

Learn  
**Python**  
with  
**Jupyter**



Serena Bonaretti

[www.learnpythonwithjupyter.com](http://www.learnpythonwithjupyter.com)



# Learn Python with Jupyter

Serena Bonaretti

[www.learnpythonwithjupyter.com](http://www.learnpythonwithjupyter.com)

For the free ebook:

License: CC BY-NC-SA

For the future printed copy:

Copyright ©202x by Serena Bonaretti. All rights reserved.

No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, or otherwise without the prior written permission of the author.

While the author has used good faith efforts to ensure that the information and instructions contained in this work are accurate, the author disclaims all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use or reliance of this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

Editing and proofreading by John Batson

Cover design by Federica Dias ([www.behance.net/federicadidas](http://www.behance.net/federicadidas))

[www.learnpythonwithjupyter.com](http://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 together again to learn how to read and write.*

*Actually, I would go further: to learn the world and ourselves better.*

—Alberto Manzi, *Non è mai troppo tardi, It's never too late*

Simple is better than complex.

—Tim Peters, *The Zen of Python*



# Content

| About this book<br>p. ix  |  |  |  |
|---|--|--|--|
| Introduction  |  |  |  |
| What we need to learn when learning coding<br>p. xiii                           |  |  |  |
| Getting ready   |  |  |  |
| The Jupyter/Python environment<br>p. 3<br>Downloading the book material<br>p. 8 |  |  |  |
| Part 1: Creating the basics   |  |  |  |
| Chapter   | Syntax   | Computational thinking   | In more depth  |
| 1. Text, questions, and art<br>p. 11  | <ul style="list-style-type: none"> <li>■ Strings</li> <li>■ Built-in function <code>input()</code></li> <li>■ Built-in function <code>print()</code></li> </ul>                | <ul style="list-style-type: none"> <li>■ Getting information from a user</li> <li>■ Printing to the screen</li> </ul>  | <i>Our fingers have memory</i><br>p. 16                |
| 2. Events and favorites<br>p. 18  | <ul style="list-style-type: none"> <li>■ Assignment symbol</li> <li>■ Concatenation symbol</li> </ul>  | <ul style="list-style-type: none"> <li>■ Creating variables</li> <li>■ Assigning values to variables</li> <li>■ Concatenating strings</li> </ul>   | <i>Dealing with NameError and SyntaxError</i><br>p. 21 |
| Part 2: Introduction to lists and if/else                                       |  |  |  |
| Chapter   | Syntax   | Computational thinking   | In more depth  |
| 3. In a bookstore<br>p. 25  | <ul style="list-style-type: none"> <li>■ Lists</li> <li>■ if/else construct</li> <li>■ Membership operator <code>in</code></li> <li>■ Indentation</li> </ul>                   | <ul style="list-style-type: none"> <li>■ List as a collection datatype</li> <li>■ Executing command based on binary conditions</li> </ul>  | <i>Let's give variables meaningful names!</i><br>p. 28 |
| 4. Grocery shopping<br>p. 30  | <ul style="list-style-type: none"> <li>■ List method <code>.append()</code></li> <li>■ List method <code>.remove()</code></li> </ul>   | <ul style="list-style-type: none"> <li>■ Methods as functions for a specific datatype</li> <li>■ Adding and removing elements to/from a list based on conditions</li> </ul>  | <i>Why do we print so much?</i><br>p. 34               |
| 5. Customizing the burger menu<br>p. 36   | <ul style="list-style-type: none"> <li>■ List method <code>.index()</code></li> <li>■ List method <code>.pop()</code></li> <li>■ List method <code>.insert()</code></li> </ul> | <ul style="list-style-type: none"> <li>■ Associating a list element to an index</li> <li>■ Finding an element index</li> <li>■ Adding and removing elements to/from a list based on index</li> </ul>   | <i>We code in English!</i><br>p. 39                    |
| 6. Traveling around the world<br>p. 41  | <ul style="list-style-type: none"> <li>■ <i>Three-s</i> rule</li> <li>■ <i>Plus one</i> rule and <i>minus one</i> rule</li> </ul>  | <ul style="list-style-type: none"> <li>■ Slicing to extract elements from a list</li> <li>■ Slicing using positive and negative indices, and in direct and reverse order</li> <li>■ Omitting indices</li> </ul>                                    | <i>Why the plus one rule?</i><br>p. 47                 |
| 7. Senses, planets, and a house<br>p. 50  | <ul style="list-style-type: none"> <li>■ Keyword <code>del</code></li> </ul>   | <ul style="list-style-type: none"> <li>■ Replacing, adding, and removing elements using list slicing</li> <li>■ List concatenation</li> <li>■ Deleting a variable vs. its content</li> <li>■ Transitioning from list methods to slicing</li> </ul> | <i>What is the kernel?</i><br>p. 56                    |

| Part 3. Introduction to the for loop       |  |   |   |
|--|--|---|---|
| Chapter                                    | Syntax   | Computational thinking  | In more depth   |
| 8. My friends' favorite dishes<br>p. 61    | <ul style="list-style-type: none"> <li>■ for loop</li> <li>■ Built-in function range()</li> <li>■ Built-in function str()</li> </ul>   | <ul style="list-style-type: none"> <li>■ For loop to repeat commands</li> <li>■ For loop to automatically slice a list</li> </ul>   | <i>Dealing with IndexError and IndentationError</i><br>p. 66                  |
| 9. At the zoo<br>p. 69                     | <ul style="list-style-type: none"> <li>■ Comparison operator ==</li> <li>■ Built-in function len()</li> <li>■ # for comments</li> <li>■ Abbreviating index with i</li> </ul> | <ul style="list-style-type: none"> <li>■ Binary condition in command repetition</li> <li>■ Code commenting</li> </ul>   | <i>Dealing with TypeError</i><br>p. 73  |
| 10. Where are my gloves?<br>p. 76          | <ul style="list-style-type: none"> <li>■ Comparison operators !=, &gt;, &gt;=, &lt;, &lt;=</li> </ul>  | <ul style="list-style-type: none"> <li>■ Searching an element in a list based on element length or position, by combining for loop and if/else construct</li> <li>■ Using variables in place of hard-coded values</li> </ul>                          | <i>Let's use keyboard shortcuts!</i><br>p. 82                                 |
| 11. Cleaning the mailing list<br>p. 85     | <ul style="list-style-type: none"> <li>■ String methods<br/>  .lower(), .upper(),<br/>  .title(),<br/>  .capitalize()</li> </ul>   | <ul style="list-style-type: none"> <li>■ Changing list elements in a for loop with reassignment</li> </ul>  | <i>In what list am I changing the element?</i><br>p. 88                       |
| 12. What a mess at the bookstore!<br>p. 91 | <ul style="list-style-type: none"> <li>■ Special character "\n"</li> </ul>   | <ul style="list-style-type: none"> <li>■ Creating lists in a for loop</li> <li>■ String slicing</li> <li>■ Multiple consecutive slicing</li> </ul>  | <i>Append or concatenate. Don't assign!</i><br>p. 96                          |
| Part 4. Numbers and algorithms             |  |   |   |
| Chapter                                    | Syntax   | Computational thinking  | In more depth   |
| 13. Implementing a calculator<br>p. 101    | <ul style="list-style-type: none"> <li>■ Arithmetic operators</li> <li>■ Built-in functions int(), float(), type()</li> <li>■ Keyword elif</li> </ul>                        | <ul style="list-style-type: none"> <li>■ Number variables as strings, integers, or floats</li> <li>■ Testing multiple variable values using elif</li> <li>■ Combining code in a code unit</li> </ul>  | <i>Solving arithmetic expressions</i><br>p. 108                               |
| 14. Playing with numbers<br>p. 110         | (no new syntax)  | <ul style="list-style-type: none"> <li>■ Changing numbers based on conditions</li> <li>■ Separating numbers based on conditions</li> <li>■ Finding the maximum in a list of numbers</li> <li>■ Determining number visibility using modulus</li> </ul> | <i>Don't name variables with reserved words!</i><br>p. 113                    |
| 15. Fortune cookies<br>p. 116              | <ul style="list-style-type: none"> <li>■ Keyword import</li> <li>■ random module functions<br/>  .randing(a,b) and<br/>  .choice(list)</li> </ul>                            | <ul style="list-style-type: none"> <li>■ Module as a unit containing specific functions</li> <li>■ Importing a module</li> <li>■ Randomness in coding</li> </ul>  | <i>What if I don't use the index in a for loop?</i><br>p. 119                 |
| 16. Rock paper scissors<br>p. 121          | (no new syntax)  | <ul style="list-style-type: none"> <li>■ Testing, debugging, parallelism, divide and conquer, algorithm</li> </ul>  | <i>Why do we say Debugging, Divide and conquer, and Algorithms?</i><br>p. 127 |



Part 5. The while loop and conditions

| Chapter  | Syntax   | Computational thinking   | In more depth   |
|--|--|--|---|
| 17. Do you want more candies?<br>p. 131        | <ul style="list-style-type: none"> <li>■ Keyword <code>while</code></li> </ul>   | <ul style="list-style-type: none"> <li>■ While loop to ask for unknown number of inputs</li> <li>■ Counter</li> <li>■ Initializing and changing for the variable in the condition</li> </ul> | <i>Writing code is like writing an email!</i><br>p. 135     |
| 18. Animals, unique numbers, and sum<br>p. 137 | (no new syntax)  | <ul style="list-style-type: none"> <li>■ Identifying various kinds of conditions</li> <li>■ Problem solving using divide and conquer</li> </ul>  | <i>Don't confuse the while loop with if/else!</i><br>p. 147 |
| 19. And, or, not, not in<br>p. 150             | <ul style="list-style-type: none"> <li>■ The logical operators <code>and</code>, <code>or</code>, and <code>not</code></li> <li>■ The membership operator <code>not in</code></li> </ul> | <ul style="list-style-type: none"> <li>■ Merging conditions</li> <li>■ Reversing conditions</li> </ul>   | <i>What is GitHub?</i><br>p. 155                            |



## 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](http://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](http://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. 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. 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).

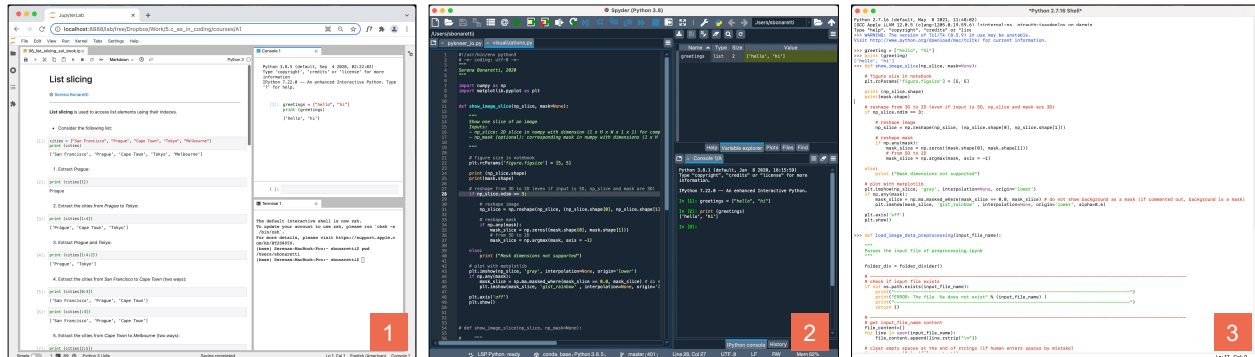


Figure 1. Three IDEs to code in Python: (1) the Jupyter environment, (2) Spyder, and (3) the Python IDLE.

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<br>( <i>words</i> )   | Operators<br>( <i>punctuations</i> )                                | Constructs and<br>loops<br>( <i>sentences</i> ) | Unit of code<br>( <i>paragraphs</i> ) | Software<br>( <i>texts</i> )                   |
|--|---|---|---------------------------------------|--|
| string, list,<br>integer, float,<br>Boolean, tuple,<br>dictionary, set | assignment,<br>membership,<br>arithmetic,<br>comparison,<br>logical | if/else construct,<br>for loop,<br>while loop   | functions                             | classes (object-<br>oriented pro-<br>gramming) |

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.







# 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!

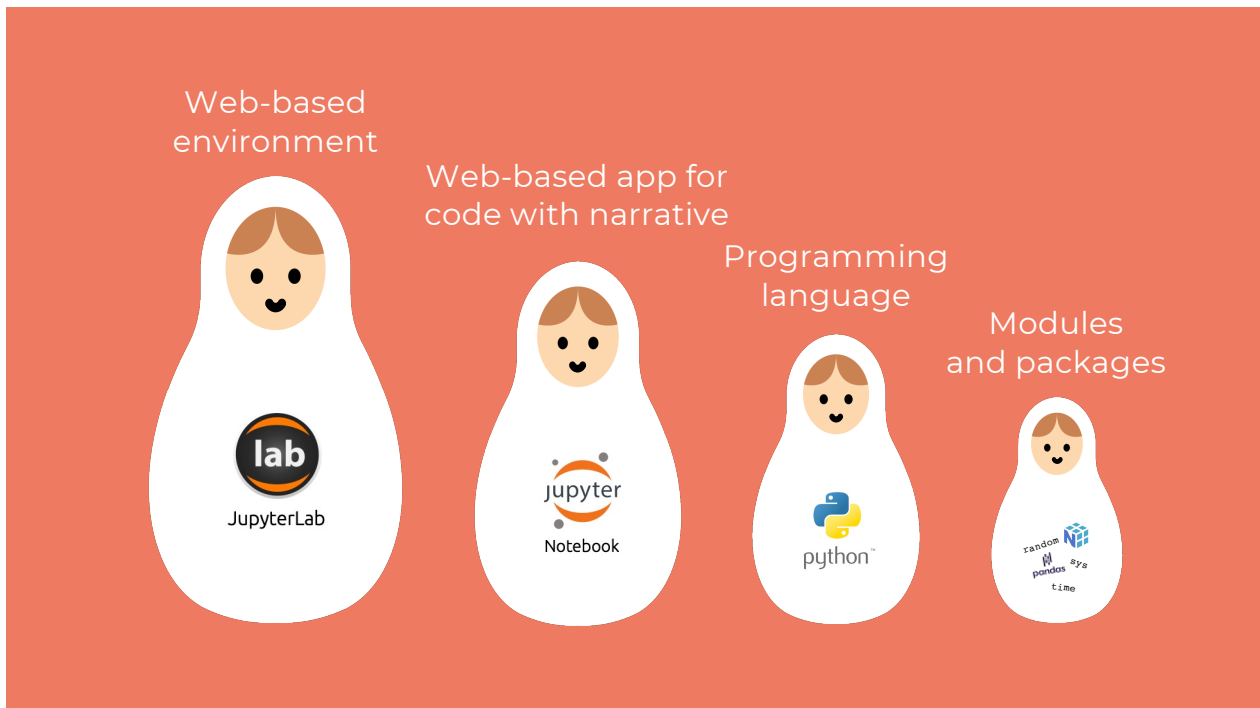


Figure 2. The Jupyter/Python environment represented as a Russian doll, where each component is included in the previous one.

## 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!

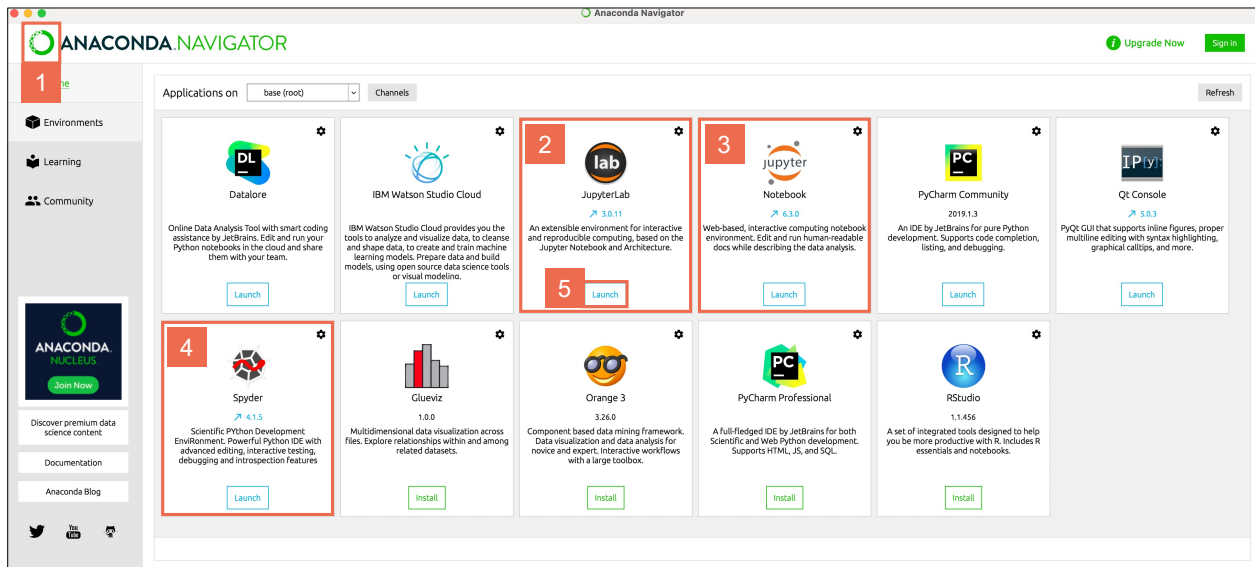


Figure 3. Anaconda interface: (1) icon, (2) JupyterLab, (3) Jupyter Notebook, (4) Spyder, and (5) JupyterLab launch button.

## 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 panel 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

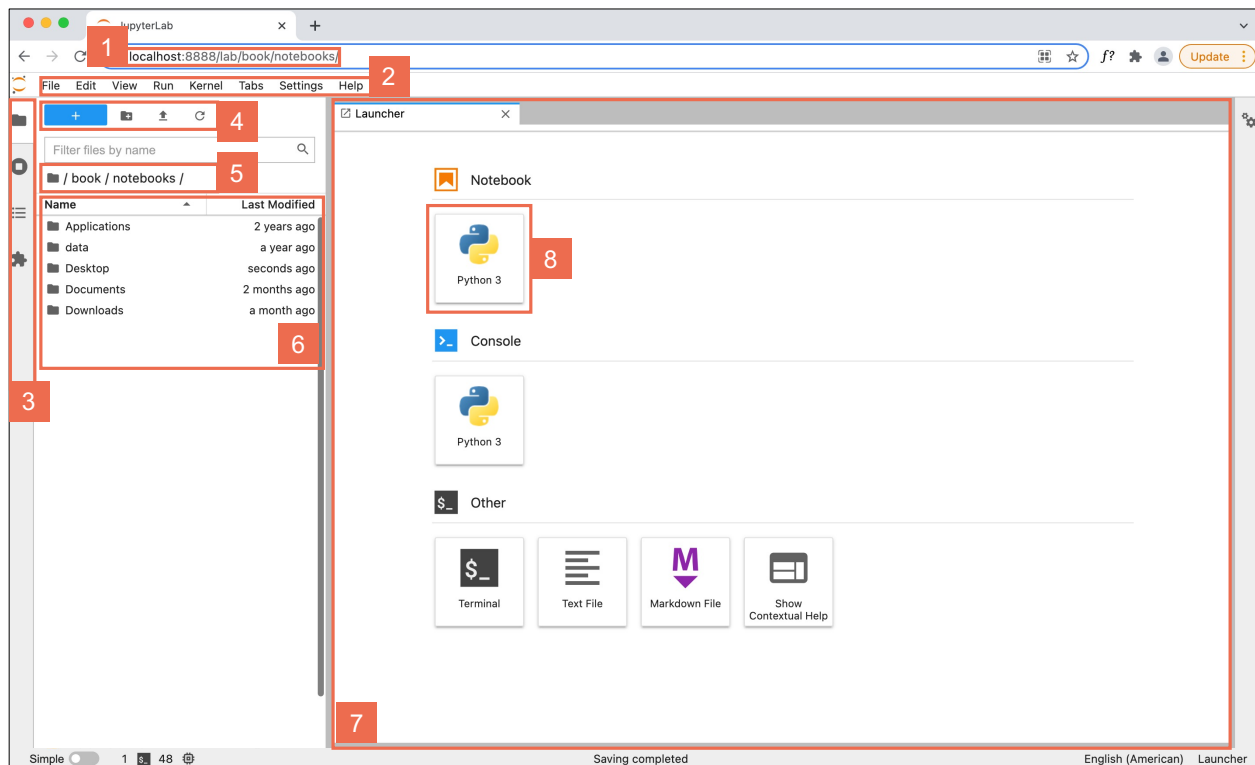


Figure 4. JupyterLab interface, containing: (1) local URL, (2) top bar, (3) lateral tabs, (4) folder browser top bar, (5) folder browser, (6) folder content, (7) launcher, and (8) Jupyter Notebook launch button.

arrow pointing up, to upload a new file; and a circular arrow, to refresh the content of the current directory—in coding, 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 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.

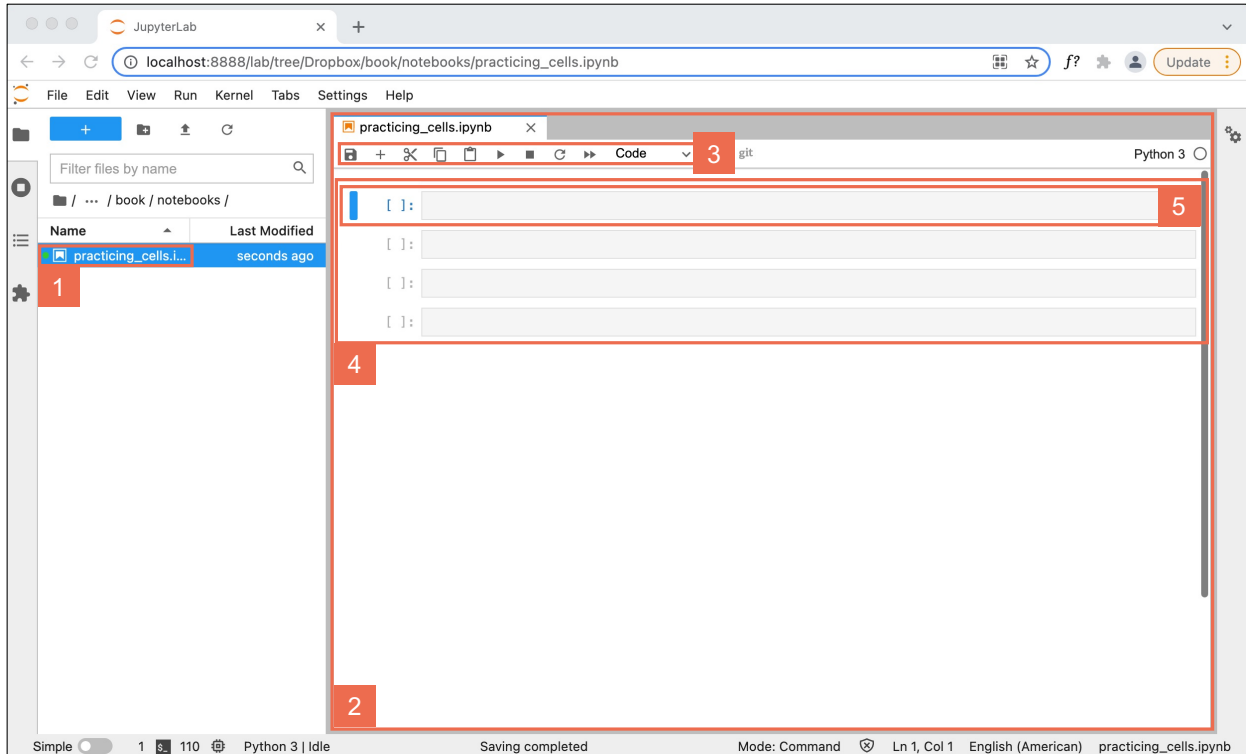


Figure 5. A Jupyter Notebook opened in JupyterLab. (1) Notebook in the folder browser, (2) Jupyter Notebook, (3) Jupyter Notebook top bar, (4) cells, (5) currently active cell.

- *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 32. 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](http://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
      string'      Everything you write between quotes
                  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

- |  |   |  |
|--|---|--|
| <ol style="list-style-type: none"> <li>1. What's your name? is</li> <li>2. <code>input()</code> is a built-in function and</li> <li>3. When running a cell containing <code>input()</code></li> <li>4. A built-in function is always followed</li> </ol> |  | <ol style="list-style-type: none"> <li>a. it is colored green</li> <li>b. by round brackets</li> <li>c. a string</li> <li>d. we can answer a question</li> </ol> |
|--|---|--|

## 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? 
```

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? 
```

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. <code>print()</code> is a string                      | T | F |
| 2. <code>print()</code> 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]: 1 print ("/\_/\"
      2 print (>^.^< ")
      3 print (" / \"
      4 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]: 1 "/\_/\"
      2 ">^.^< "
      3 " / \"
      4 "(__)_"

'(__)_'
```

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]: 1 print ("(__)_"
      2 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 the figure in the next page. 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.

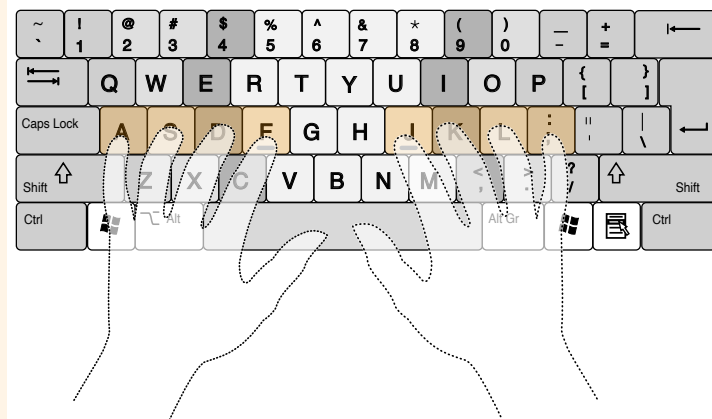


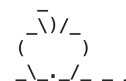
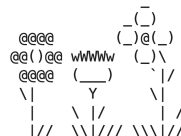
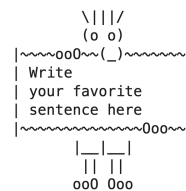
Figure modified from <https://commons.wikimedia.org/w/index.php?curid=9666341>.  
By Cy21 - Own work, CC BY-SA 3.0.

There are plenty of websites to learn touch typing in a fun way, such as [www.typing.com](http://www.typing.com) and [www.typingclub.com](http://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?

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:

### REGISTRATION FORM

first\_name = \_\_\_\_\_

last\_name = \_\_\_\_\_

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?

- The command `first_name = "Fernando"` assigns the string "Fernando" to the variable `first_name` T    F
- The command `print(first_name)` will print out Fernando T    F
- 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?")    favorite_food is assigned input what's
   your favorite food?
```

```
[1]: 1 print ("Hi! My name is " + name)           print Hi! My name is concatenated with
      2 print ("My favorite food is " +           name
      favorite_food)                             print My favorite food is concatenated
      2 print (name + "'s favorite food is " +    with favorite_food
      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 favorite 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?")       name is assigned input what's your name?
      What's your name? Serena
```

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?")                       favorite_food is assigned input what's
      What's your favorite food? pasta        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)       print Hi! My name is concatenated with
      2 print ("My favorite food is " +           name
      favorite_food)                             print My favorite food is concatenated
      2 print (name + "'s favorite food is " +    with favorite_food
      favorite_food)                             print name concatenated with 's favorite
      food is concatenated with favorite_food
      Hi! My name is Serena
      My favorite food is pasta
      Serena's favorite food is pasta
```

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?

- On the programming shelf there are 2 books T F
- If the customer wants a book that is in the programming shelf, you print: Yes, we sell it! T F
- 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)
```

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

```
['Learn Python', 'Python for all', 'Intro to Python']
```

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]: | <pre>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</pre> | <pre>wanted book is assigned input Hi! What book would you like to buy? print wanted book</pre> |
|------|---|---|

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]: | <pre>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!</pre> | <pre>if wanted book in books print "Yes, we sell it!" else print "Sorry, we do not sell that book"</pre> |
|------|---|--|

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]: | <pre>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</pre> | <pre>wanted book is assigned input Hi! What book would you like to buy? print wanted book</pre> |
|------|---|---|

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

|      |  |  |
|------|--|--|
| [5]: | <pre>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</pre> | <pre>if wanted book in books print Yes, we sell it! else print Sorry, we do not sell that book</pre> |
|------|--|--|

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" |                         |                                   |
| <b>if</b>      |                         |                                   |
| <b>in</b>      |                         |                                   |
| <b>else</b>    |                         |                                   |
| =              |                         |                                   |
| +              |                         |                                   |
| 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"] shopping list is assigned
      2 print (shopping_list) 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?") new item is assigned input What
      2 if new_item in shopping_list: else do I have to buy?
      3     print (new_item + " is/are already in the if new item in shopping list
      shopping list") print new item concatenated with
      is/are already in the shopping
      4     print (shopping_list) list
      5 else: print shopping list
      6     shopping_list.append(new_item) else
      7     print (shopping_list) 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 item to remove is assigned input
      remove?") what do I have to remove?
      2 if item_to_remove in shopping list: if item to remove in shopping
      list
      3     shopping_list.remove(item_to_remove) shopping list dot remove item to
      remove
      4     print (shopping_list) print shopping list
      5 else: else
      6     print (item_to_remove + " is/are not in the print item to remove concatenated
      shopping list") with is/are not in the shopping
      list
      7     print (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

- |   |   |  |
|---|---|--|
| <ol style="list-style-type: none"> <li>1. The variable <code>shopping_list</code> contains</li> <li>2. If the new item is not in the shopping list</li> <li>3. If the item to remove is in the shopping list</li> <li>4. The method <code>.append()</code> allows us</li> <li>5. The method <code>.remove()</code> allows us</li> </ol> |  | <ol style="list-style-type: none"> <li>a. we remove it from the shopping list</li> <li>b. to remove an element from a list</li> <li>c. "carrots", "chocolate", and "olives"</li> <li>d. we add it to the shopping list</li> <li>e. to add an element at the end of a list</li> </ol> |
|---|---|--|

## 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
      4         shopping list")
      5     print (shopping_list)
      6 else:
      7     shopping_list.append(new_item)
      8     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']
```

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
      2 remove?")
      3 if item_to_remove in shopping_list:
          shopping_list.remove(item_to_remove)
          print (shopping_list)
      4 else:
          print (item_to_remove + " is/are not in the
          shopping list")
          print (shopping_list)
      5
      6
      7
      What do I have to remove? olives
      ['carrots', 'chocolate', 'apples']
```

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
      2 remove?")
      3 if item_to_remove in shopping_list:
          shopping_list.remove(item_to_remove)
          print (shopping_list)
      4 else:
          print (item_to_remove + " is/are not in the
          shopping list")
          print (shopping_list)
      5
      6
      7
      What do I have to remove? grapes
      grapes is/are not in the list
      ['carrots', 'chocolate', 'apples']
```

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.

| <i>Punctuation symbol</i> | <i>What it's called</i>        | <i>What it does</i>    |
|---------------------------|--------------------------------|------------------------|
| ' ' 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
      2 to remove?")
      3 if item_to_remove in shopping list:
      4     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:

|   |  |
|---|--|
| <pre>[4]: 1 item_to_remove = input ("What do I have     2 to remove?")     3 if item_to_remove in shopping list:     4     shopping_list.remove(item_to_remove)     5     6     print (shopping_list)     7     8     What do I have to remove? olives     9     ['carrots', 'chocolate', 'apples']</pre> | <pre>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</pre> |
|---|--|

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"]    today's menu is assigned burger, salad,
      2  print (todays_menu)                       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")  side dish index is assigned today's
      2  print (side_dish_index)                    menu dot index of salad
                                           print side dish index
```

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

```
[ ]: 1  todays_menu.pop(side_dish_index)            today's menu dot pop side_dish_index
      2  print (todays_menu)                       print today's menu
```

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

```
[ ]: 1  todays_menu.insert(side_dish_index, "fries")  today's menu dot insert at side dish
      2  print (todays_menu)                       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"]  today's menu is assigned burger, salad,
      2  print (todays_menu)                    coke
      ['burger', 'salad', 'coke']              print today's menu
```

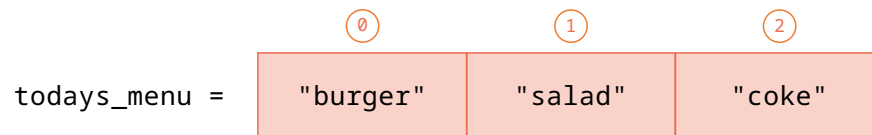
We create a list called `todays_menu` containing three elements of type string—"burger", "salad", and "coke" (line 1)—and we print it out (line 2).

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")  side dish index is assigned today's
      2  print (side_dish_index)                    menu dot index of salad
      1                                              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 the figure below: "burger" is in position 0, "salad" in position 1, and "coke" in position 2.



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)  today's menu dot pop side_dish_index
      2  print (todays_menu)              print today's menu
      ['burger', 'coke']
```

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")  today's menu dot insert at side dish
      2  print (todays_menu)                       index fries
      ['burger', 'fries', 'coke']                 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 |
|------------------------|--------------|
| <code>.append()</code> |              |
| <code>.remove()</code> |              |
| <code>.index()</code>  |              |
| <code>.pop()</code>    |              |
| <code>.insert()</code> |              |

## 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 "shou1d" 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:



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
      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
      ['Prague', 'Tokyo']      with a step of two
```

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])      print cities in positions from one to four
                                     with a step of one
      ['Prague', 'Cape Town', 'Tokyo']
```

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])      print cities in positions from zero to three
      ['San Diego', 'Prague', 'Cape Town']
```

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])      print cities from the beginning of the list
                                     to position three
      ['San Diego', 'Prague', 'Cape Town']
```

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

```
[7]: 1 print (cities[2:5])      print cities in positions from two to five
      ['Cape Town', 'Tokyo', 'Melbourne']
```

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:])      print cities from position two to the end of
                                     the list
      ['Cape Town', 'Tokyo', 'Melbourne']
```

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])
      ['San Diego', 'Cape Town', 'Melbourne']
```

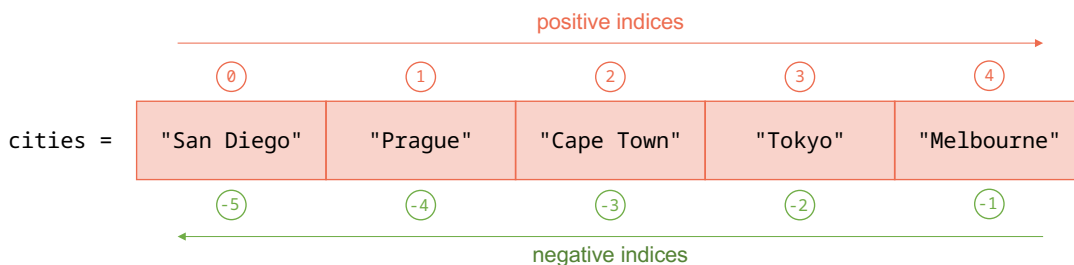
`print` cities in positions from `zero` to `five` with a step of `two`

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

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

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

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



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

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

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

## 7. Slice "Melbourne":

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

`print` cities in positions 4

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

```
[12]: 1 print (cities[-1])      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
      ['Prague', 'Cape Town', 'Tokyo']  minus one
```

This is 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 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
      ['Tokyo', 'Cape Town', 'Prague']  with a step of minus one
```

When slicing—and coding, in general—it is extremely important to be aware of the result we expect. When slicing in reverse order, having the result in mind can really avoid confusion. So, let's start from there. We want to print 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])
```

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

```
['Tokyo', 'Cape Town', 'Prague']
```

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])
```

print from the beginning of the list to the end of the list with a step of minus 1

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

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:

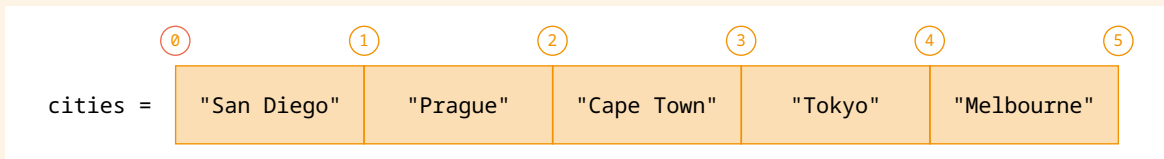
| <i>Slicing syntax</i>                   | <i>What it does</i> |
|---|---------------------|
| <code>list_name[index]</code>           |                     |
| <code>list_name[start:stop:step]</code> |                     |
| <code>list_name[:stop:step]</code>      |                     |
| <code>list_name[start::step]</code>     |                     |
| <code>list_name[start:stop]</code>      |                     |
| <code>list_name[negative_index]</code>  |                     |
| <code>list_name[::negative_step]</code> |                     |
| <code>list_name[::-1]</code>            |                     |

## 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 this figure:



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])           print cities in positions from one
      ['Prague', 'Cape Town', 'Tokyo'] 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:

- The produce between apples and grapes (included)
  - All the vegetables
  - All the fruits
  - The vegetables between carrots and asparagus (included)
  - 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:

- All the clothes
  - All the stationery
  - All the electronics
  - The second and the last stationery items
  - The first and the last electronics items
  - 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:
- All furniture
  - All textiles
  - All decorative elements
  - 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):

- All flowers
- All trees
- All flowers and trees starting with *p* (find them by hand, no coding required)
- "pine", "rose", and "eucalyptus"
- 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](http://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"]
      2 print (senses)
      ['eyes', 'nose', 'ears', 'tongue', 'skin']
```

senses is assigned eyes, nose, ears, tongue, skin  
print senses

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"
      2 print (senses)
      ['eyes', 'smell', 'ears', 'tongue', 'skin']
```

senses in position one is assigned smell  
print senses

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")
      2 senses.pop(nose_index)
      3 senses.insert(nose_index, "smell")
```

nose index is assigned senses dot index of nose  
senses dot pop nose index  
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]: | <pre>1 planets = planets[:5] + ["Uranus", "Saturn"] +   planets[5:] 2 print (planets)</pre> | <pre>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</pre> |
|      | <pre>['Mercury', 'Mars', 'Venus', 'Earth', 'Neptune', 'Uranus', 'Saturn', 'Jupiter']</pre>  |   |

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 we want to delete every second element (line 1). Finally, we print the remaining list (line 2).

- Delete "house":

```
[13]: 1 del house
      2 print (house)
```

```
del house
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
```

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?

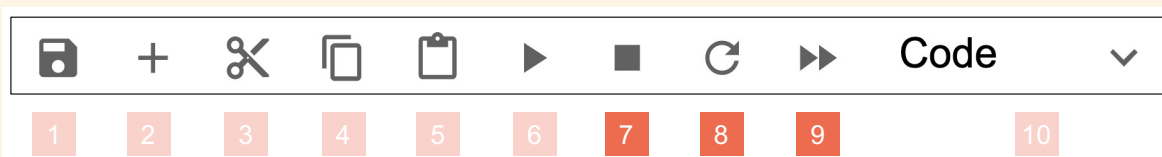
The kernel is the component of Jupyter Notebook that executes code. When we run a cell, the kernel tells Python to execute computations and save variables. Every Notebook has its own kernel. And when we open a Notebook, a new kernel is automatically created and is ready to execute code. Now you may ask: 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?")      name is assigned input what's your
      What's your name?       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 the figure below (modified from the *Getting ready* part):



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 the figure above). 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 the figure above). 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_barnerslee = ["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

- |  |   |  |
|--|---|--|
| <ol style="list-style-type: none"> <li>1. The for loop allows us</li> <li>2. The variable <code>index</code></li> <li>3. In the first loop, the variable <code>index</code></li> <li>4. The built-in function <code>range()</code> determines</li> <li>5. The built-in function <code>range()</code> can take</li> </ol> |  | <ol style="list-style-type: none"> <li>a. a <i>start</i> and a <i>stop</i> as an argument</li> <li>b. how many times commands are repeated</li> <li>c. to repeat the indented commands</li> <li>d. changes value at each loop</li> <li>e. is assigned the value 0</li> </ol> |
|--|---|--|

## Computational thinking and syntax

Let's start by running the first cell:

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

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

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

Let's run the second cell:

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

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

print My friends' names are:  
print friends

We print 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])
```

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

for index in range from zero to four  
print index: concatenated with string of index  
print friend: concatenated with friends in position index

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

- **for**: The keyword starting a for loop. Like all keywords, it is bold green in Jupyter Notebook.
- **index**: A variable that is assigned a different value at each loop iteration (we'll talk more about this in a bit).
- **in**: A membership operator, the same that you learned in the construct `if...in/else` in Chapter 3.

- `range()`: A built-in Python function. You can recognize this as a function because it is followed by round brackets and is colored green in Jupyter Notebook—like `input()` and `print()`. We'll talk more about `range()` in a bit too.
- `:` that is, the colon punctuation.

To better understand what this line does, let's begin from the built-in function `range()`. It takes two arguments: `0` and `4`. They are two integers that we can call—guess what?—start and stop! So, what does `range()` do? Create a separate cell in the notebook, and 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   | <code>for index in range(0,4):</code> | <code>print("index:"+str(index))</code> | <code>print("friend:"+friends[index])</code> |
|--------|---------------------------------------|---|--|
| First  | <code>index = 0</code>                | <code>index: 0</code>                   | friend: friends[0] → Geetha                  |
| Second | <code>index = 1</code>                | <code>index: 1</code>                   | friend: friends[1] → Luca                    |
| Third  | <code>index = 2</code>                | <code>index: 2</code>                   | friend: friends[2] → Daisy                   |
| Fourth | <code>index = 3</code>                | <code>index: 3</code>                   | 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)
      print Their favorite dishes are:
      print dishes
      Their favorite dishes are:
      ['sushi', 'burgers', 'tacos', 'pizza']
```

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] +
      3         "'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", "mississippi", "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",  
2         "dolphin"]  
      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  
4         # what you really wanted to see  
5         if animals[i] == wanted_to_see:  
6             # print out that that's the animal  
7             # you really wanted to see  
8             print ("I saw a " + animals[i] +  
9                   " and I really wanted to see it!")  
6         else:  
7             # just print out what you saw  
8             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]: | <pre>1 animals = ["giraffe", "penguin", 2           "dolphin"] 3 print (animals) 4 ['giraffe', 'penguin', 'dolphin']</pre> | <pre>animals is assigned giraffe, penguin, dolphin print animals</pre> |
|------|--|--|

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]: | <pre>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 " + 6           str(i) + " is " + animals[i])</pre> | <pre>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</pre> |
|      | <pre>--- 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</pre>  |  |

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 a 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
      4     # what you really wanted to see
      5     # if animals[i] == wanted_to_see:
      6         # print out that that's the animal
      7         # you really wanted to see
      8         print ("I saw a " + animals[i] +
      9         " and I really wanted to see it!")
      10
      11     else:
      12         # just print out that you saw it
      13         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

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 statement at line 6, and we jump directly to the statement under the else, which is at line 9. There, we print "I saw a dolphin".

## Complete the table

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

| <i>Expression</i>                              | <i>Meaning</i> |
|--|----------------|
| To run a cell<br>(Chapter 1)                   |                |
| To write readable code<br>(Chapter 3)          |                |
| The function takes one argument<br>(Chapter 5) |                |
| The function returns an integer<br>(Chapter 5) |                |
| To reassign to a variable<br>(Chapter 7)       |                |
| The element is hard-coded<br>(Chapter 8)       |                |
| To comment code<br>(Chapter 9)                 |                |
| To hard-code<br>(Chapter 9)                    |                |
| To implement code<br>(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",
2         "sunglasses", "ring"]
      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))
```

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

```
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
```

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,
6         # and its number of characters
7         print ("The element " + accessories[i] +
8               " is in position" + str(i)) +
9               " and it has 6 characters")
10
11 for each position in the list
12 for i in range len of accessories
13 if the length of the element equals
14 six
15 if len of accessories in position i
16 equals six
17 print the element, its position, and
18 its number of characters
19 print The element concatenated
20 with accessories in position i
21 concatenated with is in position
22 concatenated with string of i
23 concatenated with and it has six
24 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(accessories[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
4     # than 6
5     if len(accessories[i]) < 6:
6         # print the element, its position,
7         # and its number of characters
8         print ("The element " + accessories[i] +
9               " is in position" + str(i)) +
10              " and it has less than 6 characters")
11
12 for each position in the list
13 for i in range len of accessories
14 if the length of the element is less
15 than six
16 if len of accessories in position i
17 less than 6
18 print the element, its position, and
19 its number of characters
20 print The element concatenated
21 with accessories in position i
22 concatenated with is in position
23 concatenated with string of i
24 concatenated with and it has less
25 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
      6     # than the threshold
      7     if len(accessories[i]) > n_of_characters:
      8         # print the element, its position,
      9         # and its number of characters
      10        print ("The element " + accessories[i] +
      11                " is in position" + str(i) +
      12                " and it has more than " +
      13                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

|   |  |
|---|--|
| <pre> 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)) </pre> | <pre> 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 </pre> |
| <pre> 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 </pre>   |  |

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:

|   |  |
|---|--|
| <pre> [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 &lt;= 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)) </pre> | <pre> 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 </pre> |
| <pre> 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 </pre>   |  |

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
      6     # than of equal to the threshold
      7
      8     if i >= position:
      9
      10        # print the element, its position,
      11        # and its position characteristic
      12        print ("The element " + accessories[i] +
      13              " is in position" + str(i) +
      14              ", 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()

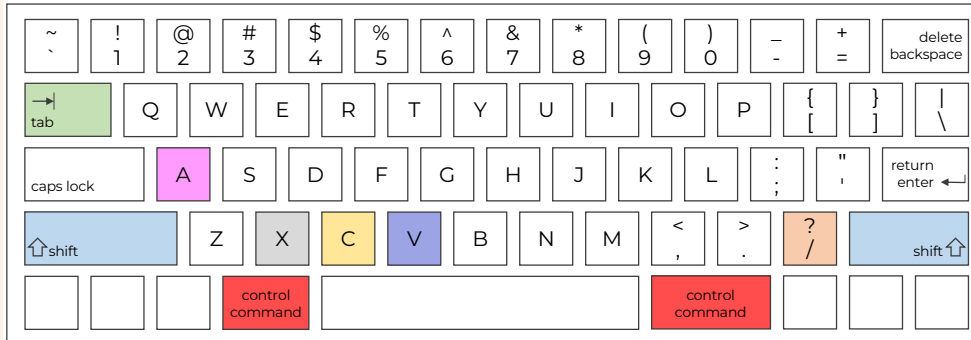
| Data types | Built-in functions | Keywords | List methods |
|------------|--------------------|----------|--------------|
|            |                    |          |              |
|            |                    |          |              |
|            |                    |          |      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

## 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 the figure below.



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 the 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 the figure above). 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:

```
[ ]: 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

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

```
[ ]: 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[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. <code>print(greeting.lower())</code>      |  | a. 'HELLO, HOW ARE YOU?' |
| 2. <code>print(greeting.upper())</code>      |   | b. 'Hello, how are you?' |
| 3. <code>print(greeting.title())</code>      |   | c. 'hello, how are you?' |
| 4. <code>print(greeting.capitalize())</code> |   | 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()      sports in position 1 is assigned
                                           sports in position 1 dot upper
2 mountain_sports.append(sports[1])      mountain_sports dot append sports
                                           in position 1
3 print(sports)                          print sports
4 print(mountain_sports)                  print mountain_sports
['diving', 'HIKING']
['CLIMBING', 'HIKING']
```

In this example, we first change the element in position 1 to uppercase (line 1), and then we append the changed element to the list `mountain_sports` (line 2). When we print 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:

|  |   |
|--|---|
| <pre>[3]: 1 current_sport = sports[1].upper()       2 mountain_sports.append(current_sport)       3 print(sports)       4 print(mountain_sports)       ['diving', 'hiking']       ['CLIMBING', 'HIKING']</pre> | <pre>current_sport is assigned sports in position 1 dot upper mountain_sports dot append current_sport print sports print mountain_sports</pre> |
|--|---|

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:

|  |   |
|--|---|
| <pre>[3]: 1 mountain_sports.append(sports[1].upper())       2 print(sports)       3 print(mountain_sports)       ['diving', 'hiking']       ['CLIMBING', 'HIKING']</pre> | <pre>mountain_sports dot append sports in position 1 dot upper() print sports print mountain_sports</pre> |
|--|---|

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: " +
10 authors[i])
11
12     # get the initial of the current author
13     author_initial = authors[i][0]
14
15     print ("The author's initial is: " +
16 author_initial)
17
18     # if the author's initial is A
19     if author_initial == "A":
20         # add the author to the shelf a
21         shelf_a.append(authors[i])
22         print ("The shelf A now contains: " +
23 str(shelf_a) + "\n")
24
25     # otherwise (author's initial is not A)
26     else:
27         # add the author to the shelf s
28         shelf_s = shelf_s + [authors[i]]
29
30         print ("The shelf S now contains: " +
31 str(shelf_s) + "\n")
```

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

|   |  |
|---|--|
| <pre>27 # print out the final shelves 28 print ("The authors on the shelf A are: " +       str(shelf_a) 29 print ("The authors on the shelf S are: " +       str(shelf_s)</pre> | <pre>print out the final shelves print The authors on the shelf A are: concatenated with str of shelf a print The authors on the shelf S are: concatenated with str of shelf s</pre> |
|---|--|

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 <code>.append()</code> and list concatenation perform two different actions | T | F |
| 4. The special character <code>"\n"</code> creates an empty line after a print            | T | F |

### Computational thinking and syntax

Let's run the first cell:

|   |   |
|---|---|
| <pre>[1]: 1 authors = ["Alcott", "Saint-Exupéry",       "Arendt", "Sepulveda", "Shakespeare"]</pre> | <pre>authors is assigned Alcott, Saint-Exupéry Arendt, Sepulveda, Shakespeare</pre> |
|---|---|

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:

|   |  |
|---|--|
| <pre>[2]: 1 # initialize the variables as empty lists       2 shelf_a = []       3 shelf_s = []</pre> | <pre>initialize the variables as empty lists shelf a is assigned an empty list shelf s is assigned an empty list</pre> |
|---|--|

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:

|  |   |
|--|---|
| <pre>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])</pre> | <pre>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</pre> |
|--|---|

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:

|  |  |
|--|--|
| <pre> 11 # get the initial of the current author 12 author_initial = authors[i][0]  13 print ("The author's initial is: " +     author_initial) </pre> | <pre> 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 </pre> |
|--|--|

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:

|   |  |
|---|--|
| <pre> 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: " + 20           str(shelf_a) + "\n") 21 22 # otherwise (author's initial is not A) 23 else: 24     # add the author to the shelf s 25     shelf_s = shelf_s + [authors[i]] 26 27     print ("The shelf S now contains: " + 28           str(shelf_s) + "\n") </pre> | <pre> 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 </pre> |
|---|--|

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:

|   |  |
|---|--|
| <pre>27 # print out the final shelves 28 print ("The authors on the shelf A are: " +       str(shelf_a) 29 print ("The authors on the shelf S are: " +       str(shelf_s)</pre> | <pre>print out the final shelves print The authors on the shelf A are: concatenated with str of shelf a print The authors on the shelf S are: concatenated with str of shelf s</pre> |
|---|--|

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. <pre>for i in range (len(hot_drinks)):     print (hot_drinks[i])</pre>  | a. ['TEA', 'coffee',<br>'hot chocolate'] | ★.create list elements              |
| 2. <pre>for i in range (len(hot_drinks)):     if hot_drinks[i][0] == "c":         print (hot_drinks[i])</pre>  | b. tea<br>coffee<br>hot chocolate        | ♣.change list elements              |
| 3. <pre>for i in range (len(hot_drinks)):     if len(hot_drinks[i]) == 3:         hot_drinks[i] = hot_drinks[i].upper()     print (hot_drinks)</pre>           | c. ['coffee', 'hot chocolate']           | ■.print list elements<br>one by one |
| 4. <pre>long_words = [] for i in range (len(hot_drinks)):     if len(hot_drinks[i]) &gt;= 6:         long_words.append(hot_drinks[i]) print (long_words)</pre> | d. coffee                                | ▲.find list elements                |
- 

## Recap

- To create and fill a list in a for loop, we have to: (1) initialize an empty list *before* the for loop and (2) fill the list using `.append()` or list concatenation *in* the for loop
- String slicing works the same way as list slicing
- In multiple consecutive slicings, we execute one slicing at a time, starting from the left
- The 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
      8     # if the author's initial is A
      9     if author_initial == "A":
      10        # add the author to the shelf a
      11        shelf_a = authors[i]
      12
      13        print ("The shelf A now
      14                contains: " + str(shelf_a))
      15 # print out the final shelves
      16 print ("The authors on the shelf A are: "
      17        + str(shelf_a))
      The shelf A now contains: Alcott
      The shelf A now contains: Arendt
      The authors on the shelf A are: Arendt
```

Let's go through the for loop 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", "FR-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 and algorithms! There is a general misconception that in order to be good at coding one has to be 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 **remainder** (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 ( $4 \times 1 = 4$ ). 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:")
      2 print (number)
      Insert a number: 9
      9
      number is assigned input Insert a number:
      print number
```

We use the built-in function `input()` to ask the user to type a number, and we save the answer in the variable `number` (line 1). Then, we print 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)
      str
      type number
```

To know the type of a variable, we use the built-in function `type()`, which **takes a variable as an input and returns its type**. In the printout, we see that the type of `number` is `str`, which is an abbreviation for string. But shouldn't 9 be an integer? Yes! However, `number` is a string because the built-in function `input()` **returns strings**, regardless of what a user types on a keyboard (characters, numbers, or symbols). To transform the value of `number` into an actual number that we can use in calculations, we have to transform its type from string to integer.

- Transform `number` into an integer, print it out, and check its type:

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

The built-in function `int()` **takes a non-integer variable as an input and returns it as an integer**. Note that to actually transform a variable type, we need to **reassign** the output of the built-in function `int()` to the variable itself (line 1). At line 2, we print `number`, which is still 9. However, this time `number` is of type `int`, as we can see from `type(number)` at line 3. What if we want a decimal number? In that case, we have to transform the variable type into `float`!

- Transform `number` into a float, print it out, and check its type:

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

The built-in function `float()` **takes a non-decimal variable as an input and returns it as a decimal**. Also in this case, we need to reassign the output of `float()` to the variable itself to actually change the data type (line 1). From the print at line 2, we see that the variable `number` is now `9.0`, that is, a

decimal number. And from the command at line 3, we 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)      number is assigned str of number
      2 print (number)          print number
      3 type (number)           type number
      9.0
      str
```

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
      2         operator:")
      3 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
      2         second number:"))
      3 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
```

|   |   |
|---|---|
| <pre> 9 elif operator == "/": 10     result = first_number / second_number 11 elif operator == "//": 12     result = first_number // second_number 13 elif operator == "%": 14     result = first_number % second_number 15 else: 16     print ("You didn't enter an 17     arithmetic operator") 18 print (result) 19 7.0 </pre> | <pre> 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 result </pre> |
|---|---|

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:

|  |   |
|--|---|
| <pre> [17]: 1 print (str(first_number) + " " + operator       + " " + str(second_number) + " = " +       str(result)) </pre> | <pre> 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 </pre> |
| <pre>4.0 + 3.0 = 7.0</pre>   |   |

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]: 1 # first input
2 first_number = float(input("Insert the
first number:"))
3
4 # operator
5 operator = input("Insert an arithmetic
operator:")
6
7 # second input
8 second_number = float(input("Insert the
second number:"))
9
10 # computations
11 if operator == "+":
12     result = first_number + second_number
13
14 elif operator == "-":
15     result = first_number - second_number
16
17 elif operator == "*":
18     result = first_number * second_number
19
20 elif operator == "**":
21     result = first_number ** second_number
22
23 elif operator == "/":
24     result = first_number / second_number
25
26 elif operator == "//":
27     result = first_number // second_number
28
29 elif operator == "%":
30     result = first_number % second_number
31
32 else:
33     print ("You didn't enter an
arithmetic operator")
34
35 # print the result
36 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:

|      |   |             |   |
|------|---|-------------|---|
| [1]: | 1 | (6 + 2) * 3 | open round bracket six plus two close round bracket times three |
|      |   | 24          |   |

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:  $area = 2\pi r^2 + 2\pi r h$  and  $volume = \pi r^2 h$ . *Hint:* What is the value of  $\pi$ ? 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:

- The temperature in Miami is  $75^\circ\text{F}$ . What is the temperature in Celsius?
- The temperature in Lisbon is  $17^\circ\text{C}$ . 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 for each position in the list
2 for i in range (len(numbers)): for i in range len of numbers
3
4 # if current number >= 10 if current number is greater than or equal to ten
5 if numbers[i] >= 10: if numbers in position i is greater than or equal
6 # subtract 1 subtract one
7 numbers[i] = numbers[i] - 1 numbers in position i is assigned numbers in
8 # otherwise otherwise
9 # otherwise otherwise
10 else: else:
11 # add 2 add two
12 numbers[i] = numbers[i] + 2 numbers in position i is assigned numbers in
13 # print the final result print the final result
14 print (numbers) print numbers
[11, 5, 14, 9, 17]
```

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 initialize empty lists
2 even = [] even is assigned an empty list
3 odd = [] odd is assigned an empty list
4
5 # for each position in the list for each position in the list
6 for i in range (len(numbers)): for i in range len of numbers
7
8 # if the current number is even if the current number is even
9 if numbers[i] % 2 == 0: if numbers in position i modulo two equals zero
10 # add it to the list even add it to the list even
11 even.append(numbers[i]) even dot append numbers in position i
12 # otherwise otherwise
13 else: else:
14 # add it to the list odd add it to the list odd
15 odd.append(numbers[i]) odd dot append numbers in position i
16
17 # check the final results check the final results
18 print (even) print even
19 print (odd) print odd
[2,10,0]
[7,5,9]
```

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
      2 first element of the list
      3 maximum = numbers[0]
      4
      5 # for each position in the list
      6 starting from the second
      7 for i in range (1, len(numbers)):
      8
      9 # if the current number is greater
      10 than the current maximum
      11     if numbers[i] > maximum:
      12
      13         # assign the number to maximum
      14         maximum = numbers[i]
      15
      16 # print the maximum of the list
      17 print (maximum)
      18
      19 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
      2 print (len)
      10
```

len is assigned ten  
print len

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)

-----
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
```

numbers is assigned one, two, three  
len numbers

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)
    2
    Land is always on the mind of a flying bird
```

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

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 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
      2 message = random.choice(fortune_cookies)
      3 print(message)
      The man on the top of the mountain did not fall there
```

pick a message  
message is assigned random dot  
choice fortune cookies  
print message

In this case, we use another function from the module `random` called `.choice()`, which takes a list as an input and returns a randomly selected element of the list (line 2). Finally, we print the message (line 3).

What is the difference between `.choice()` and `.randint()`? When using `.choice()`, we do not know the *position* of the element the computer randomly selects, whereas when using `.randint()`, we know where the element is in the list.

### Match the sentence halves

- |  |   |  |
|--|---|--|
| <ol style="list-style-type: none"> <li>1. In <code>range(start, stop, step)</code></li> <li>2. In <code>.randint(a,b)</code></li> <li>3. The function <code>.randint(a,b)</code></li> <li>4. The function <code>.choice(list)</code></li> <li>5. The syntax to use a string or list method is</li> <li>6. The syntax to use a function from a module is</li> </ol> |  | <ol style="list-style-type: none"> <li>a. module name, dot, function name()</li> <li>b. returns a random element from a list</li> <li>c. stop is excluded</li> <li>d. variable name, dot, method name()</li> <li>e. the endpoint <code>b</code> is included</li> <li>f. returns a random integer between <code>a</code> and <code>b</code> (included)</li> </ol> |
|--|---|--|

## 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:

|   |  |
|---|--|
| <pre>[1]: 1 import random       2       3 # repeat the commands 3 times (index not       4 # needed)       5 for _ in range (0,3):       6       7     # create a random number between 10       8     # and 20       9     random_number = random.randint(10,20)      10      11     # print the number      12     print ("The random number is" +      13           random_number)      14      15 The random numbers is: 14      16 The random numbers is: 17      17 The random numbers is: 12</pre> | <pre>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</pre> |
|---|--|

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<sup>1</sup>. 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 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`.

<sup>1</sup>[https://en.wikipedia.org/wiki/Rock\\_paper\\_scissors](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
      3 scissors?")
      print(player_choice)
      rock, paper, or scissors? rock
      rock
```

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!
```

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

Also in this last case, the code structure is similar: an if condition (line 1) nesting an if/elif/else construct (lines 4–9). We check if the computer picked "scissors" and if the player chose "scissors" (line 4), "rock" (line 6), or "paper" (the else in line 8). As in cell 4, we construct the conditions so that the print statements are **parallel** to 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
6  # computer random pick
7  computer_pick = random.choice(possibilities)
8
9  # asking the player to make their choice
10 player_choice = input("Rock, paper, or
11 scissiors?")
12
13 # determine who wins
14 # if the computer picks rock
15 if computer_pick == "rock":
16     # compare to the player's choice
17     if player_choice == "rock":
18         print("Tie!")
19     elif player_choice == "paper":
20         print("You win!")
21     else:
22         print("The computer wins!")
23
24 # if the computer picks paper
25 if computer_pick == "paper":
26     # compare to the player's choice
27     if player_choice == "paper":
28         print("Tie!")
29     elif player_choice == "scissiors":
30         print("You win!")
31     else:
32         print("The computer wins!")
33
34 # if the computer picks scissiors
35 if computer_pick == "scissiors":
36     # compare to the player's choice
37     if player_choice == "scissiors":
38         print("Tie!")
39     elif player_choice == "rock":
40         print("You win!")
41     else:
42         print("The computer wins!")
43
44 Rock, paper, or scissiors? rock
45 You win!
```

When merging code, we usually do some **editing to improve code use and readability**. In this case, we erased the print of `computer_pick` (which was in cell 1, line 8) because we do not want the player to know the computer choice in advance. Similarly, we delete the print of `player_choice` (which was in cell 2, line 3), as the player already sees their choice from the entry at line 9. Other

editing might include improving comments, making variable names more meaningful, restructuring parts of the code, etc.

Let's now zoom out and observe the procedure we use to implement the game. We first defined three steps (see the exercise *Complete the sentences*). Then, we implemented each step separately (see 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?

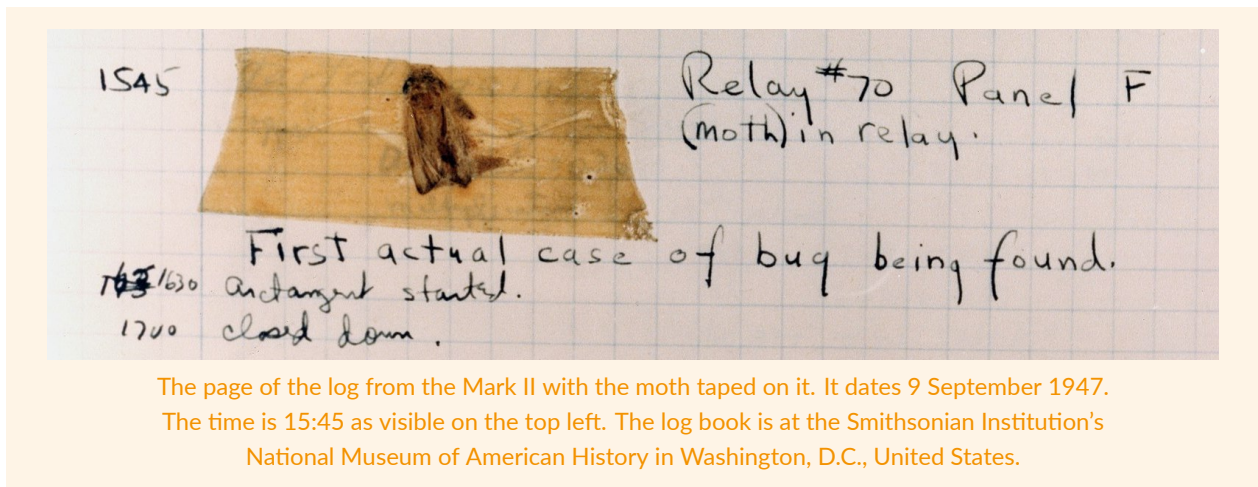
Do you know why we say *debugging*, *divide and conquer*, and *algorithms*? The term *debugging* originated in 1947, when a moth was found in a relay of Mark II computer at Harvard University, causing the computer to malfunction. The moth was then taped to a log sheet, with the annotation *Relay 70 Panel F (moth) in relay. First actual case of a bug being found* (see figure below). Although the word *debugging* is not mentioned in the annotation, it became popular thanks to Grace Hopper, who worked on the same computer<sup>a,b</sup>. *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<sup>c</sup>. 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<sup>d</sup>.

<sup>a</sup><https://en.wikipedia.org/wiki/Debugging>

<sup>b</sup>[https://en.wikipedia.org/wiki/Grace\\_Hopper](https://en.wikipedia.org/wiki/Grace_Hopper)

<sup>c</sup>[https://en.wikipedia.org/wiki/Divide\\_and\\_rule](https://en.wikipedia.org/wiki/Divide_and_rule)

<sup>d</sup><https://en.wikipedia.org/wiki/Algorithm>



## Let's code!

1. *Trivia night!*. Trivia is a quiz game where players have to answer questions about various subjects. For this implementation of Trivia, prepare 3 questions and their corresponding answers for 3 different topics. Ask the player to pick a topic, and then ask a randomly picked question about that topic. Finally, tell the player whether the answer is correct. If not, print 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!

```
[ ]: 1 # initialize variable
      2 number_of_candies = 0
      3
      4 # print the initial number of candies
      5 print("You have " + str(number_of_candies) +
      6 " candies")
      7
      8 # ask if one wants a candy
      9 answer = input ("Do you want a candy?
      10 (yes/no)")
      11
      12 # as long as the answer is yes
      13 while answer == "yes":
      14     # add a candy
      15     number_of_candies += 1
      16
      17     # print the current number of candies
      18     print("You have " + str(number_of_candies)
      19     + " candies")
      20
      21     # ask again if they want more candies
      22     answer = input ("Do you want more
      23     candies? (yes/no)")
      24
      25 # print the final number of candies
      26 print("You have a total of " +
      27 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]: | <pre> 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") </pre> | <pre> 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 </pre> |
|------|---|---|

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:

|  |  |
|--|--|
| <pre> 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) 18         + " candies") 19 20     # ask again if they want more candies 21     answer = input ("Do you want more 22         candies? (yes/no)") 23 24 # print the final number of candies 25 print("You have a total of" + 26     str(number_of_candies) + " candies") </pre> | <pre> 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 </pre> |
| <pre> 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 </pre>   |  |

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!

- The computer randomly picks one of the three animals. This is pretty straightforward:

```
1 import random
2
3 # computer pick
4 computer_pick = random.choice(animals)
5 print(computer_pick)
```

dolphin

```
import random
computer pick
computer pick is assigned random dot
choice animals
print computer pick
```

We import the package `random` (line 1), and we use its function `.choice()` to make the computer pick a random element from the list `animals` (line 4). Then, we print `computer_pick` as a check (line 5).

2. *The player has to guess the animal picked by the computer.* This task is also easy:

|  |  |
|--|--|
| <pre>1 # player guess 2 player_guess = input ("Guess the animal!   Choices: giraffe, dolphin, penguin:")</pre> | <pre>player guess player guess is assigned input Guess the animal! Choices: giraffe, dolphin, penguin:</pre> |
| <pre>Guess the animal! Choices: giraffe, dolphin, penguin: giraffe</pre>                                       |  |

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:

|  |  |
|--|--|
| <pre>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)</pre> | <pre>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</pre> |
| <pre>That's not the right animal! Try again! Guess the animal! Choices: giraffe, dolphin, penguin: dolphin Well done! You guessed dolphin</pre>  |  |

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 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!

|   |   |
|---|---|
| <pre>1 # initializing the counter 2 n_of_attempts = 1 3</pre> | <pre>initializing the counter n of attempts is assigned one</pre> |
|---|---|

|  |   |
|--|---|
| <pre> 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)) </pre> | <pre> 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) </pre> |
|--|---|

```

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:

|   |  |
|---|--|
| <pre> [1]: 1 import random 2 3 # computer pick 4 computer_pick = random.choice(animals) 5 6 # print(computer_pick) 7 8 # player guess 9 player_guess = input ("Guess the animal!   Choices: giraffe, dolphin, penguin:") 10 11 # initializing the counter 12 n_of_attempts = 1 </pre> | <pre> 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 </pre> |
|---|--|

|   |  |
|---|--|
| <pre> 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) </pre> | <pre> 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 </pre> |
| <pre> 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 </pre>  |  |

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`:

|   |   |
|---|---|
| <pre> 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) </pre> | <pre> 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 </pre> |
|---|---|

[7, 9, 3, 2, 3, 0, 9, 6]

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:

|  |  |
|--|--|
| <pre> 1 import random 2 3 # initialize the number list 4 unique_random_numbers = [] 5 </pre> | <pre> import random  initialize the number list unique random numbers is assigned an empty list </pre> |
|--|--|

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

[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):

|  |   |
|--|---|
| <pre> 1 import random 2 3 # initialize the number list 4 unique_random_numbers = [] 5 </pre> | <pre> import random initialize the number list unique random numbers is assigned an empty list </pre> |
|--|---|

|   |   |
|---|---|
| <pre> 6 # as long as the length of the list is not 8 7 while len(unique_random_numbers) != 8: 8 9     # create a random number between 0 and 10 10    number = random.randint(0,10) 11 12    # if the number is already in the list 13    if number in unique_random_numbers: 14        # print that the number is in the list 15        print ("The number " + str(number) + 16              " is already in the list") 17 18    # otherwise 19    else: 20        # add the new number to the list 21        unique_random_numbers.append(number) 22 23 # print the list 24 print(unique_random_numbers) </pre> | <pre> 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 </pre> |
| <pre> 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] </pre>   |   |

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!

|  |  |
|--|--|
| <pre> [2]: 1 import random 2 3 # initialize the number list 4 unique_random_numbers = [] 5 6 # initialize the counter 7 counter = 0 8 9 # as long as the length of the list is not 8 10 while len(unique_random_numbers) != 8: 11 12     # create a random number between 0 and 10 13     number = random.randint(0,10) </pre> | <pre> 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 </pre> |
|--|--|

|   |  |
|---|--|
| <pre> 14 15     # increase the counter by 1 16     counter += 1 17 18     # if the number is already in the list 19     if number in unique_random_numbers: 20         # print that the number is in the list 21         print ("The number " + str(number) + 22                " is already in the list") 23 24     # otherwise 25     else: 26         # add the new number to the list 27         unique_random_numbers.append(number) 28 29 # print the final list and the total amount of 30 # generated numbers 31 print(unique_random_numbers) 32 print("The total amount of generated numbers 33 is: " + str(counter)) </pre> | <pre> increase the counter by one counter is incremented by one  if the number is already in the list if number in unique random numbers: print that the number is in the list print The number concatenated with str number concatenated with is already in the list  otherwise else: add the new number to the list unique random numbers dot append number  print the final list and the total amount of generated numbers print unique random numbers print The total amount of generated numbers is: concatenated with str counter </pre> |
| <pre> 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 </pre>  |  |

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:

|  |   |
|--|---|
| <pre> 1 # ask the user for an integer 2 number = int(input("Enter an integer: ")) 3 </pre> | <pre> ask the user for an integer number is assigned int input Enter an integer: </pre> |
|--|---|



|  |  |
|--|--|
| <pre> 4 # as long as the number is positive 5 while number &gt;= 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 </pre> | <pre> 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: </pre> |
|--|--|

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:

|  |   |
|--|---|
| <pre> 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 &gt;= 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 </pre> | <pre> 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 </pre> |
|--|---|

|   |  |
|---|--|
| <pre> 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)) </pre> | <pre> 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 </pre> |
| <pre> 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 </pre>   |  |

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:

|  |  |
|--|--|
| <pre> [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 number &gt;= 0: 9 10        # if the number is a multiple of 3 11        if number % 3 == 0: 12 13            # add the number to the sum 14            sum_of_numbers += number 15 16        # ask for the next integer 17        number = int(input("Enter another     integer: ")) </pre> | <pre> 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: </pre> |
|--|--|

|  |   |
|--|---|
| <pre> 18 # print the final sum 19 print("The sum of the multiples of 3 is: +     str(sum_of_numbers)) </pre>   | <pre> print the final sum print The sum of the multiples of 3 is: concatenated with str sum of numbers </pre> |
| <pre> 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 </pre> |   |

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:

- |  |  |
|--|--|
| <ol style="list-style-type: none"> <li>1. An <code>if/else</code> construct checks whether a condition is true or false</li> <li>2. A <code>for</code> loop is the repetition of a group of commands</li> <li>3. A <code>while</code> loop is the repetition of a group of commands</li> </ol> | <ol style="list-style-type: none"> <li>a. for a determined number of times</li> <li>b. as long as a condition holds</li> <li>c. and executes code accordingly</li> </ol> |
|--|--|

## 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.

- The computer randomly picks a fruit.

```
[2]: 1  import random                            import random
     2
     3  # computer pick                            computer pick
     4  computer_pick = random.choice(fruits)    computer pick is assigned random
     5                                           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`.

- The player has to guess the fruit picked by the computer.

```
6  # player guess                                player guess
7  player_guess = input ("Guess the fruit!      player guess is assigned input
  Choices: mango, orange, banana: ")          Guess the fruit! Choices: mango,
8                                              orange, banana:
```

We use the built-in function `input()` to ask the player to enter a fruit (line 7).

- 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                      check the player guess
10 if player_guess == computer_pick:           if player guess is equal to
11     print("That's right! The fruit is " +    computer pick
    computer_pick )                            print That's right! The fruit is
10 else:                                       concatenated with computer pick
11     print("Nope! Try again!")              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:

|   |   |
|---|---|
| <pre> 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: ") </pre> | <pre> 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: </pre> |
|---|---|

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
5     and 10
6     if numbers[i] >= 5 and numbers[i] <= 10:
7
8         # print the current number
9         print ("The number " + str(numbers[i])
10        + " 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

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

We use a for loop to browse all the elements in the list (line 2). Then, we check if each number is between 5 and 10 (line 5). To be in between two numbers, a number must be greater than or equal to the smaller number *and* smaller than or equal to the greater number. The two conditions (greater than or equal to *and* smaller than or equal to) must be valid at the same time. To check if two (or more) conditions are valid simultaneously, we join them using the **logical operator and**.

We use the logical operator **and** when we want to check whether **all conditions** are **valid**

Let's look at the syntax. For each condition *both before and after* the logical operator `and`, we have to write: (1) a variable (e.g., `numbers[i]`), (2) a comparison operator (e.g., `>=`), and (3) a term of comparison (e.g., `5`). At the end of the code, we print the numbers that satisfy both conditions (line 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 string of vowels
      2 vowels = "aeiou" vowels is assigned aeiou
      3
      4 # initialize counter initialize counter
      5 counter = 0 counter is assigned zero
      6
      7 # for each position in the message for each position in the message
      8 for i in range(len(message)): for i in range len of message
      9
      10 # if the current element is punctuation # if the current element is
      11 # or vowel punctuation or vowel
      12 if message[i] in punctuation or if message in position i in
      13 message[i] in vowels: punctuation or message in position
      14 # print a message i in vowels
      15 print(message[i] + " is a vowel # print a message
      16 or a punctuation") print message in position i
      17 # increase the counter concatenated with is a vowel or a
      18 counter += 1 punctuation
      19 # print the final amount increase the counter
      20 print("The total amount of punctuation counter is increased by one
      vowels is " + counter) print the final amount
      print(The total amount of punctuation
      or vowels is concatenated with
      counter
```

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

```

i is a vowel or a punctuation
e is a vowel or a punctuation
a is a vowel or a punctuation
! is a vowel or a punctuation
! is a vowel or a punctuation
! is a vowel or a punctuation
The total amount of punctuation or vowels is 9

```

Similarly to what we did for punctuation, we create a string containing vowels (line 2). We also create a counter, which we will use to calculate the number of characters that are punctuation or vowels, and we initialize it to zero (line 5). Then, we get to the core of the solution! We use a for loop to browse all the characters in the string `message` (line 8). **For loops for strings work exactly the same way as for loops for lists.** In the loop body, we check if each character is a punctuation or a vowel by using the membership operator `in` (line 11), which we learned in Chapter 3. More specifically, we check if `message[i]` is in the string `punctuation` or in the string `vowels`. Note that as for the for loop, the membership operator `in` works for strings the same way as it works for lists. Since only one of the conditions can be valid (a character cannot be both a punctuation and a vowel at the same time!), we merge the two conditions—that is, `message[i] in punctuation or message[i] in vowels`—using the **logical operator `or`**.

We use the logical operator `or` when we want to check whether **at least one condition is valid**

The syntax is the same as for the logical operator `and`: 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])
      9
```

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



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]: | <pre> 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]) 9 </pre> | <pre> 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 </pre> |
|------|--|--|

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]:  | <pre> 1 numbers = [1, 4, 7] </pre> | <pre> numbers is assigned one, four, seven </pre> |
| [10]: | <pre> 1 import random 2 </pre>     | <pre> import random </pre>                        |

|  |  |
|--|--|
| <pre> 3 # for five times 4 for _ in range (5): 5 6     # generate a random number between 0 and 10 7     number = random.randint(0, 10) 8 9     # print the number as a check 10    print (number) 11 12    # if the new number is not in numbers 13    if number not in numbers: 14        # add the number to numbers 15        numbers.append(number) 16 17    # print the final list 18    print(numbers) </pre> | <pre> for five times for underscore in range five generate a random number between zero and ten number is assigned random dot randint zero ten print the number as a check print number if the new number is not in numbers if number not in numbers: add the number to numbers numbers dot append number print the final list printnumbers </pre> |
| <pre> 6 6 10 7 9 [1, 4, 7, 6, 10, 9] </pre>  |  |

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](https://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](https://gitlab.com)) and Bitbucket ([bitbucket.org](https://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 a 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 this 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; and c) Print the numbers that are divisible by 3 but not 5. Perform task c) 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.

Dear coder,

Thanks for learning with me!

Visit [www.learnpythonwithjupyter.com](http://www.learnpythonwithjupyter.com) to:

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

I am looking forward to receiving feedback and comments about the book. Write me at [serena@learnpythonwithjupyter.com](mailto:serena@learnpythonwithjupyter.com) and let me know what you think about it!

Best,  
Serena

**Next release: Chapter 20 on July 8, 2023**