we going to print to the screen? The answer is straightforward, but before running the cell, let's quickly 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 finally 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 a new built-in function: `print()`. **`print()` 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 that of the previous lines. If we delete the `print()` function from the code in cell 5, it will display only the very last *string*:

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

There are a few more things to point out by observing the code in cell 5. In a Jupyter Notebook cell, we can write several lines of code. The lines will get executed **sequentially**. In other words, when we run a cell, Python first executes line number 1, then line number 2, and so on, until the last line of the cell is reached. In addition, in a string, **spaces matter**. Spaces are *characters*, so a space is an element of a *string* and it takes its own place. However, spaces do not matter between code elements. For example, the two lines below are equivalent:

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

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,

```
[]:
1 print ("/\_/\  ")
2 print (">^.^<  ")
3 print (" / \   ")
4 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 more confusing. Instead, when we align quotes, brackets, and other symbols — as you will see in the following chapters — we create code that is more **readable** and **less prone to errors**. We will also talk quite a bit about tricks to minimize the amount of possible errors that we might introduce in code.

One more question before the recap: where do we see the 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 in 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 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.

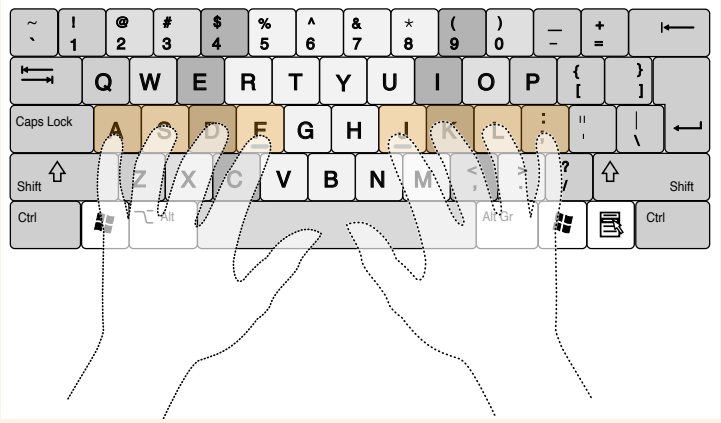


Figure 1.1. Starting finger position on a keyboard[a].

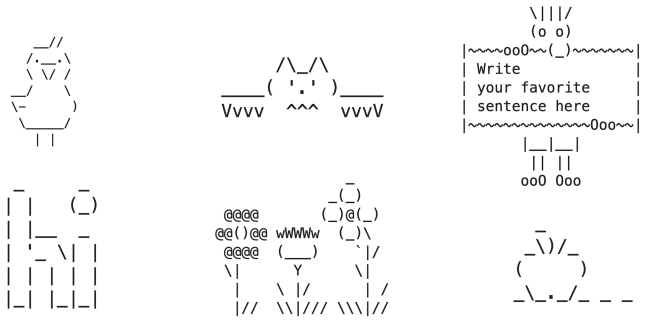
There are plenty of websites to learn touch typing in a fun way, such as `www.typing.com` and `www.typingclub.com`. They are free, and creating an account is not compulsory. They provide gradual exercises starting from typing single letters, to syllables, to words, up to whole sentences. Give it a try?

[a] Modified from a Figure by Cy21.

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 pages 8 and 9.

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. Events and favorites

Variables, assignment, and string concatenation

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

1. Organizing an event

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

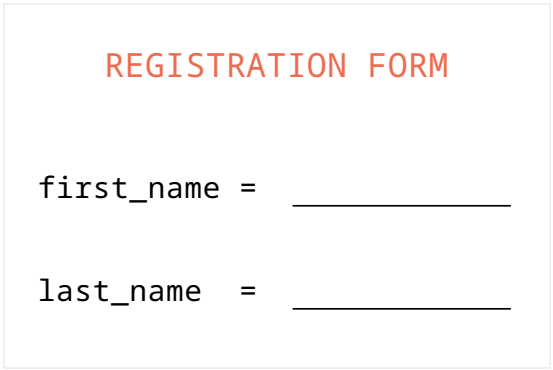


Figure 2.1. Registration form for the event participants.

- The first participant comes in and you fill out the form:

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

- Then you print out what you entered in the registration form:

```
[]:  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 = "Fernando"` assigns the string `"Fernando"` to the variable `first_name` T F
2. The command `print(first_name)` will print out `Fernando` T F
3. The command `print(last_name)` will print out `last_name` T F

Computational thinking and syntax

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

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

At line 1 we create a **variable** called `first_name`. To the variable `first_name` we **assign** the string `"Fernando"`, which is the **value**. Similarly, at line 2 we create a variable called `last_name`, to which we assign the string `"Pérez"` as a value. In general, we can assign any value to a variable. For example, we can register our second 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 (`"Fernando"` or `"Guido"`, `"Pérez"` or `"van Rossum"`). We can define variables as follows:

A **variable** is a label assigned 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, which 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
      Fernando
      Pérez
```

As you might expect, at line 1 we print to the screen the *value* assigned to the variable `first_name`, which is `Fernando`. At line 2 we print the value assigned to the variable `last_name`, which is `Pérez`. Who is Fernando Pérez? The creator of Jupyter Notebook! And Guido van Rossum? The creator of Python!

2. Favorites

Time to put together what we have learned so far! Let's read the following code:

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

```
[]:   1  favorite_food = input ("What's your      favorite_food is assigned input what's
         favorite food?")                         your favorite food?
```

```
[]:
1  print ("Hi! My name is " + name)

2  print ("My favorite food is " +
   favorite_food)
2  print (name + "'s favorite food is " +
   favorite_food)
```

```
print Hi! My name is concatenated with
name
print My favorite food is concatenated
with favorite_food
print name concatenated with 's favorite
food is concatenated with favorite_food
```

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 *What's your favorite food?* is asked before the question *What's your name?* T F
3. If the answer to the first question is *Terry* and the answer to the second question is *mango*, then the third print will show *Terry's favorite food is pizza* 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? Serena
```

```
name is assigned input what's your name?
```

The name we enter in the text box will be assigned to the variable `name`.

Let's run the second cell:

```
[4]:
1  favorite_food = input ("What's your
   favorite food?")
What's your favorite food? pasta
```

```
favorite_food is assigned input what's
your favorite food?
```

Similarly to the above example, what we enter in the text box will be assigned to the variable `favorite_food`.

Let's now run the last cell of this Notebook. What do we expect the prints to be?

```
[5]:
1  print ("Hi! My name is " + name)

2  print ("My favorite food is " +
   favorite_food)
2  print (name + "'s favorite food is " +
   favorite_food)
Hi! My name is Serena
My favorite food is pasta
Serena's favorite food is pasta
```

```
print Hi! My name is concatenated with
name
print My favorite food is concatenated
with favorite_food
print name concatenated with 's favorite
food is concatenated with favorite_food
```

At line 1, we print out the union of the string `"Hi! My name is "` and the value assigned to the variable `name`. When dealing with *strings*, the symbol + is called a **concatenation symbol**, not *plus*! Concatenating simply means *chaining together*. + allows us to **merge strings**, and we can pronounce it as *concatenated with*.

We have now learned the very basics on which we will build our coding skills and knowledge. Now let's take just a few minutes to complete the following exercise, which will help us summarize clearly the syntax we 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. `"My favorite food is"` 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 symbol** 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. It's important to learn how to read error messages so that we can fix errors quickly and keep coding. There are different kinds of errors, and we'll learn how to fix them over the course of the book. This is an example of an error:

```
---------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-6-a0c307bd3f14> in <module>
----> 1 print ("Hi! My name is " + ame )
      2 print ("My favorite food is " +  favorite_food )
      3 print ( name + "'s favorite food is " + favorite_food )
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.

In this case we are dealing with a **Name error**. The last line of the message says: `NameError: name 'ame' is not defined`. This is a very common error message. It means that there is not variable `'ame'` in your code. This error message usually pops up in two cases: when we misspell a variable name, or when we have not run a previous Jupyter Notebook cell containing the initialization (or creation) of the variable. In this example we have misspelled the variable `'name'`. This variable is present at lines 1 and 3. Which line should we look at? 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 can correct the typo, rerun the cell, and quickly move on with coding! Another very common error message is the following:

```
File "<ipython-input-1-daed5bd3b17e>" , line 1
    print ("Hi! My name is " name)
                            ^
SyntaxError: invalid syntax
```

In this case we have made a **syntax error**. The last line of the message says: `SyntaxError: invalid syntax`, which means that we have forgotten some symbol or punctuation. Where is the error? For syntax errors, we look at two lines in the message: at the end of the very first line, we see that we made the error at line 1; after the line of code, we see a *hat* symbol ^ that shows us the part of the command where there is something missing.

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 out.
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 out. Then, pick a book of your choice, create the two variables again, assign the corresponding values, and print them out.
3. *Where are you from?* Ask a person what country they come from and where they live. Then print out three sentences like in cell 5 of the code in this chapter.
4. *What's your favorite song?* Ask a person their favorite song and favorite singer. Then print out three sentences like in cell 5 of the code in this chapter.

PART 2

INTRODUCTION TO LISTS AND IF/ELSE CONSTRUCTS

In this part, you will learn about lists, which are simply lists 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 if/else constructs, which allow for executing 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 conditions? 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 block 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 summarized, 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 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 out 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 is made up of three components: (1) the **keyword `if`**, colored *bold green* in Jupyter Notebook, (2) the condition itself, and (3) the punctuation mark colon **:**
- **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, this is colored bold green, like keywords. In general, make sure not to confuse keywords, in bold green, with built-in functions, in fainter green.

Finally, 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. Under an `if` or an `else` condition, we can write as many commands as we want, 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 by using the example in the first row 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 contain 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 name**. 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?")
```

```
answer 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 consistently. 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 out 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 out 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 out the list to check whether we added the element correctly (line 7).

How do 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, at the end 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.

Finally, 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 out 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.

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 out a message saying that `item_to_remove` is not in the shopping list (line 6) and print out the shopping list for final check (line 7).

Complete the table

In Python, we use a lot of punctuation marks. Sum up what you have seen so far by completing the following table, using the example in row 1.

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 (lines 2–4). Let's rewrite it 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`. Therefore, 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)
```

```
today's menu is assigned burger, salad, coke
print today's 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 today's menu dot index of salad
print side dish index
```

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

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

```
today's menu dot pop side_dish_index
print today's menu
```

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

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

```
today's menu dot insert at side dish index fries
print today's 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']
```

```
today's menu is assigned burger, salad, coke
print today's menu
```

We create a list called `todays_menu` containing three elements of type string—`"burger"`, `"salad"`, and `"coke"` (line 1)—and we print it out (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 today's menu dot index of salad
print side dish index
```

The method **`.index()`** looks for the element `"salad"` in the list `todays_menu` and tells 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 out (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.

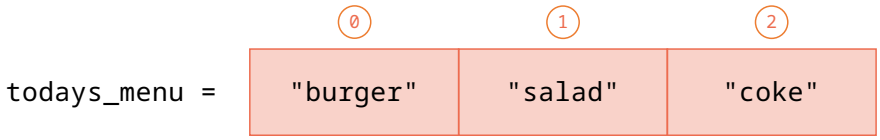


Figure 5.1. Representation of the list `todays_menu`: each square is a list element, and the number above is the corresponding index.

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']
```

```
today's menu dot pop side_dish_index
print today's 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']
```

```
today's menu dot insert at side dish
index fries
print today's 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.

Finally, when dealing with lists, we must always be aware of that **each element has a position**. 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 the list. On the other side, `.index()` gives us some information about the 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 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*; therefore, in this chapter, we will focus on this topic. 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 for 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 out (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:

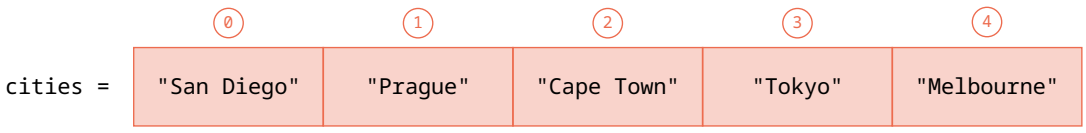


Figure 6.1. Representation of the list `cities`: each square is a list element, and the number above is the corresponding index.

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. 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, you might wonder: why didn't we write the step in the example 2, where we sliced the cities from `"Prague"` to `"Tokyo"`? 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 simply omit it:

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

`print cities from the beginning of the list to position 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`. And once more, we can simplify the three-s-rule! How? The **stop** coincides with the **last element of the list**, so we can just omit it:

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

`print cities from position two to the end of the list`

So far, we have seen the three-s rule applied in its entirety (example 3), and without start (example 4), stop (example 5), and step (example 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])                    print cities in positions from zero to five
                                                with a step of two
     ['San Diego', 'Cape Town', 'Melbourne']
```

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])                     print cities from the beginning to the end
                                                of the list with a step of two
      'San Diego', 'Cape Town', 'Melbourne']
```

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`:

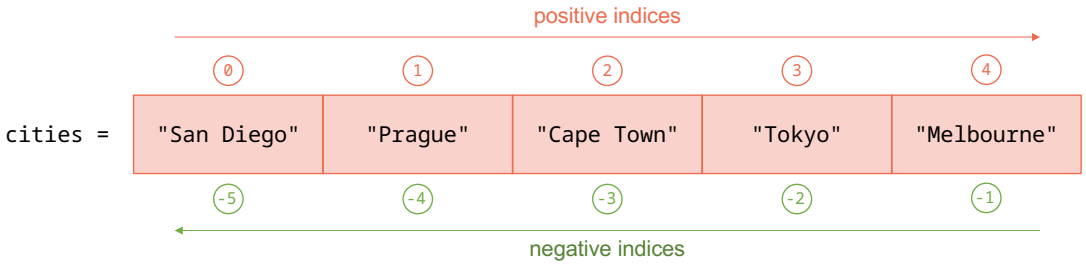


Figure 6.2. In a list, indices can be positive (from left to right) or negative (from right to left).

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])                       print cities in positions 4
      Melbourne
```

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])                    print cities in position minus one
      Melbourne
```

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])                 print cities in positions from minus four to
                                               minus one
      ['Prague', 'Cape Town', 'Tokyo']
```

This is in an alternative to example 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 examples. Slicing in reverse order is 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])                print cities in positions from three to zero
                                               with a step of minus one
      ['Tokyo', 'Cape Town', 'Prague']
```

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 out `"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 from the beginning of the list to the
end of the list with a step of minus 1
```

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.

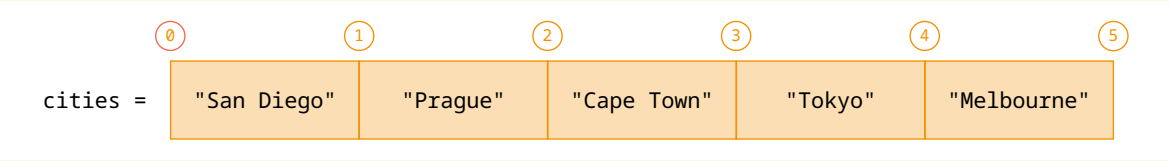


Figure 6.3. List representation where each element is in between indices.

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 *inverse* 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 them 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 out 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 from `www.learnpythonwithjupyter.com` 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**.

- Let's start with the following list:

```
[1]:  1  senses = ["eyes", "nose", "ears", "tongue",
          "skin"]                                        senses is assigned eyes, nose, ears,
                                                         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 out (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
                                                         of 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 3 to 1, and we do not need to create an extra variable—`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! 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!

- Let's start with 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 out (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 out 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. 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 examples!

- 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 from
zero to two concatenated with
venus concatenated with planets
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"`. So, 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. As you may have realized, this line is an alternative to the method `.insert()` (line 1). Finally, we print out the obtained list to check the correctness of the operation (line 2).

A nice 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. Thus, we obtain our modified train!

- 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 from
the beginning of the list to
position five concatenated with
Uranus, Saturn concatenated with
planets from position 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 (start omitted) 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 (stop omitted). As you'll probably notice, when we want to insert several consecutive elements in the middle of a list, we just **embed all the elements in a list** (line 1). Finally, we print out the modified list to check whether we added the new elements correctly (line 2).

Now a trick! We saw that the start of the first list segment and the stop of the second list segment are omitted. In addition, 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. Therefore, when adding new elements using slicing, we can **just count the stop of the first list segment**. That will coincide with the start of the second list segment. The remaining start and stop can be omitted!

An important note before continuing: in the past three examples, we started analyzing code from the right side of the assignment symbol. 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 symbol, 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!

- Consider 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 out (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]`. **`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()` (line 1). After removing the element, we print out 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 form 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 it becomes 6 because of the plus one rule (line 1). Then we print out 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 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: 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 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)
<ipython-input-13-ef0756c89224> in <module>
       1 del house
----> 2 print (house)
NameError: name 'house' is not defined
```

```
del house
print house
```

Finally, we want to delete the whole house! So we write the keyword `del` followed by the variable name `house` (line 1). This time, we get an error when we print out the list `house`. It's a *Name Error*, telling us 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

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 rerun the second cell with the corrected code. However, when dealing with multiple variables, or if we make several mistakes for a single variable, it is good practice to reset the kernel and start 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

   Then, 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)

   Then, 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)

   Then, 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. We will learn its syntax 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 string of 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 string of 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
2. The variable `index`
3. In the first loop, the variable `index`
4. The built-in function `range()` determines
5. The built-in function `range()` can take

a. a *start* and a *stop* as an argument
b. how many times commands are repeated
c. to repeat the indented commands
d. changes value at each loop
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)
```

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

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

We print out 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])
```

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

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 then write and run the following code:

```
[4]:  1  list(range (0,4))                    list of 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 out. 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. We could call the variable `index` any name—for example, *loop_id*, *iteration_number*. However, it is convention to call it `index`, so we will adopt it. 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 can use `index` **to automatically slice list elements one by one**. As you now know, `index` changes at every iteration, and it can be 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 summarize what the code at 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, `index` is assigned `0` (line 1), so we print `index: 0` (line 2), and then `friends` in position `index`—which is `0`—and thus `friend: Geetha` (line 3).
- In the second iteration, `index` is assigned `1` (line 1), so we print `index: 1` (line 2), and then `friends` in position `index`—which is `1`—and therefore `friend: Luca` (line 3).
- In the third iteration, `index` is assigned `2` (line 1), so we print `index: 2` (line 2), and then `friends` in position `index`—which is `2`—and therefore `friend: Daisy` (line 3).
- In the fourth iteration, `index` is assigned `3` (line 1), so we print `index: 3` (line 2), and then `friends` in position `index`—which is `3`—and therefore `friend: Juhan` (line 3).

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 = 0	index: 0	friend: friends[0] → Geetha
Second	index = 1	index: 1	friend: friends[1] → Luca
Third	index = 2	index: 2	friend: friends[2] → Daisy
Fourth	index = 3	index: 3	friend: friends[3] → 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 **lines of code 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 out 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
friend: sushi
index: 1
friend: burgers
index: 2
friend: tacos
index: 3
friend: pizza
```

```
for index in range from 0 to 4
print index: concatenated with
string of 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. However, at line 3 we print out the dish names one by one. Once again, let's go through the code one iteration at a time:

- In the first iteration, `index` is assigned `0` (line 1), so we print `index: 0` (line 2), and then we print `dishes` in position `index`—which is `0`—and thus `dish: sushi` (line 3)
- In the second iteration, `index` is assigned `1` (line 1), so we print `index: 1` (line 2), and then we print `dishes` in position `index`—which is `1`—and thus `burgers` (line 3)
- In the third iteration, `index` is assigned `2` (line 1), so we print `index: 2` (line 2), and then we print `dishes` in position `index`—which is `2`—and thus `tacos` (line 3)
- In the fourth iteration, `index` is assigned `3` (line 1), so we print `index: 3` (line 2), and then we print `dishes` in position `index`—which is `3`—and thus `pizza` (line 3).

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])
```

```
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
```

```
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
```

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 assign them to the variable `index`, one by one at each iteration (line 1). Just one note: beyond the start and the stop, the built-in function `range()` can also take a step as an argument, like so:

```
[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 these 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.

Finally, the body of the for loop is constituted of one line of code, where we print out 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.

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 is assigned _____ .
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 built-in Python function that creates a sequence of integers spanning from the start (included) to the stop (excluded)
- `str()` is a built-in Python 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 string of 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)
<ipython-input-13-ef0756c89224> in <module>
       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`.

IndentationError. 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 string of index
```

```
File"/var/ipykernel_54813/8597.py" , 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, an if or else condition, a while loop header (Chapter 17), a function definition (Chapter 28), or a class definition (Chapter 35).

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 out all the countries as a list and all the countries one by one. Then, print out all the cities as a list and all the cities one by one. Finally, print out 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 out all the animals as a list and all the animals one by one. Then, print out all the continents as a list and all the continents one by one. Finally, print out 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, 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 out 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 of animals
print beginning of loop
print each element and its position
print the element in position concatenated
with string of 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 out 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          # just print out what you saw
     9          print ("I saw a " + animals[i])
```

```
for each position in the list
for i in range from zero to len of animals
if the current animal is what you really
wanted to see
if animals in position i equals wanted to
see
print out 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:
just print out 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 out (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 of animals
print beginning of loop
print each element and its position
print the element in position concatenated
with string of i concatenated with is
concatenated with animals in position i
```

We run the for loop three times, and each time we print out 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, then the stop would be `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 (`---`) in this example. Alternatives can be arrows (`-->`), tildes (`~~~`), or any other character on the keyboard. At line 5, we print out 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 in a different position `i` at each iteration—that is, `"giraffe"`, `"penguin"`, or `"dolphin"`.

Finally, 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, or **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 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: **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 out 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 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 out 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          # just print out that you saw it
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 of animals
if the current animal is what you really
wanted to see
if animals in position i equals wanted to
see
print out 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:
just print out that you saw it
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. Then, 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 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 step-by-step:

- In the first loop: at line 2, `i` is assigned `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"`
- In the second loop: at line 2, `i` is assigned `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!"`
- Finally, in the third loop: at line 2, `i` is assigned `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 state-

ment 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 they are 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]:
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 " +
       i + " is " + animals[i])
```

```
for each position in
the list
for i in range from
zero to len of 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)
<ipython-input-5-db98c59ed681> in <module>
       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 in the text above, 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]:
1  planets = planets + "Jupyter"
2  print (planets)
```

```
planets is assigned
planets concatenated
with jupyter
print planets
```

```
---------------------------------------
TypeError       Traceback (most recent call last)
<ipython-input-5-db98c59ed681> in <module>
----> 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") with list`. We are trying to concatenate a string to a list. Where? The green arrow points to line 1. Around the concatenation symbol, 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 out 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 out the planets one by one. Then, create a variable for your next destination. Finally, print out all the planets, specifying if they are your next destination.
2. *Months.* Given the following list:

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

   Print out the names of winter months using a for loop. Then, print out the names of summer months using a for loop. Choose a month you like and assign it to a variable. Print out 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 out 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 next three chapters. 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 out (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 of accessories
print each element and its position
print The element concatenated
with accessories in position i
concatenated with is in position
concatenated with string of 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 (if you need to refresh your memory that `range()` creates a list of integers, see cell 4 on page 63). 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 with 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 of accessories
if the length of the element equals
six
if len of 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 string of i
concatenated with and it has six
characters
```

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

We want to find the list element composed of 6 characters. As mentioned above, 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, like you learned in Chapter 9 at cell 4. If the current element satisfies the condition, we print out 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. Note the comments on lines 1,3, and 5.

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

```
[3]:
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 of accessories
if the length of the element is less
than six
if len of accessories in position i
less than 6
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 string of 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 out 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:

[4]:
```
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 of accessories
if the length of the element is
greater than the threshold
if len of 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 string of i
concatenated with and it has more
than concatenated with string of
n of characters concatenated with
characters
```

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

In this example, we add 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 out 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 use 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 in both places. Instead, by using the variable `n_of_characters`, we change the value in just one place (line 2). It is **good practice to create variables** containing values instead of hard-coding within a block of code. Variables are usually written **at the beginning of a block of code** so that they are easy to find, especially when the code is composed of several lines.

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)):
```

```
defining the threshold
n of characters is assigned six
for each position in the list
for i in range len of accessories
```

```
5     # 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 the length of the element is not
equal to the threshold
if len of 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 string of i
concatenated with and it has a
number of characters different from
concatenated with string of 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 out 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:

[6]:

```
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 of 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 string of i
concatenated with which is less than
or equal to concatenated with string
of 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 assigned `0`, thus `accessories[i]` is `accessories[0]`, which is `"belt"`
- In the second loop, `i` is assigned `1`, thus `accessories[i]` is `accessories[1]`, which is `"hat"`
- In the third loop, `i` is assigned `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:

```
[6]:
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 of 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 string of i
concatenated with which is at least
concatenated with string of 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'll notice for `!=` (example 4), `<=` (example 5), and `>=` (example 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 practical 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 slows down the typing rhythm and interrupts the writing flow. 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.

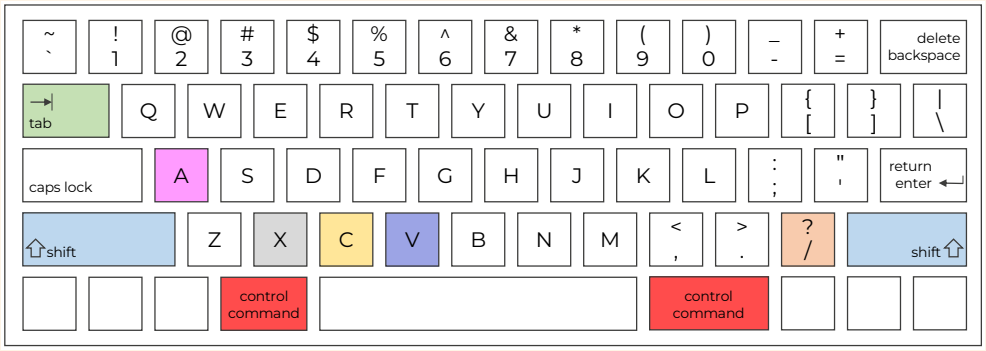


Figure 10.1. Example of keyboard. The colored keys are commonly used for shortcuts.

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* session 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 out (e.g., *Word search with 16 words*)
   b. Find words composed of 5 letters. More specifically, print out 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 out 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 out 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 word search game for this exercise solution on the community 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 out
   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 out 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 out 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 out 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[i]))
```

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

print -> loop: concatenated with
string of i

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

print the modified list
print Now the list is: concatenated
with string of emails in position i
```

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 of emails

print -> loop: concatenated with
string of i

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

print the modified list
print Now the list is: concatenated
with string of 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 0 is: Pablo.Hernandez@live.com
After the change, the element in position 0 is: pablo.hernandez@live.com
-> Loop: 2
Before the change, the element in position 0 is: LI.Min@hotmail.com
After the change, the element in position 0 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 3, 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, and the loop number favors checking what

happens at each specific iteration.

At line 5, 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 7, 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 page 32). How do we know that `.lower()` is a *string* 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
- **`.title()`** to change the first character of a string to uppercase and all the remaining characters to lowercase
- **`.capitalize()`** 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 changed element to itself**. In other words, we need to **overwrite the current element with its new version**. If we do not do that, then the list element will remain unchanged.

At line 9, we print out 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 5.

At line 10, we print out the new list. 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())`
2. `print(greeting.upper())`
3. `print(greeting.title())`
4. `print(greeting.capitalize())`

a. `'HELLO, HOW ARE YOU?'`
b. `'Hello, how are you?'`
c. `'hello, how are you?'`
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()`.

In what list am I changing the element?

Sometimes, we have to change a list element before adding it to an existing list. This can create confusion about *where* 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 the mountain sport to the following list, making sure the string is uppercase:

```
[2]: 1 mountain_sports = ["CLIMBING"]
```

```
mountain_sports is assigned
CLIMBING
```

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()

     2 mountain_sports.append(sports[1])

     3 print(sports)
     4 print(mountain_sports)
['diving', 'HIKING']
['CLIMBING', 'HIKING']
```

```
sports in position 1 is assigned
sports in position 1 dot upper
mountain_sports dot append sports
in position 1
print sports
print mountain_sports
```

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 out 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 1 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 out 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 1 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, like in a Russian doll (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 a temporary variable by directly substituting its content in a nested command. Finally, when we print out 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, so you 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 out 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 out 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 out 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 an empty list
shelf s is assigned an empty list

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

print out 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 of 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 of shelf s
concatenated with backslash n
```

```
27 # print out the final shelves                    print out the final shelves
28 print ("The authors on the shelf A are: " +      print The authors on the shelf A are:
   str(shelf_a)                                     concatenated with str of shelf a
29 print ("The authors on the shelf S are: " +      print The authors on the shelf S are:
   str(shelf_s)                                     concatenated with str of 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 special 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",          authors is assigned Alcott, Saint-Exupéry
     "Arendt", "Sepulveda", "Shakespeare"]            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     initialize the variables as empty lists
     2 shelf_a = []                                  shelf a is assigned an empty list
     3 shelf_s = []                                  shelf s is assigned an 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                  for each position in the list
6 for i in range (len(authors)):                   for i in range len of authors
7
8    # print out the current element               print out the current element
9    print ("The current author is: " +            print The current author is: concatenated
     authors[i])                                   with authors in position i
```

We create a for loop to browse all the elements in the list `authors` (line 6), and we print out 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 out (line 13). How do we get an author's initial? Let's focus on the right side of the assignment symbol—`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 obtaining a string, whereas with the second pair of square brackets `[0]`, we slice a string obtaining a 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 of 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 of 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 out 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.

At the end of the print commands at lines 19 and 24, you'll notice `"\n"`. What's that? It's a **special**

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 Chapter 9, cell 2). You will see more special 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 out the final shelves                    print out the final shelves
28 print ("The authors on the shelf A are: " +     print The authors on the shelf A are:
   str(shelf_a)                                    concatenated with str of shelf a
29 print ("The authors on the shelf S are: " +     print The authors on the shelf S are:
   str(shelf_s)                                    concatenated with str of shelf s
```

Above, we print out 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.

Finally, 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. 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, the code contains several comments and empty lines between blocks of code to improve readability.

Match the code

Let's summarize what we learned about for loops! 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])
```

2.
```
for i in range (len(hot_drinks)):
    if hot_drinks[i][0] == "c":
        print (hot_drinks[i])
```

3.
```
for i in range (len(hot_drinks)):
    if len(hot_drinks[i]) == 3:
        hot_drinks[i] = hot_drinks[i].upper()
print (hot_drinks)
```

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)
```

a.
```
['TEA', 'coffee',
'hot chocolate']
```

b.
```
tea
coffee
hot chocolate
```

c.
```
['coffee', 'hot chocolate']
```

d.
```
coffee
```

★. create list elements

♣. change list elements

■. print list elements one by one

▲. 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 special 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*. 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      # 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]
11         print ("The shelf A now
           "contains: " + str(shelf_a))
12 # print out the final shelves
13 print ("The authors on the shelf A are: "
   + str(shelf_a)
```

```
initialize the variable
shelf a is assigned an empty list
for each position in the list
for i in range len of 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 of shelf a
print out the final shelves
print The authors on the shelf A
are: concatenated with str of 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 and focus on the names starting with A:

- When `i` is `0` (line 4), `author_initial` is `"A"` (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 out *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"` (line 6); the if condition is true (line 8), we assign `authors[i]`—that is, `"Arendt"`—to `shelf_a` (line 10), and we print out: *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"`; thus, `shelf_a` remains a string.

At line 13, we print the final version of `shelf_a`, which is a string with value `"Arendt"`.

In conclusion, assigning a variable to a list (e.g., `shelf_a = authors[i]`) changes the type of the list itself 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]]`.

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"]`
   and you 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 (position 1)
   - 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 go!

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. However, 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 symbol 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 is different from the symbol used in paper-and-pencil computations, which can be the cross symbol x or the mid-line dot operator ·.

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]:  4  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]:  5  7 // 4                                  seven floor division four
      1
```

To execute a floor division, we use the operator `//`, composed of two forward slashes and pronounced *floor division*. 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 `1`, which is the closest lower integer to `1.75`. The word *floor* indicates that we round the result down, that is—using a metaphor–to the floor of a house.

7. Modulo:

```
[7]:  7  7 % 4                                   seven modulo four
      3
```

To calculate a modulus, we use the operator `%`, which is pronounced *modulo*. This operation calculates a **reminder** (or modulus), which 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 modulus (`4+3=7`). Note that modulo is the name of the operator, while modulus is the name of the operation and a synonym for remainder. The modulus operation is 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", which are multiplication (`*`) and exponentiation (`**`)
- 3 for the "division family", which are 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 modulus 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 out the variable:

```
[8]:  1  number = input("Insert a number:")          number is assigned input Insert a number:
      2  print (number)                               print number
      Insert a number: 9
      9
```

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 out 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)                              type number
      str
```

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 out, and check its type:

```
[10]: 1  number = int(number)                       number is assigned int of number
      2  print (number)                              print number
      3  type (number)                               type number
      9
      int
```

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 out, and check its type:

```
[11]: 1  number = float(number)                     number is assigned float of number
      2  print (number)                              print number
      3  type (number)                               type number
      9.0
      float
```

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 can 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 out, and check its type:

```
[12]: 1 number = str(number)
      2 print (number)
      3 type (number)
9.0
str
```

```
number is assigned str of 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.

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 them 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 find out 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 yourself 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!

- 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 of
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, transforming `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 out `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). Because an arithmetic operator is a symbol, we keep it as a string, and we print out 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 of 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 transform the user's choice into a float before assigning it to the variable `second_number` (line 1). Then, we print out 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
```

```
if operator is equal to plus
result is assigned first number plus
second number
elif operator is equal to minus
result is assigned first number minus
second number
elif operator is equal to times
result is assigned first number times
second number
elif operator is equal to to the power of
result is assigned first number to the
power of second number
```

```
9  elif operator == "/":                          elif operator is equal to divided by
10     result = first_number / second_number       result is assigned first number divided
                                                   by second number
11 elif operator == "//":                         elif operator is equal to floor division
12     result = first_number // second_number      result is assigned first number floor
                                                   division second number
13 elif operator == "%":                          elif operator is equal to modulo
14     result = first_number % second_number       result is assigned first number modulo
                                                   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. We start by considering addition (lines 1 and 2). In the if condition, we check if the variable `operator` from 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 difference between the two numbers entered by the user, and we assign the result to the variable `result` (line 4).

As you might have noticed, the condition at line 3 started with the keyword **`elif`**, which is an abbreviation for *else if*. We use `elif` when we check **several conditions on one single variable**, which is `operator` in this case. We continue the code with a similar structure for the remaining arithmetic operations (lines 5–14). When using an `if/elif/else` construct, make sure to **always test code under all conditions**. To do that in our example, re-enter the variables `first_number`, `operator`, and `second_number` for each condition and make sure that what gets printed is the one you expected. We conclude the list of conditions with an `else` (line 15), which prints out a warning in case the user did not enter a valid arithmetic operator (line 16). Finally, we print out 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.

- Finally, let's print out the result:

```
[17]: 1 print (str(first_number) + " " + operator      print str of first_number concatenated
        + " " + str(second_number) + " = " +          with space concatenated with operator
        str(result))                                  concatenated with space concatenated with
                                                      str of second_number concatenated with
                                                      equals concatenated with str of 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 it together our code to create a real calculator by merging all lines from the code above into one single cell. This will allow us to run only one cell (instead of multiple cells) when executing the code:

[18]:

```
# first input
first_number = float(input("Insert the first number:"))

# operator
operator = input("Insert an arithmetic operator:")

# second input
second_number = float(input("Insert the second number:"))

# computations
if operator == "+":
    result = first_number + second_number

elif operator == "-":
    result = first_number - second_number

elif operator == "*":
    result = first_number * second_number

elif operator == "**":
    result = first_number ** second_number

elif operator == "/":
    result = first_number / second_number

elif operator == "//":
    result = first_number // second_number

elif operator == "%":
    result = first_number % second_number

else:
    print ("You didn't enter an arithmetic operator")

# print the result
print (str(first_number) + " " + operator + " " + str(second_number) + " = " + str(result))
```

```
first input
first_number is assigned float of input Insert the first number:

operator
operator is assigned input Insert an arithmetic operator:

second input
second_number is assigned float of input Insert the second number:

computations
if operator is equal to plus
result is assigned first number plus second number
elif operator is equal to minus
result is assigned first number minus second number
elif operator is equal to times
result is assigned first number times second number
elif operator is equal to to the power of
result is assigned first number to the power of second number
elif operator is equal to divided by
result is assigned first number divided by second number
elif operator is equal to floor division
result is assigned first number floor division second number
elif operator is equal to modulo
result is assigned first number modulo second number
else
print You didn't enter an arithmetic operator

print the result
print str of first_number concatenated with space concatenated with operator concatenated with space concatenated with str of second_number concatenated with equals concatenated with str of result
```

When we merge code in one cell at the end of an implementation, we usually edit and clean it up 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, 6, 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 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

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  6 + 2 * 3                    six plus two times three
     12
```

First we execute the multiplication, followed by the addition. Thus, we first calculate `2 * 3`, which is 6, and then `6 + 6`, which is 12.

Here is another example:

```
[2]:  1  (6 + 2) * 3
      24
```

```
open round bracket six plus two close round bracket times three
```

First, we execute the operation between round brackets `(6 + 2)`, which is 8, and then the multiplication `8 * 3`, which is 24. Note that brackets can only be round in coding.

Let's code!

1. *Math competition.* You are holding a math competition where participants have to choose among three envelopes and solve the arithmetic operation 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 of numbers

if current number is greater than or equal to ten
if numbers in position i is 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 4), we subtract `1`, and we reassign the result to `numbers[i]` (line 6), 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 4), 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 4), so we jump to the `else` (line 10). There, 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 12).

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]:
3  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])
10
17 # check the final results
18 print (even)
19 print (odd)

[2,10,0]
[7,5,9]
```

```
initialize empty lists
even is assigned an empty list
odd is assigned an empty list

for each position in the list
for i in range len of numbers

if the current number is even
if numbers in position i modulo 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 13. 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, whereas odd numbers are not. Thus, we can use the **modulo**, one of the seven arithmetic operators we learned in the previous chapter. When divided by 2, even numbers have a modulus (or remainder) equal to 0, whereas odd numbers don't (the remainder is 1!). Therefore, if the remainder of the current list number (e.g., `numbers[i]`) divided by 2 is 0 (line 9), then 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]:
3  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 of numbers

if the current number is greater than the
current maximum
if numbers in position i is 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 1). 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 the value of `numbers[0]` (from the for loop) to itself (assigned to `maximum`). 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 can skip the `else`. Finally, we print out 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? Consider the following example. Let's say we initialize `maximum` with a small number like `-999993`. However, the current list could be `-999993`, such as `[-999998, -999996, -999994]`, so we won't be able to find the maximum of the list (i.e., `-999994`). 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 used 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. As you can see, the **variable name** is colored **green**, which means it is a **reserve word**—we know that `len()` is a Python built-in function, and that variable names are colored black (line 1). When we print the variable, we do not encounter any issue (line 2). However, if we want to calculate the length of a list in subsequent code, we get an error:

```
[2]:
1  numbers = [1, 2, 3]
2  len (numbers)
```

```
numbers is assigned one, two, three
len numbers
```

```
---------------------------------------
TypeError       Traceback (most recent call last)
<ipython-input-5-db98c59ed681> in <module>
       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; instead, 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 out 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 out 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 or in scientific simulations. Read the following example and try to understand it. You can play with 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(0, 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 zero len of 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.

Let's continue with the second cell:

```
[2]:
1  import random
```

```
import random
```

We use the *keyword* **`import`** to import the *module* **`random`**. What does this mean? As you know, Python contains basic built-in functions, such as `print()`, `input()`, `len()`, `range()`, etc. However, when we code, we often need tools for recurrent tasks, such as generating random numbers, browsing directories, computing statistics, etc. For this reason, 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 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(0, 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 zero len of 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 number between them *included*—that is, `a` and `b` can be the generated random number. 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 **parameterise** `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`, and we print it (line 3). Finally, we slice the list `fortune_cookies` in position `message_index` to extract a string containing that we assign 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                                  pick a message
      2  message = random.choice(fortune_cookies)         message is assigned random dot
                                                          choice fortune cookies
      3  print(message)                                   print message
The man on the top of the mountain did not fall there
```

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 out the current loop number (e.g., `print ("This is loop number " + str(i))`) or to automatically slice list elements (e.g., `print (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 (0,3):
5      # create a random number between 10
       and 20
6      random_number = random.randint(10,20)
7      # print the number
8      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 from zero
to 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
```

We use a for loop to generate and print three random numbers (lines 4–8). 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 for other Python coders.

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 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 dice.* 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, that if 42 is present more than once, it will appear only once in `unique`. If the number is already present in `unique`, print out 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 originating at least 2,000 years ago in China[1]. In this chapter, we will learn how to implement this game 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. _____ .

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 *rock*, *paper*, 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
8  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 `possibilities`. Finally, we save the pick in the variable `computer_pick` (line 7) and we print it out (line 8). In this case, the `computer_pick` is `rock`.

[1] https://en.wikipedia.org/wiki/Rock_paper_scissors

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 *rock*, *paper*, 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 out 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—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 or the player hat (see page 14). 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 **evaluate and verify** 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. What is parallelism?

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"
2
3     # compare to the player's choice
4     if player_choice == "scissors":
5         print("Tie!")
6     elif player_choice == "rock":
7         print("You win!")
8     else:
9         print("The computer wins!")
```

```
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!
```

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 the conditions 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 the `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 coding syntax and thinking. 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]:

```
1  import random
2
3  # list of game possibilities
4  possibilities = ["rock", "paper", "scissors"]
5  # computer random pick
6  computer_pick = random.choice(possibilities)
7
8  # asking the player to make their choice
9  player_choice = input ("Rock, paper, or
   scissors?")
10
11 # determine who wins
12 # if the computer picks rock
13 if computer_pick == "rock":
14     # compare to the player's choice
15     if player_choice == "rock":
16         print("Tie!")
17     elif player_choice == "paper":
18         print("You win!")
19     else:
20         print("The computer wins!")
21
22 # if the computer picks paper
23 if computer_pick == "paper":
24     # compare to the player's choice
25     if player_choice == "paper":
26         print("Tie!")
27     elif player_choice == "scissors":
28         print("You win!")
29     else:
30         print("The computer wins!")
21
12 # if the computer picks scissors
33 if computer_pick == "scissors"
34     # compare to the player's choice
35     if player_choice == "scissors":
36         print("Tie!")
37     elif player_choice == "rock":
38         print("You win!")
39     else:
40         print("The computer wins!")
```

```
Rock, paper, or scissors? rock
You win!
```

```
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!
```

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 paragraphs *1. Computer pick*, *2. Player choice*, and *3. Determine who wins*). Finally, we merged all the code together and edited it (see *Merging the code*). 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 original project

In other words, there are three steps to solve a computational (but not strictly computational!) task:

1. Break the project into subprojects
2. Solve the subprojects separately
3. Merge the solutions of the subprojects 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 parallelism, testing, debugging, and divide and conquer

Why do we say Debugging, Divide and conquer, and Algorithms?

Do you know why we say *debugging*, *divide and conquer*, and *algorithms*? The term *debugging* originated in 1947, when a moth was found in a relay of 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].

[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

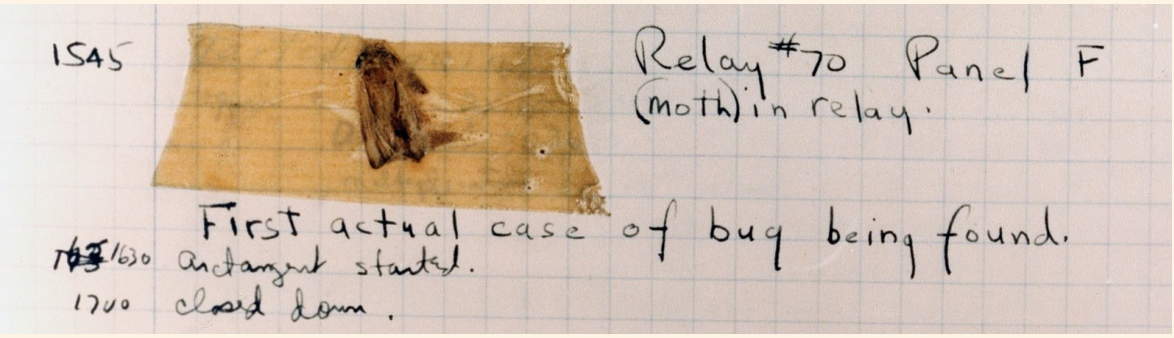


Figure 16.1. The page of the log from the Mark II with the moth taped on it. It dates 9 September 1947. The time is 15:45 as visible on the top left. The log book is at the Smithsonian Institution's National Museum of American History in Washington, D.C., United States.

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 out 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 part 5, you will learn the last construct in coding: the while loop. You will also learn various types of conditions that you can use in 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 is incremented 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 syntax and functionality 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 out the 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
22 # print the final number of candies
23 print("You have a total of" +
   str(number_of_candies) + " 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 is incremented 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
```

```
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 out 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 out 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 out 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 (line 8), (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 symbol**, and we can pronounce it as *incremented 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, `-=` (*decrease 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 while 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. 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 bunch 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, then 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 to the solution (divide and conquer!). This time, also try to take it one a 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. For each of the following examples, you will see a possible way to approach the coding task at hand. Maybe it will be similar to your thinking, or maybe it will be different. In any case, it will give you an idea of possible thinking pathways. You can play with the proposed solutions on Notebook 18. Enough talk—let's start coding!

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 they guess the animal picked by the computer. At the end of the game, tell the player how many attempts it took to guess the animal.

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
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 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
12 print("Well done! You guessed " +
   computer_pick)
```

```
as long as the player's guess and the
computer's pick are different
while player guess is 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, until `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 out a message confirming that the player guessed the right animal (line 12).

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
```

```
initializing the counter
n of attempts is assigned one
```

```
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!")
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)
```

```
as long as the player's guess and the computer's pick are different
while player guess is 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 is incremented 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
```

```
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 1
```

```
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)
```

```
as long as the player's guess and the computer's pick are different
while player guess is 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 is incremented 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
```

```
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 the `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, then print the following: *The number x is already in the list*. How many numbers did you generate before finding 8 unique numbers?

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, then print *The number x is already in the list*; and (4) how many numbers did you generate before finding 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 an
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 out 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 of the previous chapter *Writing code is like writing an email!* 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 code:

```
1  import random
2
3  # initialize the number list
4  unique_random_numbers = []
5
```

```
import random

initialize the number list
unique random numbers is assigned an
empty list
```

```
6  # as long as the length of the list is not 8                as long as the length of the list is not eight
7  while len(unique_random_numbers) != 8:                       while len of unique random numbers is not equal to eight
8
9      # create a random number between 0 and 10                create a random number between zero and ten
10     number = random.randint(0,10)                              number is assigned random dot randint zero ten
11
12     # if the number is already in the list                   if the number is already in the list
13     if number in unique_random_numbers:                        if number in unique random numbers:
14         # place holder                                         place holder
15         a = 0                                                  a is assigned zero
16     # otherwise                                                otherwise
17     else:                                                      else:
18         # add the new number to the list                       add the new number to the list
19         unique_random_numbers.append(number)                   unique random numbers dot append number
20
21 # print the list                                               print the list
22 print(unique_random_numbers)                                   print unique random numbers
[1, 8, 10, 7, 3, 0, 5, 9]
```

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 right now we can just use a placeholder, or a nonfunctional command in our code that we plan to substitute (`a=0`, line 15). Using placeholders is not very good coding practice, but sometimes we can make an exception in the very early drafting phase. 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, then print: The number x is already in the list
We substitute the placeholder `a=0` with the print commands (line 15):

```
1  import random                                                  import random
2
3  # initialize the number list                                  initialize the number list
4  unique_random_numbers = []                                     unique random numbers is assigned an empty list
5
```

```
# as long as the length of the list is not 8
while len(unique_random_numbers) != 8:


    # create a random number between 0 and 10
    number = random.randint(0,10)


    # 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 " + str(number) +
        " is already in the list")

    # otherwise
    else:
        # add the new number to the list
        unique_random_numbers.append(number)


# print the list
print(unique_random_numbers)
```

```
as long as the length of the list is not eight
while len of unique random numbers is not equal to eight


create a random number between zero and ten
number is assigned random dot randint eight 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 before finding 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]:

```
import random

# initialize the number list
unique_random_numbers = []


# initialize the counter
counter = 0

# as long as the length of the list is not 8
while len(unique_random_numbers) != 8:


    # create a random number between 0 and 10
    number = random.randint(0,10)
```

```
import random


initialize the number list
unique random numbers is assigned an empty list


initialize the counter
counter is assigned zero

as long as the length of the list is not eight
while len of unique random numbers is not equal to eight


create a random number between zero and ten
number is assigned random dot randint zero ten
```

```
14
15      # increase the counter by 1                    increase the counter by one
16      counter += 1                                   counter is incremented by one
17
18      # if the number is already in the list         if the number is already in the list
19      if number in unique_random_numbers:            if number in unique random numbers:
20          # print that the number is in the list     print that the number is in the list
21          print ("The number " + str(number) +      print The number  concatenated with
            " is already in the list")                 str number concatenated with is
                                                       already in the list
22      # otherwise                                    otherwise
23      else:                                          else:
24          # add the new number to the list           add the new number to the list
25          unique_random_numbers.append(number)       unique random numbers dot append
                                                       number
26
27  # print the final list and the total amount      print the final list and the total
    of generated numbers                               amount of generated numbers
28  print(unique_random_numbers)                       print unique random numbers
29  print("The total amount of generated numbers      print The total amount of generated
    is: + str(counter))                                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 out (line 29).

3. Sum up the multiples of 3

- Write code that continues asking a player to enter an integer until they enter a negative number. At the end, print the sum of all entered integers that are multiples of 3.

The task has two requests: (1) keep asking a player to enter an integer until they enter 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 they enter 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
5  while number >= 0:
6      # ask for the next new integer
7      number = int(input("Enter another
       integer: "))
```

```
Enter an integer: 3
Enter another integer: 6
Enter another integer: 4
Enter another integer: -1
```

```
as long as the number positive
while number is greater than or
equal to zero
ask for the next new integer
number is assigned int input Enter
another integer:
```

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 condition the variable in the while loop header (as we saw in 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, then sum them up. 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
21 # print the list of multiples of 3
19 print(numbers)
20
21 # initialize the sum to 0
22 sum_of_numbers = 0
23
```

```
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 positive
while number is greater than or
equal to zero

if the number is multiple of 3
if number modulus three is equal to
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 3
print numbers

initialize the sum to zero
sum of numbers is assigned zero
```

```
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))
```

```
calculate the sum of numbers
for i in range len of 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 can 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 operator. 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 up the numbers in the list, similarly to what we did in the exercise 5 of 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—to add the current list element (`numbers[i]`) to the amount in `sum_of_numbers` (line 26). Finally, we print out the sum at line 29.

We solved the task, 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 out their sum. 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
```

```
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 positive
while number is greater than or
equal to zero

if the number is a multiple of 3
if number modulus three is equal to
zero:
add the number to the sum
sum of numbers is incremented by
number

ask for the next integer
number is assigned int input Enter
another integer:
```

```
18 # print the final sum
19 print("The sum of the multiples of 3 is: " +
   str(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
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

In coding there are three constructs: `if/else` constructs, `for` loops, and `while` loops. Review their definitions in the following exercise:

1. An if/else construct checks whether a condition is true or false
2. A for loop is the repetition of a group of commands
3. A while loop is the repetition of a group of commands

a. for a determined number of times

b. as long as a condition holds

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 constructs, it can be easy to confuse the while loop with 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 the coming lines!

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 would 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 )
10 else:
11     print("Nope! Try again!")
```

```
check the player guess
if player guess is equal to
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 9–11). 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 9–11), and so on. This is not feasible because we cannot know how many times it is going to take the player to guess the correct fruit! In addition, we would repeat code, which means that we can use a loop! So, here is the correct solution with the while loop:

```
9  while player_guess != computer_pick:
10     # as long as the player's guess and the not right
11     player_guess = input ("Nope! Try again! Guess the fruit! Choices: mango, orange banana: ")
```

```
while player guess is not equal to computer pick:
as long as the player's guess and the computer's pick are different
player guess is assigned input Nope! Try again! Guess the fruit! Choices: mango, orange, banana:
```

As long as the `player_guess` is not equal to `computer_pick` (line 9), we ask the player to make a guess (line 11), which we check in the condition of the while loop header (line 9), 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 1 list of words, create 3 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 out 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")
```

```
The number 5 is between 5 and 10
The number 7 is between 5 and 10
The number 8 is between 5 and 10
```

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

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 7).

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 special characters—you might remember the special character `"\n"`, which is used to go to a new line (Chapter 12). The backslash `\` tells Python that the following quote `"` is an actual backslash character and not the symbol that we use to close a string. The last backslash `"\"\/` is an actual backslash because the following forward slash `/` is not a special 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)
```

```
a is a vowel or a punctuation
e is a vowel or a punctuation
a is a vowel or a punctuation
```

```
string of vowels
vowels is assigned aeiou

initialize counter
counter is assigned zero

for each position in the message
for i in range len of 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 is increased by one

print the final amount
print(The total amount of punctuation
or vowels is  concatenated with
counter
```

```
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`: 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:

[7]:
```
1  numbers = [4, 6, 7, 9]
```

```
numbers is assigned four, six, seven,
nine
```

- Print out the numbers that are **not** divisible by 2:

[8]:
```
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])
```

```
for each position in the list
for i in range len of numbers

# if the current number is not even
if not numbers in position i modulo
two equals zero

print the current number
print numbers in position i
```

```
7
9
```

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 one:

[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 of numbers

# if the current number is odd
if numbers in position i modulo two is
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 *Solving arithmetic expressions* in Chapter 13). The order from highest to lowest precedence is: not, and, or (easy-to-memorize acronym: **NAO**). When you are uncertain, write the condition to prioritize within round brackets `()`.

4. not in

- Generate 5 random numbers between 0 and 10. If the random numbers 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                                  for five times
4  for _ in range (5):                              for underscore in range five
5
6      # generate a random number between 0 and 10  generate a random number between zero
                                                    and ten
7      number = random.randint(0, 10)               number is assigned random dot randint
                                                    zero ten
8      # print the number as a check                print the number as a check
9      print (number)                               print number
10
11     # if the new number is not in numbers        if the new number is not in numbers
12     if number not in numbers:                    if number not in numbers:
13         # add the number to numbers              add the number to numbers
14         numbers.append(number)                   numbers dot append number
15
16 # print the final list                            print the final list
17 print (numbers)                                   printnumbers
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 *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 pages on its site (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 version(s), 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 (gitlab.com) and Bitbucket (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"` will 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 invite 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 is 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 is 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` (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 is 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 is 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 we did for `True`.

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 is 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 the operation 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 is greater than one
print number is less than five
print number is greater than one and number is
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`. 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 3
print number is greater than four
print number is less than five
print number is greater than four and number is
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 3
print number is greater than one
print number is less than two
print number is greater than one and number is
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 3
print number is greater than four
print number is less than two
print number is greater than four and number is
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):
False
```

```
print False and 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 `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:

[13]:

```
# 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 is incremented 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?

As you might remember from Chapter 17, for a while loop, we have to create a variable that is: (1) initialized *before the header*, (2) included in a condition *within the header*, and (3) allowed to change *in the body* to avoid infinite iterations. In the example in Chapter 17, the variable following these three rules was `answer`. In this example, it is `flag`. We initialize `flag` as a Boolean of value `True` (line 5), then we check if its value is equal to `True` in the while loop header (line 11), and finally we allow it to change to `False` (line 32) to avoid infinite loops. `flag` is a common variable name for a Boolean variable that behaves this way—`counter` is another typical variable name for a variable that keeps count of the number of iterations. We can think of a *flag* variable like a traffic light that makes the loop continue or stop. As long as the traffic light is green (i.e., `flag` is `True`), the loop will continue. When the traffic light changes to red (i.e., `flag` is assigned `False`), the loop ends. Using a Boolean flag in the while loop is somewhat like providing the answer to a condition instead of asking the header to test the condition.

When using a flag, the construction of a while loop might change. What about the variable `answer` in this new code version? We initialize `answer` at the beginning of the while loop body, where we use the built-in function `input` to ask a question to the player (line 14). Then we create an if/else condition to decide what to do based on the value of `answer` (lines 17–32). If the `answer` is `"yes"`, then we increment the counter `number_of_candies` by `1` (line 20) and we print a feedback to the player (line 23). Otherwise (i.e., `else`), we print a final feedback to the player (line 29) and we allow the flag to change (line 32).

These are several ways to write a while loop. Which one should we use? All have pros and cons. Choose the one that appears simpler and easier to understand!

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`, `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 whose solutions we can find online!

Let's code!

1. *Do you want less exercises?* Rewrite the while loop from the exercise *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 out:
   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

FOCUS ON LISTS AND FOR LOOPS

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 put the "grammar" rules for Python lists to use and highlight some new important properties that are worth knowing. The Chapter contains a lot of examples and details that will help you improve your coding skills and understand other people's code. Let's start! Follow along with Notebook 21!

1. Arithmetic operations on list elements

As you might remember from Chapter 13, in Python there are 7 arithmetic operations: addition (`+`), subtraction (`-`), multiplication (`*`), exponentiation (`**`), division (`/`), floor division (`//`), and modulo (`%`). To perform arithmetic operations *element-wise*—that is, on list elements—we use for loops. **Element-wise operations can be done (1) between two or more lists of the *same length* or (2) between a list and a number**. In both cases, we **use a for loop**. Let's see two examples for addition (but they can be valid for any operation).

- Sum two lists element-wise:

[1]:
```
1 odd_numbers = [1, 3, 5]
2 even_numbers = [2, 4, 6]
3 summed = []
4
5 for i in range (len(odd_numbers)):
6     summed.append(odd_numbers[i] +
                    even_numbers[i])
7
8 print (summed)
```
```
[3, 7, 11]
```
```
odd_numbers is assigned one, three, five
even_numbers is assigned two, four, six
summed is assigned empty list

for i in range len odd numbers
summed dot append odd_numbers in position i
plus even numbers in position i

print summed
```

We start with `odd_numbers` and `even_numbers`, which are two lists containing 3 integers each (lines 1 and 2), and `summed`, which we initialize as an empty list (line 3). Then, we create a for loop that spans the indices of one of the lists of numbers (line 5), and we append to `summed` the sum of the current element of the list `odd_numbers` to the element in the same position in the list `even_numbers` (line 6). Finally, we print the result for a check (line 8). Note that we save the result in a third list (`summed`) that we initialized as empty before the loop (line 3) and that we fill in during the loop (line 6). If we do not want to create a third list, we can overwrite one of the existing lists (e.g., `odd_numbers[i]=odd_numbers[i] + even_numbers[i]`).

- Sum a number to each element of a list:

[2]:
```
1 numbers = [1, 2, 3]
2 number = 3
3
```
```
odd_numbers is assigned one, two, three
number is assigned three
```

```
4  for i in range (len(numbers)):          for i in range len of numbers
5      numbers[i] += number                numbers in position i incremented 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, but they use arithmetic symbols with a different meaning. **The two possible operations are concatenation**, which uses the symbol `+` (pronounced as *concatenated with*) **and replication**, which uses the symbol `*` (pronounced as *replicated by [number]*). Let's see the examples:

- 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, one containing odd numbers (`odd_numbers`; line 1) and one containing even numbers (`even_numbers`; line 2). Then we concatenate them using the concatenation symbol `+` (line 3), and we store the result in a new list called `concatenate` (line 3). If we don't want to create a new variable, we can overwrite one of the two existing lists: `odd_numbers = odd_numbers + even_numbers`. Finally, we print the result (line 4), which is a list containing the elements of `odd_numbers` and `even_numbers` sequentially merged.

- Replicate a list 3 times:

```
[4]:
1  numbers = [1, 2, 3]                     odd_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 can overwrite the existing list: `numbers = numbers * number`. Finally, we print `replicated` (line 4). As you can see in the printout, `replicated` contains the list `numbers` repeated three times. When

is replication useful? Let's see 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]
```

We initialize `short_list` as a list containing one zero (line 1) and the variable `short_list` containing the value `50` (line 2). Then, we replicate `short_list` by the number of times indicated by `number` (line 3), and we store the result in the variable `long_list`. 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! Finally, note that in alternative to create the variables `short_list` and `number`, we can directly write: `long_list=[0]*50`.

3. List assignment

When we assign a list to another list, 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]
      2 new_list = given_list.copy()
      3 new_list[0] = 40
      4
      5 print (given_list)
      6 print (new_list)
```

```
[1, 2, 3]
[40, 2, 3]
```

```
given list is assigned one, two, three
new list is assigned given list dot copy
new list in position zero is assigned
forty
print given list
print new list
```

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` (line 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 like if we created a twin that is similar but independent, so that when we make changes, they happen only in the list we actually change. At the end of the example, we change the first element of `new_list` to `40` like we did in cell 8 (line 3), and we print out both lists (lines 4 and 5).

4. Adding one element or a list to a list

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 see two easy examples to refresh how the methods work.

- Add one element at the end of a list:

```
[11]: 1 numbers = [1, 2, 3]
      2 numbers.append(4)
      3 print (numbers)
```

```
[1, 2, 3, 4]
```

```
numbers is assigned one, two, three
numbers dot append four
print numbers
```

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).

- Insert the number 2 in position 1:

```
[12]: 1 numbers = [1, 3, 4]
      2 numbers.insert(1, 2)
      3 print (numbers)
```

```
[1, 2, 3, 4]
```

```
numbers is assigned one, three, four
numbers dot insert at position one, two
print numbers
```

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 out `numbers` (line 3).

There are two ways to add **a list** at the **end** of another list: **concatenation** (see cell 3 and another example below) and the **method `.extend()`**.

- 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).

- 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 use the same two lists as in cell 13 (lines 1 and 2), but 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). Then, 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 we can add a list to an existing one (e.g., `first_list = first_list + second_list`). Instead, when using `.extend()`, we can only modify the list to which we apply the method (i.e., `first_list` in cell 14). In addition, when using **`.extend()`**, we can add a list **only at the end** of another list, whereas when using **concatenation—combined with slicing**—we can add a list **at the beginning** (e.g. `first_list = second_list + first_list`) **or in the middle** of another list (e.g. `first_list = first_list[:2] + second_list + first_list[2:]`).

5. Removing elements from a list

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.

- 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! In lists containing multiple similar elements, the method **`.remove()` deletes only the first element**. How do we remove both `"ciao"` from `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 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).

Let's continue to see how to remove an element based on its position and all elements in a list. In the following two cells (17 and 18), we write the list at line 1, and we print the result at line 3. At line 2, we use a different list method. Let's have a look at the examples:

- 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
```

To remove an element based on its position, we use the method `.pop()`, which we learn in Chapter 5 (line 2). As you might remember, **the argument of the method is the position of the element to delete**.

- 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
```

To remove all elements in a list, we use the method `.clear()` (line 2). The list **becomes an empty list**.

Another way to remove elements in a list is by using list comprehension. We will see it in the next chapter.

6. Sorting a list

Sorting lists is a very common task in coding. For example, we might want to sort names alphabetically (see the exercise "A further step!" below) or a list of prices increasingly or decreasingly. In the three examples below (cells 19, 20, and 21), we will create a list of integers called `numbers` (line 1), use a new method to execute the task (line 2), and print the outcome (line 3).

- Sort the following list of integers:

```
[19]: 1 numbers = [5, 7, 6]                    numbers is assigned five, seven, six
      2 numbers.sort()                         numbers dot sort
      3 print (numbers)                        print numbers
      [5, 6, 7]
```

To sort the list `number`, we use the method **`.sort()`** (line 2). As you can see from the printout, 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:

- Sort the following list of integers in a descending way:

```
[20]: 1 numbers = [5, 7, 6]                    numbers is assigned five, seven, six
      2 numbers.sort(reverse = True)           numbers dot sort reverse is assigned True
      3 print (numbers)                        print numbers
      [7, 6, 5]
```

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) parameters starting in Chapter 28. As you can see from the printout, the list is now sorted in a descending way: that is, from the greatest to the smallest number.

- Reverse the following list of integers:

```
[21]: 1 numbers = [5, 7, 6]                    numbers is assigned five, seven, six
      2 numbers.reverse()                      numbers dot reverse
      3 print (numbers)                        print numbers
      [6, 7, 5]
```

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 `.reverse()` sorts the element based on their *position*, whereas `.sort()` (see example above) sorts the elements based on their *value*.

7. Searching elements

Let's conclude our long journey through list methods by learning how to search and count elements.

- Create a list and search for a specific element:

```
[22]: 1 letters = ["a", "g", "c", "g"]         letters is assigned a, g, c, g
      2 position = letters.index("g")          position is assigned letters dot index g
      3 print (position)                       print position
      1
```

We create the list `letters` containing strings (line 1), and we **look for the position of the element** `"g"` by using the method `.index()`, which we learned in Chapter 5. Then, we print the results (line 3). As you can see, `.index()` just gives us the position of the first element, which is `1`—because element positions start from `0` in Python.

- 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 of letters
if letters in position i is equal to 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).

- 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), and we use the **method `.count()` to count how many times** the letter `"g"` is in the list (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 learned or refreshed the 11 list methods. Fill out the table below with method definitions and alternative ways to implement the same operation. 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 methods for lists are: `.append()`, `.clear()`, `.copy()`, `.count()`, `.extend()`, `.index()`, `.insert()`, `.pop()`, `.remove()`, `.reverse()`, `.sort()`
- Of the 11 methods, the 3 methods that return a new 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 see them in this example:

```
[1]:
1  greetings = ["ciao","ciao","hello"]          greetings is assigned ciao,ciao,hello
2  for i in range (len(greetings)):             for i in range len of greetings
3      print ("---------------")                print dashes
4      print ("i == " + str(i))                 print i equal to concatenated with str i
5      print ("before the if:")                 print before the if:
6      print ("greetings")                      print greetings
7      if greetings[i] == "ciao":               if greetings in position i is equal to ciao
8          del greetings[i]                     del greetings in position i
9      print ("after the if:")                  print after the if:
10     print ("greetings")                      print greetings

---------------
(a)  i == 0
(b)  before the if:
(c)  ['ciao', 'ciao', 'hello']
(d)  after the if:
(e)  ['ciao', 'hello']
(f)  ---------------
(g)  i == 1
(h)  before the if:
(i)  ['ciao', 'hello']
(j)  after the if:
(k)  ['ciao', 'hello']
(l)  ---------------
(m)  i == 2
(n)  before the if:
(o)  ['ciao', 'hello']

---------------------------------------
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 Paragraph 5 (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`, which we learned in Chapter 6 (line 8). 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), and the list before deletion (lines 5 and 6) and after deletion (lines 9 and 10).
Note that for clarity of the following explanation, the printed lines are identified with letters, which are not actually printed when running the code.

Let's see what happens at each loop:

- First loop (`i==0`): before the `if`, the list is complete `["ciao", "ciao", "hello"]` (line (c)). After the `if`, `greetings` contains only `["ciao", "hello"]` (line (e)). Three changes happened: (1) the string `"ciao"` in position `0` (in orange in Figure 21.1) is removed; (2) the element indices restarted from `0`, changing the positions of the remaining elements (that is, the green `"ciao"` was in position `1` before the if and moved to position `0` after the if, and the string `"hello"` was in position 2 before the if and moved to position `1` after the if); and (3) the length of the list changed from 3 to 2. The changes (2) and (3) will have consequences in the second and third loops.

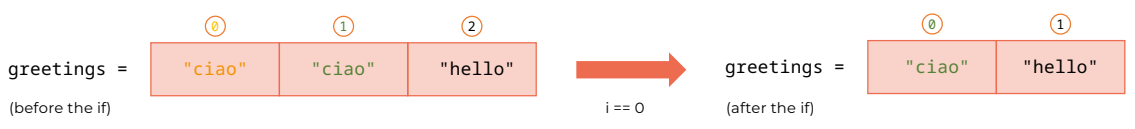


Figure 21.1. Change of list content, element positions, and list length after deletion of a list element.

- Second loop (`i==1`): before the `if`, the list is the same as it was at the end of the previous loop, that is `["ciao", "hello"]` (line (i)). And after the if, the list remains the same (k) because the current element `greetings[1]`, that is, `"hello"`, does not satisfy the if condition. Why wasn't the string `"ciao"` in position `0` (green in Figure 21.1) deleted? The change of list index in the previous loop moved `"ciao"` from position `1` to position `0`, so we skip its deletion because we are currently at the second iteration of the for loop!
- Third loop (`i==2`): before the `if`, the list is still `["ciao", "hello"]` (line (0)). Than, we get an index error at line 6 of the code, where the if conditions is. This is because `i` is now `2`, but `greetings[2]` does not exist because we shortened the list when we deleted the first `"ciao"` in the first loop. Thus, the error "out of range" is due to a failed attempt to slice the list `greetings` in position `2`, which does not exist! Note that the index `i` is currently `2` because in the header of the for loop (line 2), we stated that `i` goes from `0` to the length of the list (`len(greetings)`), which is the *initial* list length and does not adapt to length changes during the loop!

In conclusion, by using a for loop to delete an element in a list, we can cause two errors: (1) we **skip list elements** that we should delete because of the index shift, and (2) we get **out of range errors** related to the index because we shorten the list by removing some elements.

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	4	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 are 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 potatoes. 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 if/else.
   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. More about the for loop

Various ways of repeating commands on lists and beyond

In the past several chapters, we have learned how to use the for loop to browse lists (Chapters 8 and 9), search elements in lists (Chapter 10), change list elements (Chapter 11), and create lists by adding one element at a time (Chapter 12). In addition, we have used the for loop to repeat commands independently of lists (see the *In more depth* section in Chapter 15). We will start this Chapter by briefly refreshing what we already know for sake of completeness. Then, we will discover new for loops that we can use with lists, each of them with their own characteristics and usage. Ready? Follow along with notebook 22!

1. Repeating commands

As the definition says,

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

Let's get a refresher on this concept with the following example:

- 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 ways to use the for loop with lists. You already know the first one: the for loop through indices. In this section, we'll learn the for loop through elements, through indices and ele-

ments, and list comprehension. Note that *through indices*, *through elements*, and *through indices and elements* are not technical terms; however, we will use them to distinguish between the different types of for loops. On the contrary, *list comprehension* is a technical term that you can find in any Python book or coding website. In all the examples in this section, 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

You already know this for loop type. Let's refresh our memories with the following example.

- Capitalize each string using a for loop through *indices*:

```
[2]:  1  last_names = ["garcia", "smith", "zhang"]

      2

      3  for i in range (len(last_names)):
      4      print ("The element in position " + str (i) +
             " is: " + last_names[i])

      5      last_names[i] = last_names[i].title()

      6

      7  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']
```

```
last names is assigned garcia, smith, zhang

for i in range len last names
print The element in position concatenated with str of i concatenated with is concatenated with last names in position i
last names in position i is assigned last names in position i dot title

print last names
```

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(0,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 creates a range of integers**: for example, `list(range(0,4))` 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).

2.2 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)
```

```
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
```

```
The current element is: garcia
The current element is: smith
The current element is: zhang
['Garcia', 'Smith', 'Zhang']
```

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))`. Let's now focus on 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 in a for loop through elements, **the variable in position (2)**—that is, `last_name`—**automatically browses list elements one after the other, without knowing their position**. This is the opposite of what happens in a for loop through *indices*, where the variable in position (2)—that is, `i`—browses list positions without knowing the corresponding elements; 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.

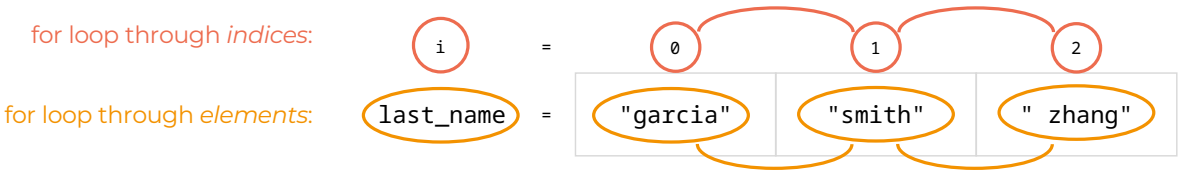


Figure 22.1. Schematics of a for loop through *indices*, where an index browses positions (orange), and a for loop through *elements*, where a variable browses elements (yellow).

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 procedure follows in the third iteration for `"zhang"`. However, when we print the final list, all strings are still lower case (line 6). This is **because the for loop through *elements* does not keep track of element positions, so it is impossible to know *where* to overwrite a list element**. 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()
```

```
last names is assigned garcia, smith, zhang

for i last_name in enumerate last_names
print The element in position concatenated with str of i concatenated with is concatenated with last name
last names in position i is assigned last name 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']
```

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 data type into a list to be printed):

```
[]:  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 Chapter 29. During the for loop in this example, the variable `i` is 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 6). The **for loop through indices and positions** 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*. It might look complex at first glance, but we are going to untangle it right away!

- Capitalize each string using *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 symbol; 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.

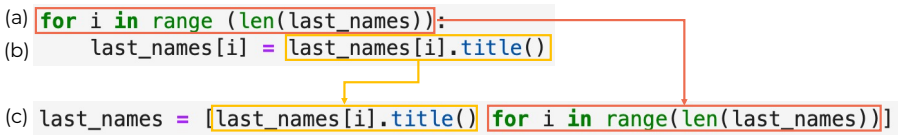


Figure 22.2. Comparison between a for loop through *indices* (lines a–b) and list comprehension (line c).

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). As you can see, **list comprehension is a one-line command to create or modify a list in a fast and compact way**. We conclude the previous example by printing the new list (line 3).

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"]      last names is assigned garcia, smith,
                                                        zhang
      2  last_names = [last_name.title() for           last names is assigned last name dot
                       last_name in last_names]         title for last name in last names
      3  print (last_names)                             print last names
['Garcia', 'Smith', 'Zhang']
```

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?

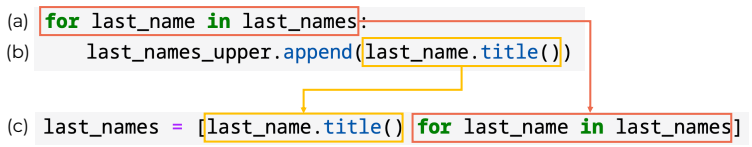


Figure 22.3. Comparison between for loop through *elements* (lines a–b) and list comprehension (line c).

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. 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.

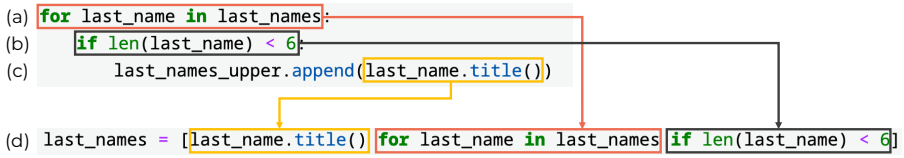


Figure 22.4. Comparison between for loop through *elements* with condition (lines a, b, and c) and list comprehension (line c).

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 essentially 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 list comprehensions, **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 must write the element that we are going to insert into the list. Thus, if we want 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 do 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*
Get the current index				
Change list elements				
Delete list elements				
Browse a full list				
Browse only a part of a list				

3. Nested for loops

As the last topic for this Chapter, let's learn about nested for loops. A **nested for loop** is a **for loop within 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)):
      2     print ("-- " + vowels[i])
      3
      3     for j in range (i + 1, len(vowels)):
      4         print (vowels[i])
```

```
-- A
E
I
O
U
-- E
I
O
U
```

for i in range len vowels
print dash dash concatenated with vowels in position i
for j in range i plus one len vowels
print vowels in position i

```
-- 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 (line 1); 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 (line 3); 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 `3` and we print `vowels[3]`, 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 coincides with the stop, which is `5` too.

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. Find all Markdown rules of syntax at the following website: `https://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 out 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 out 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 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 out 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 out 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

What is a list of lists?

> A **list of lists is a list** whose **elements are lists**.

Lists of lists follow the same rules as lists; they just add an "extra layer" of indices. In this Chapter, you will learn how to slice lists of lists, use nested for loops to iterate through them, and explore ways to flatten them. Follow along with Notebook 23. Let's go!

1. Slicing

To slice a list of lists, we modify the slicing rules that we learned for lists in Chapter 6: by **adding an 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-element**. 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])
```

```
['dog', 'cat']
['cow', 'sheep', 'horse', 'chicken', 'rabbit']
['panda', 'elephant', 'giraffe', 'penguin']
```

```
print animals in position zero
print animals in position one
print animals in position two
```

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 sub-element `"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. Nested for loops

To browse elements in a list of lists, we can use a nested for loop, 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 a 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 might go about doing this before looking into the following code.

- Print the sub-list elements one-by-one using a 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 a nested for loop 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 transform a list of lists into a list whose elements are `instruments`'s sub-elements.

- Flatten the list using a nested for loop 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     flattened += group                   flattened 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 the list comprehension we can use a nested for loop 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

Lists of lists and images

You surely know that digital images are composed of pixels, that is, small colorful squares organized in a grid. We can think of the grid as a list of lists where each sub-element corresponds to a pixel of a specific color. Let's consider Figure 23.1.

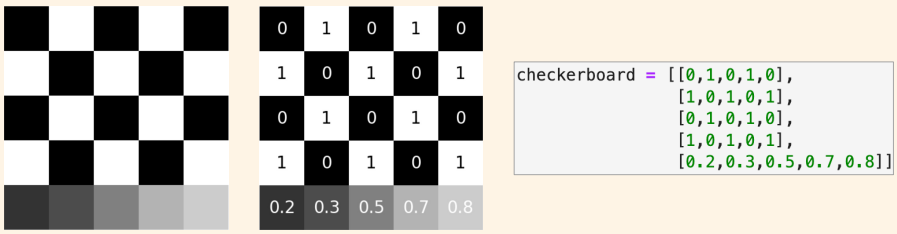


Figure 23.1. Digital representation of a checkerboard. Left: Image rendering. Center: Numerical values corresponding to the checkerboard colors. Right: List of lists encoding the checkerboard colors.

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 is represented by the sub-list `[0,1,0,1,0]`, and the second (and the fourth) row is represented by the sub-list `[1,0,1,0,1]`. The last row of the checkerboard contains pixels colored with various shades of grey. Each pixel corresponds to a decimal (float) number. Darker greys are closer to `0` (that is, to black), whereas brighter greys are closer to `1` (that is, to white).

What about digital *colored* images? Each pixel is encoded by an 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.

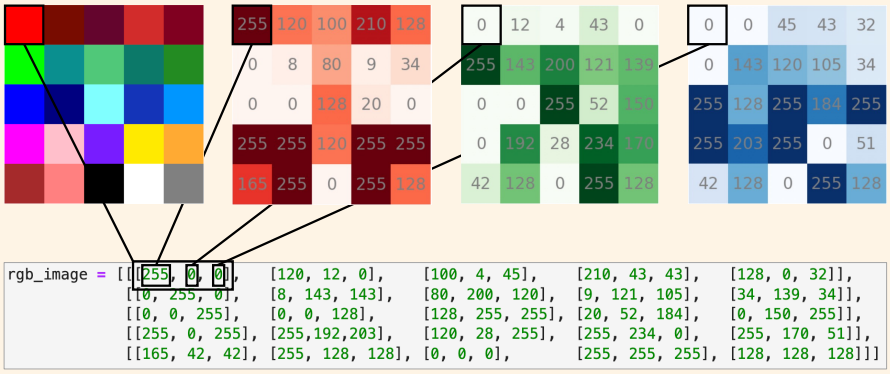


Figure 23.2. Digital representation of a colored image. Top (from left to right): RBG image, red components, green components, and blue components. Bottom: list of lists behind the rendered colored image.

Each pixel is represented by a sub-list composed of three numbers. For example, the top left pixel is red and is represented by the sub-list `[255, 0, 0]`, where `255` represents the amount of red, the first `0` is for the amount of green, and the second zero `0` is for the amount of blue. 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 list using (1) a for loop through indices and (2) a for loop through values.
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 datatype called dictionary. In the last Chapter, you will integrate your knowledge of strings with new methods and tricks. Let's go!

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_items
     2  print (classics.keys())
     3  # authors as a list
     4  print (list(classics.keys()))
     5
     6  # titles as dict_items
     7  print (classics.values())
     8  # titles as a list
     9  print (list(classics.values()))
```

```
authors as dict_items
print classics dot keys
authors as a list
print list classics dot keys

titles as dict_items
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:

[]:

```
# deleting the first book (syntax 1)
del classics["Austen"]
print (classics)

# deleting the second book (syntax 2)
classics.pop("Joyce")
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 discover dictionaries step-by-step. Let's start by running the first cell.

[1]:

```
classics = {"Austen":"Pride and Prejudice",
            "Shelley":"Frankenstein",
            "Joyce":"Ulyssessss"}
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 `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]:

```
# as dict_items
print (classics.items())
# as a list of tuples
print (list(classics.items()))
```

```
as dict_items
print classics dot items
as a list of tuples
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 printout, `.items()` returns a specific type called `dict_items`, which contains a list whose elements are the items. We can ignore `dict_items` and print the contained list by enclosing the method output into the built-in function `list()` (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_items                    authors as dict_items
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_items                     titles as dict_items
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 its datatype, called `dict_keys` (line 4). To extract the list of keys from the `dict_keys`, we can use the built-in function `list()`. Finally, to extract dictionary *values*, we use the method `.values()` (line 7), which returns the list of values embedded in another datatype called `dict_values`. Once again, to extract the list of values, we use `list()` (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"])      print Wrong title: concatenated
                                                    with classics at key Joyce
2  classics["Joyce"] = "Ulysses"                    classics at key Joyce is assigned
                                                    Ulysses
3  print ("Corrected title: " + classics["Joyce"])  print Corrected title: concatenated
                                                    with classics at key Joyce

Wrong title: Ulyssessss
Corrected title: Ulysses
```

To slice a *value*, the syntax is `dictionary[key]` (pronunciation: dictionary at key), as we can see in `classics["Joyce"]` (line 1). Isn't it similar to the slicing syntax for lists? Let's analyze some similarities and differences between dictionaries and lists 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 lists and dictionaries lies in the way we de-

fine 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 are in no specific order**. Also, note that **numbers cannot be used as keys!**

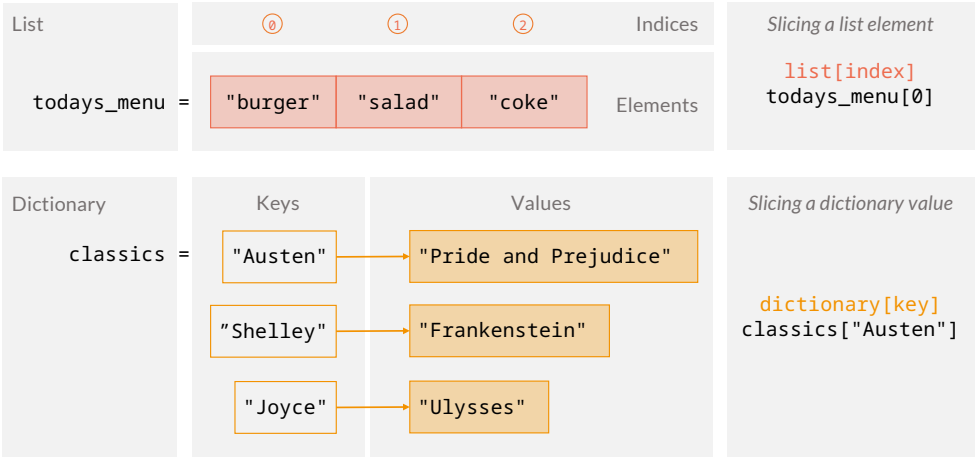


Figure 24.1. Comparing structure and slicing syntax for lists (top) and dictionaries (bottom).

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"`, 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 `key:value` pair** 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 is to use a **slicing-like syntax**, where we write: (1) dictionary name (`classics`); (2) new key in between square brackets (`["Swift"]`); (3) assignment symbol (=); and (4) new value (`"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 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 (`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 comma, 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 1
```

```
[{'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 keys and their values.
   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*.

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 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. Thus, you 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 an 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 is incremented 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 an 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 go about doing so? 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 is incremented 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")
2  print (tips)
```

```
tips dot get food dot append onion tarte
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 a key:value pair 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():                for k in tips dot keys
     2     print (k)                        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:

```
[8]: 1 for k,v in tips.items():             for k v in tips dot items
     2     print ("{:>7}: {}".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 symbols in the string contain two pairs of curly brackets, one with the 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}`? It is composed of three parts: (1) the symbol `:` indicates to print the whole text; (2) the symbol `>` specifies that the text is aligned to the right; and (3) the symbol `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 in 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"]
```

```
---------------------------------------
KeyError       Traceback (most recent call last)
Cell In[3], line 1
----> 1 tips ["city"]
KeyError: 'city'
```

tips at key 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'`—it should be `'cities'`! To know where the error is, we look for the green arrow, which shows that we need to correct at line 1. To fix it, we just replace `"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 a *key:value* pair 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 = {}                          letter counter is assigned an empty
                                                dictionary
2
3  for letter in greetings:                     for letter in greetings
4      if letter not in letter_counter.keys():  if letter not in letter counter dot keys
5          letter_counter[letter] = 1           letter counter at key letter is assigned
                                                one
6      else:                                    else
7          letter_counter[letter] += 1          letter counter at key letter is
                                                incremented by one
8
9  for k,v in letter_counter.items():           for k v in letter counter dot items
10     print (k,v)                              print k v
```

```
h 2
e 2
l 2
o 3
! 1
  3
w 1
a 1
r 1
y 1
u 1
? 1
```

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 the

same way for lists and strings. Then, for each character, we check if it is a key of `letter_counter` and we act accordingly (lines 4–7). More precisely, we first 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 `key:value` pair, 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 is `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**ll**o). 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**ll**o). 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 = {}                                  letter counter is assigned an empty
                                                        dictionary
2
3  for letter in greetings:                             for letter in greetings
4      letter_counter[letter] =                         letter counter at key letter is assigned
         letter_counter.get(letter, 0) + 1              letter counter dot get letter zero plus
                                                        one
5
6  for k,v in letter_counter.items():                   for k v in letter counter dot items
7      print (k,v)                                      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**. Thus, 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 `hello`—then `.get(letter, 0)` returns `0`. Then, we add `1` to `0`, and we create a new `key:value` pair 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 `key:value` pair 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` for 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 require a significant amount of computer memory, and it would 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 a 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)
```

```
create the dictionary
sparse dict is assigned an empty
dictionary
for i in range len of sparse vector
if sparse vector in position i is 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 of sparse_vector

print
for k v in sparse dict dot items
print k v
```

```
3 1
5 7
8 4
12 8
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 `key:value` pair 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 0 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 is 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 in correspondence with the key `"length"`. Then, we overwrite the nonzero values into the list. With a for loop, we browse each `key:value` pair 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]:
```
registry = {"Shaili":4, "Chris":90,
"Maria":70}
```
```
registry is assigned Shaili:4, Chris:90,
Maria:70
```

- Sort the dictionary items according to their *keys*:

[8]:
```
# create a new dictionary
sorted_registry = {}


# sort the keys
sorted_keys = list(registry.keys())

sorted_keys.sort()

# fill out the new dictionary
for k in sorted_keys:
    sorted_registry[k] = registry[k]


print (sorted_registry)
```
```
create a new dictionary
sorted registry is assigned empty
dictionary

sort the keys
sorted keys is assigned list registry dot
keys
sorted keys dot sort

fill out the new dictionary
for k in sorted keys
sorted registry at key k is assigned
registry at key k

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()` and then transform its output—whose type is `dict_keys`—into a list using the built-in function `list()` (line 5). Then, 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 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]:
```
# create a new dictionary
sorted_registry = {}

# sort keys according to values
sorted_keys = sorted(registry,
                     key = registry.get)

# fill out the new dictionary
for k in sorted_keys:
    sorted_registry[k] = registry[k]


print (sorted_registry)
```
```
create a new dictionary
sorted registry is assigned empty dictionary

sort keys according to values
sorted keys is assigned sorted registry key
is assigned registry dot get

fill out the new dictionary
for k in sorted keys
sorted registry at key k is assigned
registry at key k

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 dictionary combined with the method `.get` (note the absence of round brackets). Note that `sorted()` can be used also with lists and strings—mainly with only one argument—as an alternative to the method `.sort()`. The difference is that `sorted()` returns a variable (e.g., `sorted_list = sorted(original_list)`), whereas `.sort()` directly acts on the 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)` is used to initialize a `key:value` pair 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 `key:value` pair
- `.setdefault()`: Returns the value of the specified key. If the key does not exist, then it inserts the new `key:value` pair 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. Follow along with Notebook 27. As usual, try to solve the tasks before looking into the solution. Let's start!

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 rsecwyadrkd
```

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 from the beginning to the
                                                     end 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 from the beginning to the
                                                     end 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 lists do. Let's quickly look at a refresher on how they work.

- 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 symbol `+` to obtain `"sunscreen"` (line 3). We print the result to check for correctness (line 4).

- 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 so 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. Let's start by **changing a substring based on its *position.

- 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)
<ipython-input-13-ef0756c89224> in <module>
----> 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
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 out the resulting string (line 3).

- 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).

What if we want to **modify a substring based on its *content*** instead of position? Let's have a look!

- 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).

What about **removing substrings**? If we want to remove based on **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[:6] + favorites[15:]`, and we get: `I like my family, and my friends`. On the other side, 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 parallel: 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!

- 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                    initializing positions
2  positions = []                              positions is assigned empty list
3
4  # find all positions of e                   find all positions of e
5  for i in range (len(us)):                   for i in range len of us
6      if us[i] == "e":                        if us in position i is equal to e
7          positions.append(i)                 positions dot append i
8  print (positions)                           print positions
[1, 5]
```

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).

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")                    positions is assigned us dot find f
2  print (positions)                           print positions
-1
```

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,      hobbies is assigned I like going to the
   traveling, and singing"                     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")          n substrings is assigned string dot count ing
2  print (n_substrings)                        print n substrings
4
```

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.

- 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()          list of strings is assigned string dot
                                                  split
      2 print (list_of_strings)                   print list of strings
      ['How', 'are', 'you']
```

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`.

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)   string from list is assigned space dot
                                                      join list of strings
      2 print (string_from_list)                      print string from list
      How are you
```

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 peculiar 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 concatenated 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!

- 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
2  print ("Good " + part_of_day + "!")

3  # (2) variable separation by comma
4  print ("Good", part_of_day, "!")
5  # (3) the method .format()
6  print ("Good {}!".format(part_of_day))

7  # (4) f-strings
8  print (f"""Good {part_of_day}!""")
```

```
Good morning!
Good morning !
Good morning!
Good morning!
```

```
(1) string concatenation
print Good concatenated with part of day
concatenated with !
(2) variable separation by comma
print Good part of day !
(3) the method .format()
print Good placeholder ! dot format part
of day
(4) f-strings
print f Good part of day !
```

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).

What if we want to print a string combined with a numerical variable? Let's have a look!

- Given a string and a numerical variable:

```
[25]:
1  part_of_day = "morning"
1  time_of_day = 10
```

```
part of day is assigned morning
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.
```

```
(1) string concatenation
print Good concatenated with part of
day concatenated with ! backslash n
It's concatenated with str time of day
concatenated with a.m.
(2) variable separation by comma
print Good part of day ! backslash n It's
time of day a.m.
(3) the method .format()
print Good placeholder ! backslash n It's
placeholder a.m. dot format part of day
time of day
(4) f-strings
print f Good part of day !
It's time of day a.m.
```

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 printout, 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).

What about printing numbers with a reduced number of decimals? Let's see the following examples.

- Given the numerical variable:

[27]:
```
1  number = 1.2345
```

```
number is assigned 1.2345
```

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))
```

```
(1) string concatenation
print The number is concatenated
with str round number two
(2) variable separation by comma
print The number is round number
two
```

```
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}""")
```

```
(3) the method .format()
print The number is colon dot two f
dot format number
(4) f-strings
print f The number is number colon
dot two f
```

```
The number is 1.23
The number is 1.23
The number is 1.23
The number is 1.23
```

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 operations such as concatenation, assignment, or 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 many resources that you'll find 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 can 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): 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]:
print ("Shopping list:\napples\noranges")
```

```
Shopping list:
apples
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]:
print ("Dear friend,")
print ("\tI hope you are doing fine. I have some news...")
```

```
Dear friend,
	I hope you are doing fine. I have some news...
```

```
print Dear friend,
print 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]:
print ("The wise said: \"To live a happy life...\"")
```

```
The wise said: "To live a happy life..."
```

```
print 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!')
     It's the best time!
```

```
print 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 more 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 will learn how to use the coding syntax and computational thinking that you have learned so far to create units of code called *functions*. Let's get started!

28. Printing Thank you cards

Function inputs

To this point, we've learned Python data types—lists, strings, Booleans, dictionaries, integers, and floats. We've also learned how to combine these data types with operators—assignment, membership, arithmetic, comparison, and logical—to create commands—that is, statements, if/else conditions, for loops, and while loops. The next step is to learn how to combine commands into **units of code** called **functions**. We are already somewhat familiar with functions because we have frequently used Python built-in functions, such as `print()`, `len()`, `range()`, etc. In this Part, we 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 in 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)` (line 10) 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]:
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
```

What happens? Apparently nothing! Let's run the two following cells:

```
[2]:
1  print_thank_you ("Maria")
Thank you Maria
```

```
print thank you Maria
```

```
[3]:
1  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 with 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 line), 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 line), 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* session 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 three 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**. Then, 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**. Finally, we must **use all parameters** in the function code. 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")
```
```
---------------------------------------
TypeError          Traceback (most recent call last)
<ipython-input-13-ef0756c89224> in <module>
----> 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")
```
```
Thank you Mrs Maria Lopez
```

```
print thank you Mrs Maria Lopez
```

```
[9]:  1  print_thank_you ("Mr", "Xiao")
      Thank you Mr Xiao
```

print 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). 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 its version 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 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")
      ---------------------------------------
      TypeError          Traceback (most recent call last)
      <ipython-input-13-ef0756c89224> in <module>
      ----> 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
```

```
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
```

```
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)
```

```
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 (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 a 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, minimizing copy-pasting is crucial not only because it is tedious, but doing so also reduces 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 (see Chapter 16). 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* session 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! Moreover, how do 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. Finally, be aware that `help()` can be used for any functions, including Python built-in functions, like you can see in the following example:

```
[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 key/value pairs in a dictionary, etc. We will learn more about *returning* 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. Do so 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 is 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 take a closer look at how to 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
```

```
def create username first name last name
Creates a lowercase username made of initial of first name and full last name
```

```
    Parameters                                  Parameters
    ----------
    first_name : string                         first name : string
        First name of a person                  First name of a person
    last_name : string                          last name : string
        Last name of a person                   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 0 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
```

- Test the function for two customers:

```
[2]: 1  username_1 = create_username("Julia", "Smith")      username one is assigned create
                                                           username Julia Smith
     2  print (username_1)                                  print username one
jsmith
```

```
[3]: 1  username_2 = create_username("Mohammed", "Seid")    username two is assigned create
                                                           username Mohammed Seid
     2  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). In the following command, `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 *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 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
```

- Test the function 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* session 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 (cells 5 and 6, line 2).

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 = [["Maria", "Lopez"], ["Julia",
   "Smith"], ["Mohammed", "Seid"]]
```

```
customers is assigned Maria Lopez
Julia Smith Mohammed Seid
```

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: {'mlopez': '7097', 'jsmith': '6891', 'mseid': '3189'}
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. Similarly, the second variable—

number_customers—contains the returned variable n_customers (cell 7, line 36). When we print number_customers 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 = 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: ({'mlopez': '6350', 'jsmith': '7863', 'mseid': '1953'},3)
Database: {'mlopez': '6350', 'jsmith': '7863', 'mseid': '1953'}
Number of customers: 3
```

In this second case, we assign both returned variables to a 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). Obviously, we could have directly printed the sliced variable in both cases—that is, `print ("Database:", outputs [0])` and `print ("Number of customers:", outputs [1])`.

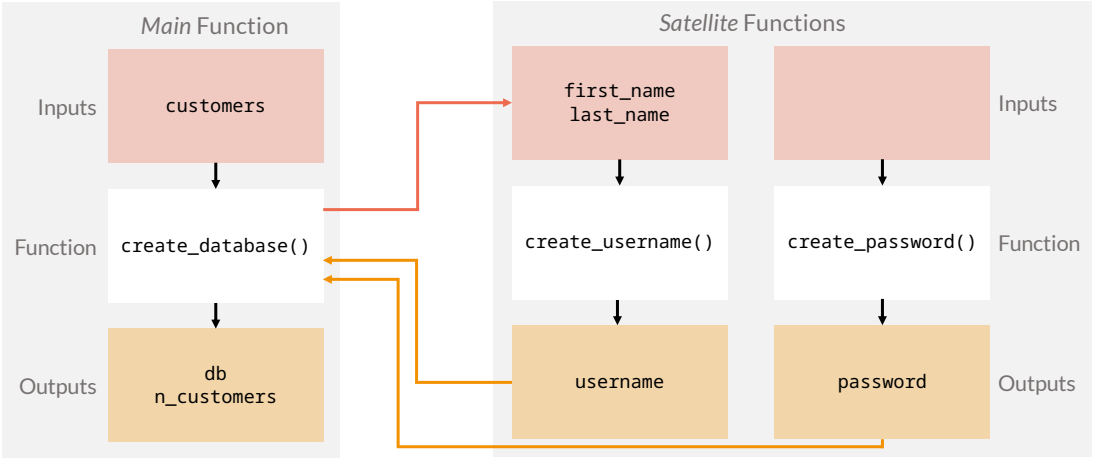


Figure 29.2. Modular organization of code: Main and satellite functions.

Before concluding this Chapter, let's briefly analyze how we *modularized* our code with the help of Figure 29.2. 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 some coding examples outside this book, you may find that the the main function is actually called `main()`. The other two functions—`create_username()` and `create_password()`—are **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* session 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 running 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.

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 exercises a, b, and c, you will implement some techniques to preprocess text—that is, to clean text for the main analysis—and in exercise 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
tohowtoacademy, 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 example 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

    E.g. 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
E.g. 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 letters")

    # 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 letters

age has to be between 0 and 125
if age less than zero or age greater
than 125
raise ValueError age must be between 0
and 125



returning output

if age greater than 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 possibilities for **type** 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—to test one parameter at the time (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`. 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! We've learned quite a lot since then, haven't we? 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 `.isalpha()`, `.isdigit()`, `.isalnum()`, `.islower()`, `.isupper()`, `.istitle()`—see 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 letters (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.alpha()
----> 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 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.

[1] `https://en.wikipedia.org/wiki/List_of_the_verified_oldest_people`

Let's conclude 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 (lines 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 do not create 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 return it as `return message`. We directly return the concatenation. Note that **in the docstrings we only indicate the type**—`string`—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

Review what you have learned in this chapter with the following exercise:

1. `raise` is a
2. `TypeError()` and `ValueError()` are
3. `int` is a
4. `int()` is a
5. `isinstance()` is a

a. built-in function
b. keyword
c. built-in function
d. exceptions
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, 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 point *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 exercises. The first three exercises will invite you to implement satellite functions, each of them representing a sub-task. The last function will suggest 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)
```

```
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
```

```
24
```

Does your implementation look similar to the one above?

- Compare the previous iterative function with the following recursive function:

[2]:

```
def factorial_rec (n):
    """Calculates the factorial of a given
    integer using recursion

    Parameters
    ----------
    n : integer
        The input integer

    Returns
    -------
    integer
        The factorial of the input integer
    """

    # if integer is greater than 1
    if n > 1:
        # execute the recursion
        return factorial_rec(n-1) * n
    # otherwise
    else:
        # return 1
        return 1

# call the function
fact = factorial_rec(4)
print (fact)
```

```
24
```

```
def factorial rec n
Calculates the factorial of a given
integer using recursion

Parameters

n : integer
The input integer

Returns

integer
The factorial of the input integer


if integer is greater than 1
if n is greater than 1
execute the recursion
return factorial rec n minus one times n
otherwise
else
return one
return one

call the function
fact is assigned factorial rec four
print fact
```

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`—as in *n*umber—(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 cur-

rent 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.

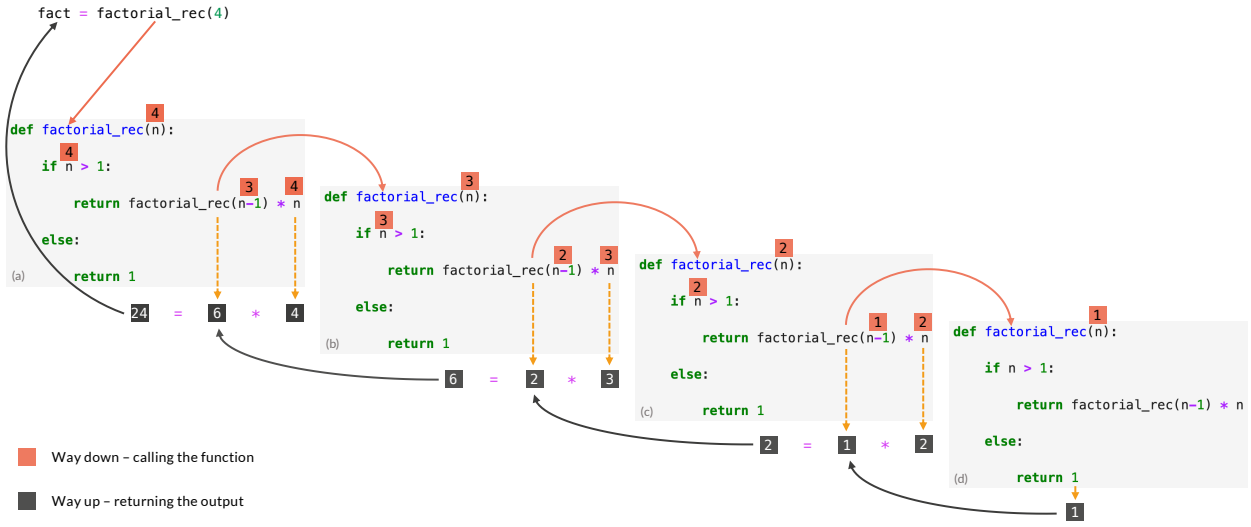


Figure 31.1. The mechanism of a recursive function.

Figure 31.1 contains four major components. First, there is the *initial* function call `fact = factorial_rec(4)` (corresponding to line 25 in cell 2)—see the top left of Figure 31.1. Second, there are four simplified representations 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 conditions—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 conditions. 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).

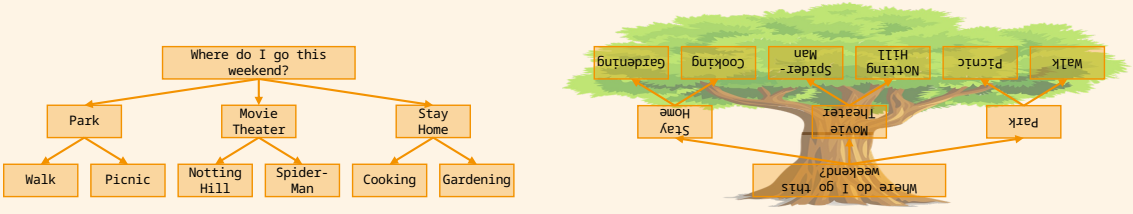
Let's conclude with some advantages and disadvantages of recursive functions. On the one hand, recursive functions contain **compact code** and are appropriate when solving **intrinsically recursive** problems—see the *In more depth* session at the end of this Chapter. On the other hand, they are **computationally 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 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—see a simplified example in Figure 31.2 (bottom). 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 (top right).




```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 (that is, 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
```

```python


# 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 to illustrate the similarity to a natural tree (top right); and simplified code to traverse the decision tree (bottom).

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!

References

- The book cover is inspired by the cover of the book “Working in Public: The Making and Maintenance of Open Source Software” by Nadia Eghbal. Stripe Press. 2020
- Some examples and exercises in the book are inspired from online material and from the book “Coding for Kids: Python: Learn to Code with 50 Awesome Games and Activities” by Adrienne Tacke. Rockridge Press. 2019

Dear coder,

Thanks for learning with me!

Visit `www.learnpythonwithjupyter.com` to:

- Find more information about the book
- Download the Jupyter Notebooks
- Join the LPWJ Community for exercise solutions and Q&A

Your feedbacks and comments are fundamental to improve the book!

- Fill out the feedback form at: `www.tinyurl.com/lpwj-feedback`, and/or
- Email me at: serena.bonaretti.research@gmail.com

I am looking forward to hearing from you!

Next release: find the date on `www.learnpythonwithjupyter.com`

Have fun coding!
Serena