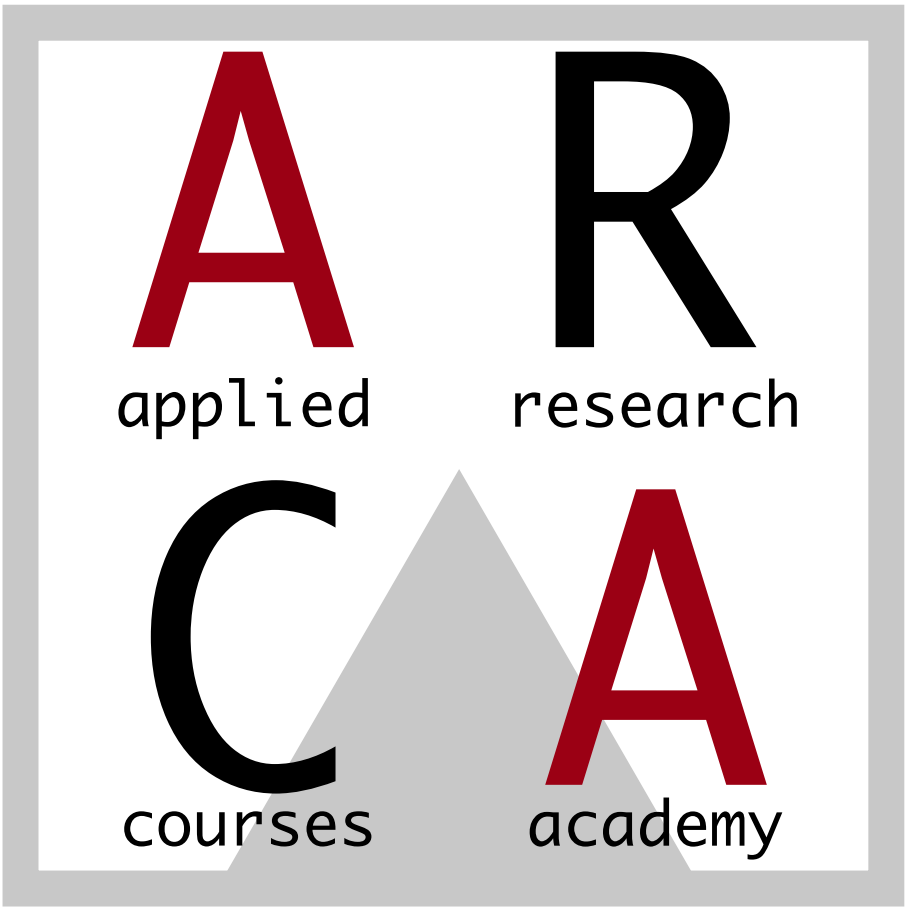
The Open Science Manual

Make Your Scientific Research Accessible and Reproducible

Claudio Zandonella Callegher and Davide Massidda

22 April, 2022 [last-updated]



Contents

Preface

The present book aims to describe programming good practices and introduce common tools used in software development to guarantee the reproducibility of the analysis results. We want to make scientific research open-source knowledge development.

The book is available online at https://arca-dpss.github.io/manual-open-science/.

A PDF copy is available at https://arca-dpss.github.io/manual-open-science/manual-open-science.pdf.

Book Content

In the book, we will learn to:

- Share our materials using the Open Science Framework (**OSF**)
- Learn how appropriately to structure and organize our materials in a **repository**
- Follow recommendations about data organization and data sharing
- Improve code readability and maintainability using a **Functional Style**
- Learn version control and collaboration using **Git** and **Github**
- Manage analysis workflow with dedicated tools
- Create dynamic documents
- Manage project requirements and dependencies using dedicated tools
- Create a container to guarantee reproducibility using **Docker**

This book provides useful recommendations and guidelines that can be applied independently of the specific programming language. However, examples and specific applications are based on the R programming language. Readers working with programming languages other than R can still find valuable guidelines and information and can later apply the same workflow and ideas using dedicated tools specific to their preferred programming language.

Finally, as most researchers have no formal training in programming and software development, we provide a very gentle introduction to many programming concepts and tools without assuming any previous knowledge. Note, however, that we assume that the reader is already familiar with the R programming language for specific examples and applications.

About the Authors

During our careers, we both moved into the field of Data Science after our PhD in Psychological Sciences. This book is our attempt to bring back into scientific research what we have learned outside of academia.

- Claudio Zandonella Callegher. During my PhD, I fell in love with data science. Understanding the complex phenomena that affect our lives by exploring data, formulating hypotheses, building models, and validating them. I find this whole process extremely challenging and motivating. Moreover, I am excited about new tools and solutions to enhance the replicability and transparency of scientific results.
- Davide Massidda.

ARCA

ARCA courses are advanced and highly applicable courses on modern tools for research in Psychology. They are organised by the Department of Developmental and Social Psychology at the University of Padua.

Contribute

Surely there are many typos to fix and new arguments to include. Anyone is welcome to contribute to this book. For small typos just send a pull request with all the corrections. To propose new chapters or paragraphs, instead, open an issue to discuss and plan them.

Acknowledgements

This book was inspired by Richard McElreath's talk *"Science as Amateur Software Development"* https://youtu.be/zwRdO9_GGhY. Talk abstract:

> Science is one of humanity's greatest inventions. Academia, on the other hand, is not. It is remarkable how successful science has been, given the often chaotic habits of scientists. In contrast to other fields, like say landscaping or software engineering, science as a profession is largely *unprofessional* - apprentice scientists are taught less about how to work responsibly than about how to earn promotions. This results in ubiquitous and costly errors. Software development has become indispensable to scientific work. I want to playfully ask how it can become even more useful by transferring some aspects of its professionalism, the day-to-day tracking and back-tracking and testing that is especially part of distributed, open-source software development. Science, after all, aspires to be distributed, open-source knowledge development.

License

1 Introduction

> Science is one of humanity's greatest inventions. Academia, on the other hand, is not. It is remarkable how successful science has been, given the often chaotic habits of scientists. In contrast to other fields, like say landscaping or software engineering, science as a profession is largely *unprofessional* - apprentice scientists are taught less about how to work responsibly than about how to earn promotions. This results in ubiquitous and costly errors. Software development has become indispensable to scientific work. I want to playfully ask how it can become even more useful by transferring some aspects of its professionalism, the day-to-day tracking and back-tracking and testing that is especially part of distributed, open-source software development. Science, after all, aspires to be distributed, open-source knowledge development.
>
> Richard McElreath's "Science as Amateur Software Development" talk
>
> https://youtu.be/zwRdO9_GGhY

Richard McElreath's words are as enlightening as always. Usually, researchers start their academic careers led by their great interest in a specific scientific area. They want to answer some specific research question, but these questions quickly turn into data, statistical analysis, and lines of code, hundreds of lines of code. Most researchers, however, receive essentially no training about programming and software development good practices resulting in very chaotic habits that can lead to costly errors. Moreover, bad

practices may hinder the transparency and reproducibility of the analysis results.

Thanks to the Open Science movement, transparency and reproducibility are recognized as fundamental requirements of modern scientific research. In fact, openly sharing study materials and analyses code are prerequisites for allowing results replicability by new studies. Note the difference between replicability and reproducibility (Nosek & Errington, 2020):

- **Reproducibility**, obtaining the results reported in the original study using the *same data* and the *same analysis*.
- **Replicability**, obtaining the results reported in the original study using *new data* but the *same analysis* (a new study with the same experimental design).

So, reproducibility simply means re-running someone else's code on the same data to obtain the same result. At first, this may seem a very simple task, but actually, it requires properly organising and managing all the analysis material. Without adequate programming and software development skills, it is very difficult to guarantee the reproducibility of the analysis results.

The aim of the present book is to describe programming good practices and introduce common tools used in software development to guarantee reproducibility of the analysis results. Inspired by Richard McElreath's talk, we want to make scientific research an open-source knowledge development.

1.1 Book Structure

The book is structured as follows.

- In Chapter 2 [work in progress], we introduce the Open Science Framework (OSF), a free, open-source web application that allows researchers to collaborate, document, archive, share, and register research projects, materials, and data.
- In Chapter 3, we describe recommended practices to organize all the materials and files of our projects and which are the advantages of creating a well structured, documented, and licensed repository.
- In Chapter 4, we discuss the main guidelines regarding organizing, documenting, and sharing data.
- In Chapter 5, we provide general good practices to create readable and maintainable code and we describe the functional style approach.
- In Chapter 6, we provide a basic tutorial about the use of the terminal.
- In Chapter **??**, we introduce version control, a powerful system for managing the development of our project. In particular, first, we provide a basic tutorial about the use of the terminal. Next, we introduce Git and GitHub for managing and tracking our projects during the development.
- In Chapter [TODO: add ref], we discuss how to manage the analysis workflow to enhance results reproducibility and code maintainability.

- In Chapter [TODO: add ref], we introduce the main tools to create dynamic documents that integrate narrative text and code describing the advantages.
- In Chapter [TODO: add ref], we discuss how to manage our project requirements and dependencies (software and package versions) to enhance results reproducibility.
- In Chapter [TODO: add ref], we introduce Docker and the container technology that allows us to create and share an isolated, controlled, standardized environment for our project.

1.2 Instructions

Let's discuss some useful tips about how to get the best out of this book.

1.2.1 Programming Language

This book provides useful recommendations and guidelines that can be applied independently of the specific programming language used. However, examples and specific applications are based on the R programming languages.

In particular, each chapter first provides general recommendations and guidelines that apply to most programming languages. Subsequently, we discuss specific tools and applications available in R.

In this way, readers working with programming languages other than R can still find valuable guidelines and information and can later apply the same workflow and ideas using dedicated tools specific to their preferred programming language.

1.2.2 Long Journey

To guarantee results replicability and project maintainability, we need to follow all the guidelines and apply all the tools covered in this book. However, if we are not already familiar with all these arguments, it could be incredibly overwhelming at first.

Do not try to apply all guidelines and tools all at once. Our recommendation is to build our reproducible workflow gradually, introducing new guidelines and new tools step by step at any new project. In this way, we have the time to learn and familiarize ourselves with a specific part of the workflow before introducing a new step.

The book is structured to facilitate this process, as each chapter is an independent step to build our reproducible workflow:

- Share our materials using online repositories services
- Learn how to structure and organize our materials in a repository
- Follow recommendations about data organization and data sharing
- Improve code readability and maintainability using a Functional Style
- Learn version control and collaboration using Git and Github
- Manage analysis workflow with dedicated tools
- Create dynamic documents
- Manage project requirements and dependencies using dedicated tools

- Create a container to guarantee reproducibility using Docker

Learning advanced tools such as Git, pipeline tools, and Docker still requires a lot of time and practice. They may even seem excessively complex at first. However, we should consider them as an investment. As soon as our analyses will become more complex than a few lines e of code, these tools will allow us to safely develop and manage our project.

1.2.3 Non-Programmer Friendly

Most of the arguments discussed in this book are the A-B-C of the daily workflow of many programmers. The problem is that most researchers lack any kind of formal training in programming and software development.

The aim of the book is exactly that: to introduce popular tools and common guidelines of software development into scientific research. We try to provide a very gentle introduction to many programming concepts and tools without assuming any previous knowledge. Note, however, that we assume the reader is already familiar with the R programming language for specific examples and applications.

2

The Open Science Framework

[Work in Progress]

3 Projects

In the previous chapter, we learned to share our materials through the Open Science Framework (OSF). However, if we simply collect all our files together without a clear structure and organisation, our repository will be messy and useless. In fact, it will be difficult for anyone to make sense of the different files and use them.

In this chapter, we describe recommended practices to organize all the materials and files into a structured project and which are the advantages of creating a well documented, and licensed repository.

3.1 Project Structure

To facilitate the reproducibility of our study results, it is important to organize our analysis into a project. A project is simply a directory where to collect all the analysis files and materials. So, instead of having all our files spread around the computer, it is a good practice to create a separate directory for each new analysis.

However, collecting all the files in the same directory without any order will only create a mess. We need to organize the files according to some logic, we need a structure for our project. A possible general project template is,

```
- my-project/
    |
    |-- data/
    |-- analysis/
    |-- code/
```

```
|-- outputs/
|-- documents/
|-- README
|-- LICENSE
```

Of course, this is just indicative, as project structures could vary according to the specific aims and needs. However, this can help us to start organizing our files (and also our ideas). A well structured and documented project will allow other collaborators to easily navigate around all the files and reproduce the analysis. Remember, our best collaborator is the future us!

3.1.1 Project Elements

Let's discuss the different directories and files of our project. Again, these are just general recommendations.

3.1.1.1 `data/`

A directory in which to store all the data used in the analysis. These files should be considered `read-only`. In the case of analyses that require some preprocessing of the data, it is important to include both the raw data and the actual data used in the analysis.

Moreover, it is a good practice to always add a file with useful information about the data and a description of the variables (see Chapter [TODO: add ref]). We do not want to share uninterpretable, useless files full of 0-1 values, right?

For example, in the `data/` directory we could have:

- `data_raw`: The initial raw data (before preprocessing) to allow anyone to reproduce the analysis from the beginning.
- `data`: The actual data used in the analysis (obtained after preprocessing).
- `data-README`: A file with information regarding the data and variables description.

In Chapter [TODO: add ref], we describe good practices in data organization and data sharing. In particular, we discuss possible data structures (i.e., wide format, long format, and relational structure), data documentation, and issues to take into account when sharing the data. [TODO: check coherence with actual chapter]

3.1.1.2 `analysis/` and `code/`

To allow analysis reproducibility, we need to write some code. In the beginning, we usually start to collect all the analysis steps into a single script. We start by importing the data and doing some data manipulation. Next, we move to descriptive analysis and finally to inferential analyses. While running the analysis, we are likely jumping back and forward in the code adding lines, changing parts, and fixing problems to make everything work. This interactive approach is absolutely normal in the first stages of a project, but it can easily introduce several errors.

As we may have experienced, very quickly this script becomes a long, disordered, incomprehensible collection of command lines. Unintentionally, we could overwrite objects values or, maybe, the actual code execution order is not respected. At this point, it may be not possible to reproduce the results and debugging would be slow and inefficient. We need a better approach to organising our code and automatizing code execution. Ready to become a true developer?

The idea is simple. Instead of having a unique, very long script with all the code required to run the analysis, we break down the code into small pieces. First, we define in a separate script our functions to execute each step of the analysis. Next, we use these functions in another script to run the analysis. For example, we could define in a separate script a function `data_wrangling()` with all the code required to prepare our data for the analysis. This function could be very long and complex, however, we simply need to call the function `data_wrangling()` in our analysis script to execute all the required steps.

This approach is named **Functional Style**: we break down large problems into smaller pieces and we define functions or combinations of functions to solve each piece. This approach is discussed in detail in Chapter [TODO: add ref]. To summarise, Functional Style has several advantages: it enhances code readability, avoids repetition of code chunks, and facilitates debugging.

In our project we can organize our scripts into two different directories:

- `analysis/`: Collecting the scripts needed to run all the steps of the analysis.
- `code/`: Collecting all the scripts in which we defined the functions used in the analysis.

This division allows us to keep everything organized and in order. We can easily move back and forward between scripts, defining new functions when required and using them in the analysis. Moreover, adequate documentation (both for the functions and for the analysis scripts) allows other collaborators to easily navigate the code and understand the purpose of each function.

In Chapter [TODO: add ref], we discuss in detail the functional style approach, considering general good practices to write tidy, documented, and efficient code. In Chapter [TODO: add ref], we describe possible methods to manage the analysis workflow.

3.1.1.3 `outputs/`

We can store all the analysis outputs in a separate directory. These outputs can be later used in all other documents of our project (e.g., scientific papers, reports, or presentations). Depending on the specific needs, we could organize outputs into different sub-directories according to the type of output (e.g., fitted models, figures, and tables) or, in case of multiple analyses, we could create different dedicated sub-directories.

Moreover, it may be useful to save intermediate steps of the analysis to avoid re-running very expensive computational processes (e.g., fitting Bayesian models). Therefore,

we could have a `cache/` sub-directory with all the intermediate results saved allowing us to save time.

However, we should ensure that all outputs can be obtained starting from scratch (i.e., deleting previous results as well as cached results). Ideally, other colleagues should be able to replicate all the results starting from an empty directory and re-running the whole analysis process on their computer.

In Chapter [TODO: add ref], we describe possible methods to manage the analysis workflow. In particular, we present the R package `trackdown` that enhances results reproducibility and introduces an automatic caching system for the analysis results.

3.1.1.4 `documents/`

A directory with all the documents and other materials relevant to the project. These may include, for example, the paper we are working on, some reports about the analysis to share with the colleagues, slides for a presentation, or other relevant materials used in the experiment.

To allow reproducibility, all documents that include analysis results should be dynamic documents. These are special documents that combine code and prose to obtain the rendered outputs. In this way, figures, tables, and values in the text are obtained directly from the analysis results avoiding possible copying and paste errors. Moreover, if the analysis is changed, the newly obtained results will be automatically updated in the documents as well when the output is rendered.

Note that it is preferable to keep the code used to run the analysis in a separate script other than the dynamic documents used to communicate the results. This issue is further discussed in Section 3.1.3.2.

In Chapter [TODO: add ref], we briefly introduce dynamic documents using Quarto. In particular, we discuss how to smoothly integrate dynamic documents in our project structure and workflow to enhance reproducibility.

3.1.1.5 `README`

All projects should have a `README` file with the general information about the project. This is the first file any colleague will look at and many online repositories automatically display it on the project homepage.

A `README` file should provide enough information so anyone can understand the project aims and project structure, navigate through the project files, and reproduce the analysis. Therefore, a `README` file could include:

- **Project Title and Authors:** The project title and list of main authors.
- **Project Description:** A brief description of the project aims.
- **Project Structure:** A description of the project structures and content. We may list the main files included in the project.

- **Getting Started:** Depending on the type of project, this section provides instructions on how to install the project locally and how to reproduce the analysis results. We need to specify both,
  - *Requirements*: The prerequisites required to install the project and reproduce the analysis. This may include software versions and other dependencies (see Chapter [TODO: add ref]).
  - *Installation/Run Analysis*: A step-by-step guide on how to install the project/reproduce the analysis results.
- **Contributing:** Indications on how other users can contribute to the project or open an issue. Contributions are what make the open-source community amazing.
- **License:** All projects should specify under which license they are released. This clarifies under which conditions other users can copy, share, and use our project. See Section 3.1.1.6 for more information about licenses.
- **Citation:** Instructions on how to cite the project. We could provide both, a plain text citation or a `.bib` format citation (see https://www.overleaf.com/learn/latex/Bibliography_management_with_biblatex#The_bibliography_file).
- **Acknowledgements:** Possible acknowledgements to recognize other contributions to the project.

`README` files are usually written in Markdown. Markdown is a lightweight markup language with a simple syntax for style formatting. Therefore, to edit a `README`, we do not need specific software but only a plain-text editor. Moreover, another advantage of Markdown files is that they can be easily rendered by a web browser and online repositories will automatically present the rendered output. For an introduction to the Markdown syntax, consider the *"Markdown Guide"* available at https://www.markdownguide.org/.

The information included in the `README` and its structure will vary according to the project's specific needs. Ideally, however, there should always be enough details to allow other researchers not familiar with the project to understand the materials and reproduce the results. For examples of `README` files, consider https://github.com/ClaudioZandonella/trackdown or https://github.com/ClaudioZandonella/Attachment.

3.1.1.6 `LICENSE`

Specifying a license is important to clarify under which conditions other colleagues can copy, share, and use our project. Without a license, our project is under exclusive copyright by default so other colleagues can not use it for their own needs although the project may be "publicly available" (for further notes see https://opensource.stackexchange.com/q/1720). Therefore, we should always add a license to our project.

Considering open science practices, our project should be available under an open license allowing others colleagues to copy, share, and use the data, with attribution and copyright as applicable. Specific conditions, however, may change according to the different licenses.

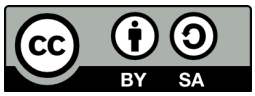
In the case of **software** or **code releasing** the most popular open-source license are:

- **MIT License:** A simple and permissive license that allows other users to copy, modify and distribute our code with conditions only requiring preservation of copyright and license notices. This could be done for private use and commercial use as well. Moreover, users can distribute their code under different terms and without source code.
- **Apache License 2.0:** Similar to the MIT license, this is a permissive license that allows other users to copy, modify and distribute our code (for private use and commercial use) with conditions requiring preservation of copyright and license notices. Differently from the MIT license, users are also required to state any change. As before, however, users can distribute their code under different terms and without source code.
- **GNU General Public License v3.0 (GPL):** This is a strong copyleft license that allows other users to copy, modify and distribute our code (for private use and commercial use) with conditions requiring preservation of copyright and license notices. In addition, however, users are also required to state any change and make available complete source code under the same license (GPL v3.0).

Note that these licenses follow a hierarchical order that affects the license of derivative products. Let's suppose we are working on a project based on someone else code distributed under the GPL v3.0 license. In this case, we are required again to make the code available under the GPL v3.0 license. Instead, if another project is based on someone else code distributed under the MIT license, we are only required to indicate that part of our project contains someone's MIT licensed code. We do not have to provide further information and we can decide whether or not to publish the code and under which license. See https://www.quora.com/What-are-the-key-differences-between-the-GNU-General-Public-license-and-the-MIT-License for further discussion.

In the case of **publishing materials** other than code (e.g., data, documents, images, or videos), we can choose one of the **Creative Commons License** (CC; see https://creativecommons.org/about/cclicenses/). These licenses allow authors to retain copyright over their published material specifying under which conditions other users can reuse the materials:s

- **Attribution (BY)**  : Credit must be given to the creator
- **Share Alike (SA)**  : Adaptations must be shared under the same terms
- **Non-Commercial (NC)**  : Only non-commercial uses of the work are permitted
- **No Derivatives (ND)**  : No derivatives or adaptations of the work are permitted

For example, if we want to publish materials allowing other users to reuse and adapt them (also for commercial use) but requiring them to give credit to the creator and keeping the same license, we can choose the **CC BY-SA** license 

.

This is not an exhaustive discussion about licenses. We should pay particular attention when choosing an appropriate license for a project if patents or privacy issues are involved. In Chapter [TODO: add ref], we discuss specific licenses related to sharing data/databases. Further information about licenses can be found at:

- Open Science Framework documentation: https://help.osf.io/hc/en-us/articles/360019739014-Licensing
- GitHub documentation: https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/licensing-a-repository
- GitHub license chooser: https://choosealicense.com/
- Creative Commons website: https://creativecommons.org/

Instructions-Box: Adding a License

To add a license to our project:

1. Copy the selected license template in a plain-text file (i.e., `.txt` or `.md`) named `LICENSE` at the root of our project. Many online repositories allow us to select from common license directly through their online interface. In Chapter **??**, we describe how to add a License on GitHub (see also https://docs.github.com/en/communities/setting-up-your-project-for-healthy-contributions/adding-a-license-to-a-repository).

2. Indicate the selected license also in the `README` specifying possible other information (e.g., different materials could be released under different conditions).

3. In the case of software, it is a good practice to attach a short notice at the beginning of each source file. For example:

```
# Copyright (C) <year>  <name of author>
# This file is part of <project> which is released under <license>.
# See file <filename> or go to <url> for full license details.
```

Now that we have added a license, other colleagues can use our project according to the specified conditions.

3.1.2 Naming Files and Directories

To create a well organized project that other colleagues can easily explore, it is also important to name directories and files appropriately. When naming a directory or a file, we should follow these general guidelines:

- **Use Meaningful Names.** Provide clear names that describe the content and aim of the file (or directory).

```
"untitled.R" # not meaningful name
```

```
"analysis-experiment-A.R"        # clear descriptive name
```

  Note that prefix numbers can be used if files (or directories) are required to appear in a specific order.

```
# ordered list of files
"01-data-cleaning.R"
"02-descriptive-analysis.R"
"03-statistical-models.R"
```

- **Prefer *lower-case* Names.** There is nothing wrong with capital letters. However, case sensitivity depends on the specific operative system. For example, in macOS and Linux systems, the files `my-file.txt` and `My-File.txt` can not coexist in the same directory. Therefore, it is recommended to always use lower-case names.
- **Specify Files Extension.** Always indicate the specific file extension.
- **Avoid Spaces.** In many programming languages, spaces are used to separate arguments or variable names. We should always use underscores (`"_"`) or dashes (`"-"`) instead of spaces.

```
"I like to/mess things up.txt" # Your machine is gonna hate you
```

```
"path-to/my-file.txt"
```

- **Avoid Special Characters.** Character encoding (i.e., how the characters are represented by the computer) can become a problematic issue when files are shared between different systems. We should always name files and directories using only basic Latin characters and avoiding any special character (accented characters or other symbols). This would save us from lots of troubles.

```
"brûlée-recipe.txt" # surely a good recipe for troubles
```

```
"brulee-reciepe.txt" # use only basic Latin characters
```

3.1.3 Project Advantages

Organizing all our files into a well structured and documented project will allow other colleagues to easily navigate around all the files and reproduce the analysis. Remember that this may be the future us when, after several months, reviewer #2 will require us to revise the analysis.

Structuring our analysis into a project, however, has also other general advantages. Let's discuss them.

3.1.3.1 Working Directory and File Paths

When writing code, there are two important concepts to always keep in mind:

- **Working Directory:** The location on our computer from where a process is executed. We can think of it as our current location when executing a task.
- **File Paths:** A character string that indicates the location of a given file on our computer. We can think of it as the instruction to reach a specific file.

When pointing to a file during our analysis (for example to load the data or to save the results), we need to provide a valid file path for the command to be executed correctly. Suppose we want to point to a data file (`my-data.csv`) that is on the Desktop in the project directory (`my-project/`).

```
Desktop/
 |
 |-  my-project/
 |     |
 |     |- data/
 |     |    |- my-data.csv
```

There are two different possibilities:

- **Absolute Paths:** Files location is specified relative to the computer root directory. Absolute paths work regardless of the current working directory specification. However, they depend on the computer's exact directories configuration. Thus, they do not work on someone else's computer. Considering our example, we would have

```
# Mac
"/Users/<username>/Desktop/my-project/data/my-data.csv"

# Linux
"/home/<username>/Desktop/my-project/data/my-data.csv"

# Windows
"c:\Users\<username>\Desktop\my-project\data\my-data.csv"
```

- **Relative Paths:** Files location is specified relative to the current working directory. They do not depend on the computer's whole directories configuration but only on the directories configuration relative to the current working directory. Therefore, if the working directory is set to the root of our project (`my-project/`), we would have

```
# Mac and Linux
"data/my-data.csv"

# Windows
"data\my-data.csv"
```

Absolute paths hinder reproducibility as they do not work on someone else's computer. We should always set the working directory at the root of our project and then use relative paths to point to any file in the project.

When opening a project directory, many programs automatically set the working directory at the root of the project but this is not guaranteed. Therefore, we should always ensure that our IDE (Integrated Development Environment; e.g. Rstudio, Visual Studio Code, PyCharm) is doing that automatically or we should do it manually.

Once the working directory is correctly set (automatically or manually), relative paths will work on any computer (up to the operative system; see *"Details-Box: Garden of Forking Paths"* below). This is one of the great advantages of using projects and relative paths: all the files are referred to relative to the project structure and independently of the specific computer directories configuration.

[TODO: check paths windows]

Details-Box: Garden of Forking Paths

The syntax to define file paths differs according to the operative system used. In particular, the main differences are between Unix systems (macOS and Linux) and Windows. Fortunately, main programming languages have ad hoc solutions

to allow the same file path to work on different operating systems (e.g. for R see Section [TODO: add ref]; for Python see https://medium.com/@ageitgey/python-3-quick-tip-the-easy-way-to-deal-with-file-paths-on-windows-mac-and-linux-11a072b58d5f). Therefore, by using adequate solutions, we do not have to bother about the operative system being used while coding.

Unix Systems

- The forward slash `"/"` is used to separate directories in the file paths.

```
"my-project/data/my-data.csv"
```

- The computer *root-directory* is indicated by starting the file path with a forward slash `"/"`.

```
# Mac
"/Users/<username>/Desktop/my-project/data/my-data.csv"

# Linux
"/home/<username>/Desktop/my-project/data/my-data.csv"
```

- The user *home-directory* (`""/Users/<username>/"` in MacOS and `""/home/<username>/"` in Linux) is indicated starting the file path with a tilde character `"~"`.

```
"~/Desktop/my-project/data/my-data.csv"
```

Windows Systems

- The backslash `"\"` is used to separate directories in the file paths.

```
"my-project/data/my-data.csv"
```

- The computer *root-directory* is indicated by starting the file path with `"C:\"`.

```
"C:\Users\<username>\Desktop\my-project/data/my-data.csv"
```

- Window does not define a user's *home-directory*. Therefore, the tilde character `"~"` actually points to the `Documents` directory.

Other Path Commands Two other common commands used in path definition are:

- `"./"` to indicate the current working directory.
- `"../"` to indicate the parent folder of the current working directory. Note that we can combine it multiple times to reach the desired file location (e.g., `"../../<path-to-file>"` to go back to two folder levels).

3.1.3.2 Centralize the Analysis

Another advantage of the proposed project structure is the idea to keep separating the actual analysis from the communication of the results. This is not an advantage of using a project per se, but it pertains to the way we structure the project. Let's clarify this point.

It often happens that in the first stages of a project, we do some preliminary analysis in a separate script. Usually, at these stages, the code is pretty raw and we go forward and backwards between code lines making changes to run the analysis and obtain some initial results. Next, probably we want to create an internal report for our research group to discuss the initial results. We create a new script, or we use some dynamic documents (e.g., Quarto or Rmarkdown). We copy and paste the code from the initial raw script trying to organize the analysis a little bit better, making some changes, and adding new parts. After discussing the results, we are going to write a scientific paper, a conclusive report, or a presentation to communicate the final results. Again, we would probably create a new script or use some dynamic documents. Again, we would copy and paste parts of the code while continuing to make changes and add new parts.

At the end of this process, our analysis would be spread between multiple files and everything would be very messy. We would have multiple versions of our analysis with some scripts and reports with outdated code or with slightly different analyses. In this situation, reproducing the analysis, making some adjustments, or even just reviewing the code, would be really difficult.

In the proposed approach, instead, we suggest keeping the actual analysis separate from the other parts of the project (e.g., communication of the results). As introduced in Section 3.1.1.2, a good practice is to follow a functional style approach (i.e., defining functions to execute the analysis steps) and to organize all the code required to run the analysis in a sequence of tidy and well-documented scripts. In this way, everything that is needed to obtain the analysis results is collected together and kept separate from the other parts of the project. With this approach we avoid having multiple copies or versions of the analysis and reproducing the analysis, reviewing the code, or making changes would be much easier. Subsequently, analysis results can be used in reports, papers, or presentations

to communicate our findings.

Of course, this is not an imperative rule. In the case of small projects, a simple report with the whole analysis included in it may be enough. However, in the case of bigger and more complex projects, the proposed approach allows to easily maintain the project and develop the code keeping control of the analysis and enhancing results reproducibility.

In Chapter [TODO: add ref] and Chapter [TODO: add ref], we describe how to adopt the functional style approach and how to manage the analysis workflow, respectively. In Chapter [TODO: add ref], we discuss how to smoothly integrate dynamic documents in our project structure and workflow to enhance reproducibility.

3.1.3.3 Ready to Share and Collaborate

Finally, one of the advantages of organizing our analysis into a well structured and documented project is that everything required for the analysis is contained in a single directory. We do not have to run around our computer trying to remember where some files were saved. All materials are in the same directory and we can easily share the whole directory with our colleagues or other users.

This aspect may seem trivial at the beginning. Overall we are just creating a directory, right?. Actually, this is the first step towards integrating into our workflow modern tools and solutions for collaborative software development, such as web services for hosting projects relying on version control systems. We are talking about Git and GitHub, two very powerful and useful tools that are becoming popular in scientific research as well.

In particular, Git is a software for tracking changes in any file of our project and for coordinating the collaboration during the development. Github integrates the Git workflow with online shared repositories adding several features and useful services (e.g., GitHub Pages and GitHub Actions). Combining online repositories (e.g., GitHub, GitLab, or other providers) with version control systems, such as Git, we can collaborate with other colleagues and share our project in a very efficient way. They may seem overwhelming at first, however, once we will get used to them, we will never stop using them.

In Chapter [TODO: add ref], we introduce the use of Git and GitHub to track changes and collaborate with others on the development of our project.

3.2 RStudio Projects

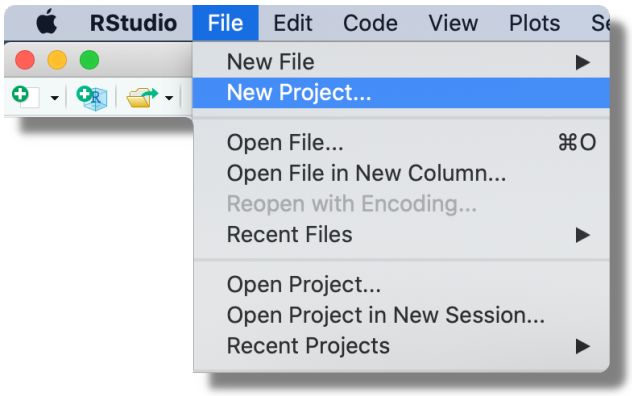
RStudio has built-in support for projects that allows us to create independent RStudio sessions with their own settings, environment, and history. Let's see how to create a project directly from RStudio and discuss some specific features.

3.2.1 Creating a New Project

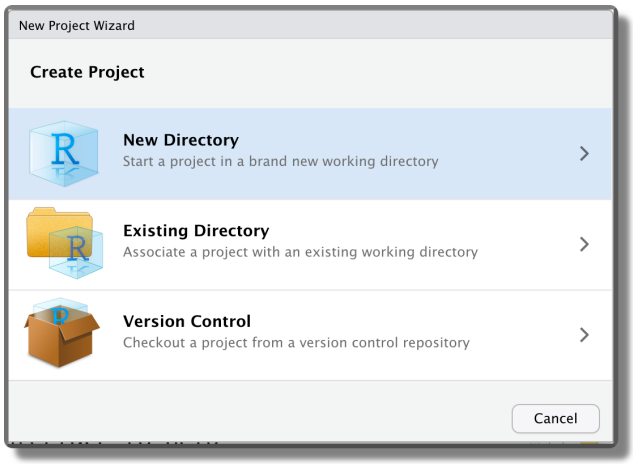
To create a new RStudio project:

1. From the top bar menu, click *"File -> New Project"*. Note that from this menu, we can also open already created projects or see a list of recent projects by clicking

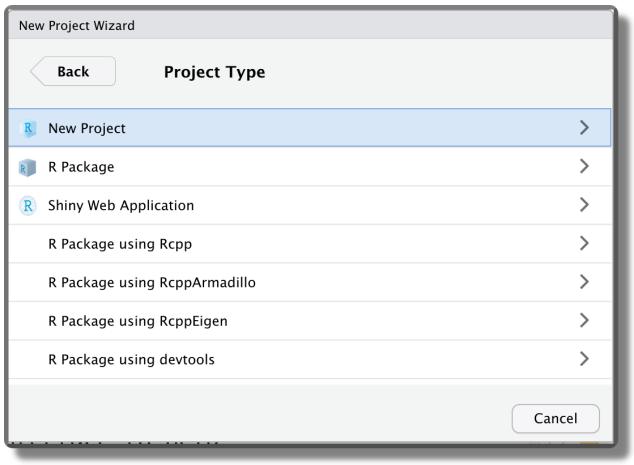
"Open Project..." or *"Recent Projects"*, respectively.



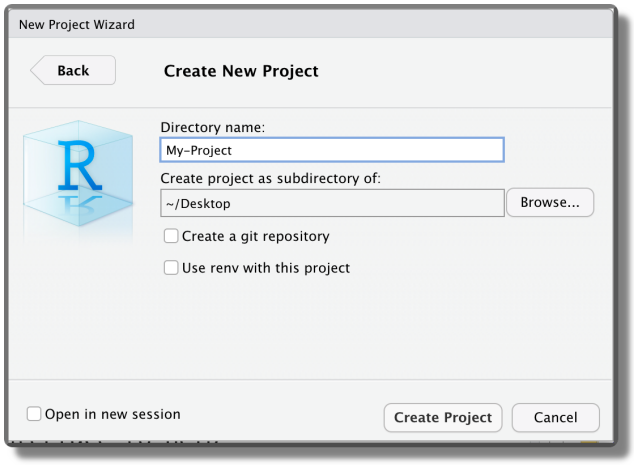
2. When creating a new project, we can decide between starting from a new directory (*"New Directory"*; the most common approach), associating a project with an already existing directory (*"Existing Directory"*), or associating a project to an online repository (*"Version Control"*; for more information see Chapter [TODO: add ref]).



3. Selecting *"New Directory"*, then we can specify the desired project template. Different templates are available also depending on the installed packages. The default option is *"New Project"*.

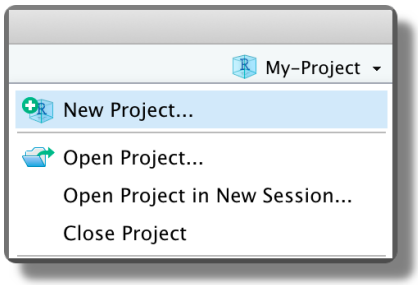


4. Finally, we can indicate the location where to create the project directory and specify the sdirectory name. This will be used also as the project name. Note that two more options are available *"Create a git repository"* and *"Use renv with this project"*. We discuss these options in Chapter [TODO: add ref]) and Chapter [TODO: add ref], respectively.



5. Selecting *"Crete Project"*, the project directory is created in the specified location and a new RStudio session is opened. Note that the Rstudio icon now displays the project name currently open 

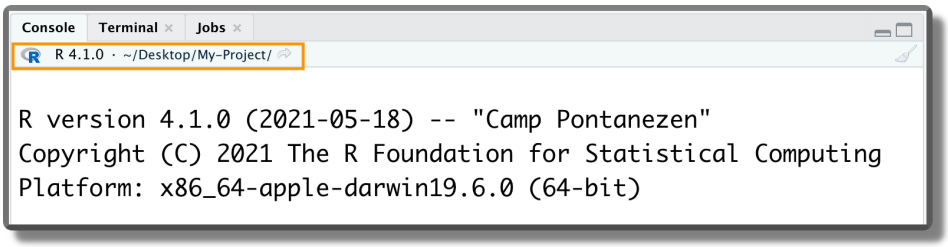
. The current project is also indicated in the top right corner. Click on it to see other project options and to close the project (*"Close Project"*; or from the top bar menu *"File -> Close Project"*).



3.2.2 Project Features

We now discuss the main features of RStudio Projects:

- **Working Directory and File Paths.** When opening a project, RStudio automatically sets the working directory at the project root. We can check the current working directory by looking at the top of the console panel or using the R command `getwd()`.



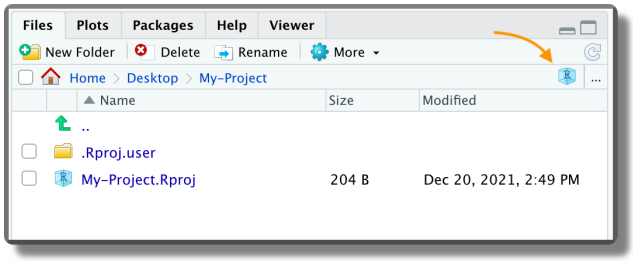
  As discussed in Section 3.1.3.1, this is a very useful feature because now we no longer have to bother about setting the working directory manually (we can finally forget about the `setwd()` command). Moreover, we can refer to any file using relative paths considering as reference the project root.

Details-Box: File Paths in R

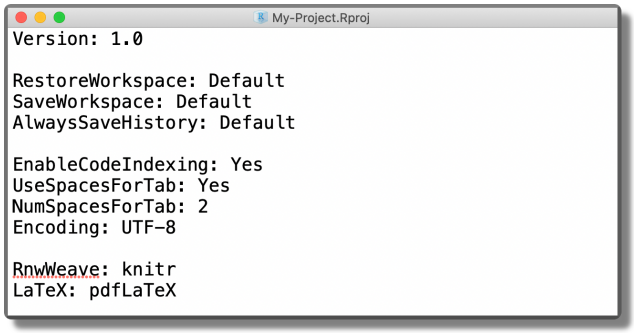
In R, we can specify file path using forward slash `"/"` or backslash `"\"` independently of the operative system we are currently using. However, the backslash has a special meaning in R as it is used as *escape character*. Thus, to specify a file path using backslash we have to double them (e.g., "my-project\\data\\my-data.csv"). All this leads to a simple solution:

> Always use forward slash `"/"` to specify file paths to avoid any troubles (e.g., "my-project/data/my-data.csv").

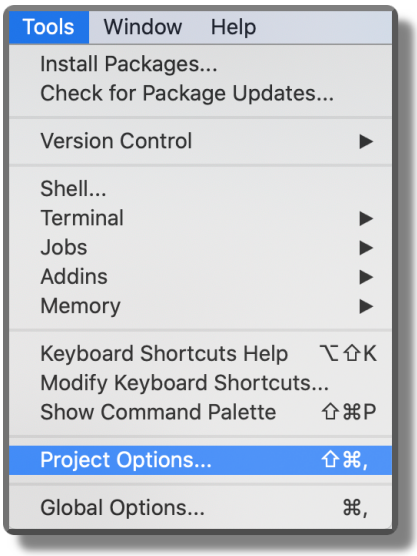
- **`<project-name>.Rproj` File.** The default Project template creates an empty directory with a single file named `<project-name>.Rproj` (plus some hidden files). Clicking on the `<project-name>.Rproj` file from the file manager (not from RStudio), we can open the selected project in a new RStudio session. Clicking on the `<project-name>.Rproj` file from the file panel in RStudio, instead, we can change the project settings (see the next point). Moreover, from the file panel in RStudio, if we click the Project icon on the top right corner (orange arrow in the image below), we are automatically redirected to the project root.



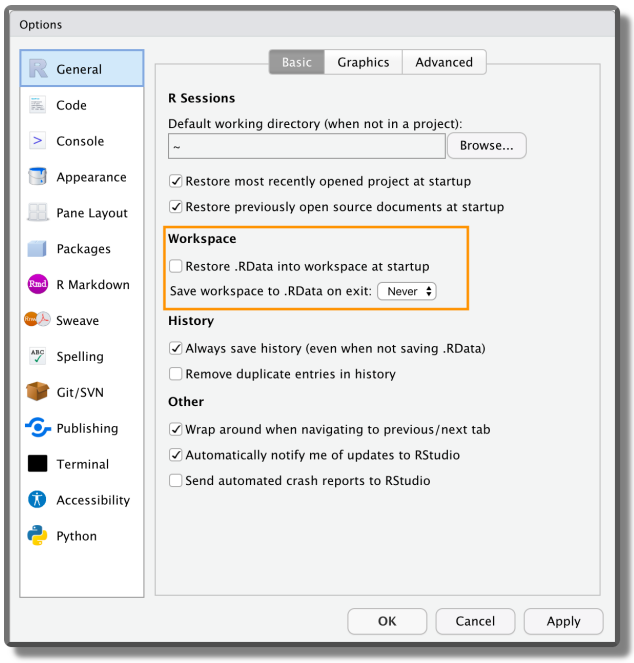
The `<project-name>.Rproj` file is simply a text file with the project settings. Using a text editor, we can see the actual content.



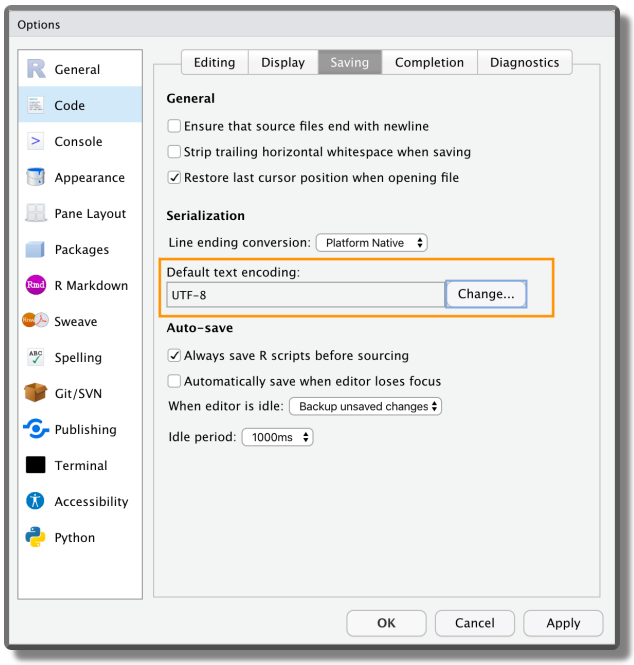
- **Project Settings.** We can specify specific settings for each project. From the top bar menu, select *"Tools -> Project Options"* and specify the required options according to our needs.



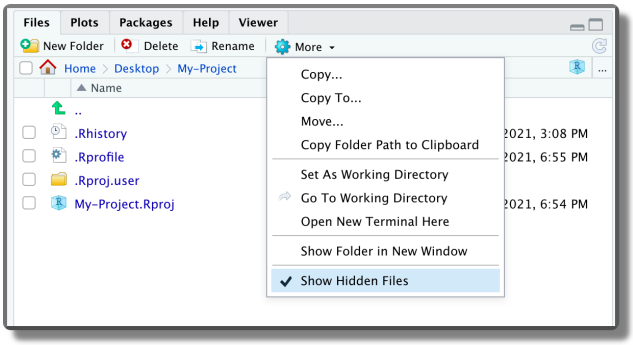
Some recommended settings should be applied as default to all projects. To do that, select from the top bar menu *"Tools -> Global Options"*. From the panel *"General"* in the *"Workspace"* section, ensure the box *"Restore..."* is **not** selected and the option *"Save..."* is set to **Never** (see figure below). This ensures that the workspace is not saved between sessions and every time we start from a new empty environment. The reason for doing this is that it forces us to write everything needed for our project in scripts and then we use scripts to create the required objects. It is a short-term pain for a long-term winning, as it enhances reproducibility and avoids bugs due to overwritten objects in the environment. Moreover, we can finally say goodbye to the horrible `rm(list=ls())` line of code found at the top of many scripts.



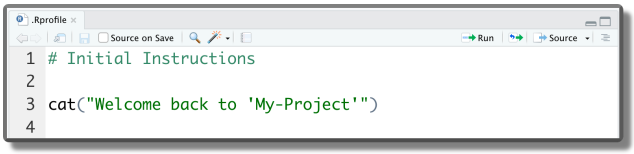
Another important setting is the encoding. From the panel *"Code"* ensure that the default text encoding is set to *"UTF-8"*. Encoding defines how the characters are represented by the computer. This could be problematic for different alphabets or special characters (e.g., accented characters). The UTF-8 encoding is becoming the modern standard as it covers all the possibilities.



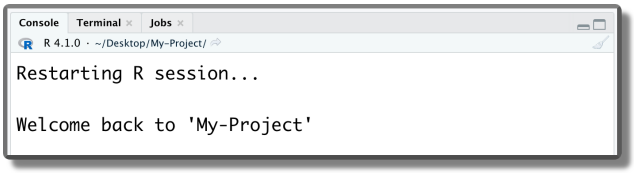
- **`.Rprofile` File.** This file is a special script that is automatically executed at the beginning of each session (or when the session is restarted). This file is not available by default but we can create it by naming an R script as `.Rprofile` (**without extension!**) [TODO: check in windows https://stackoverflow.com/questions/28664852/saving-a-file-as-rprofile-in-windows]. Note that names that begin with a dot `"."` are reserved for the system and are hidden from the file manager. To see the hidden files, select from the files panel in RStudio the option *"Show Hidden Files"*.



This file can be used to automate the execution of recurrent operations such as loading the required packages, setting R global options, or other package settings (e.g., ggplot themes). In the example below, we simply set a welcome message.



These commands are executed at the beginning of each session or when the session is restarted (`Ctrl` + `Shift` + `F10` on Windows and Linux; `Cmd/Ctrl` + `Shift` + `0` on macOS).



- **Like Multiple Office Desks.** Another advantage of RStudio projects is that we can quickly move from one project to another. When opening a former project, all panels, tabs, and scripts are restored in the same configuration as we left them. This allows us to go straight back into our workflow without wasting any time.

We can think of it as having a dedicated office desk for each of our projects. On the table, we can leave everything that is required to work on that project and we simply move between different office desks according to the current project we are working on.

To know more about RStudio projects, see https://r4ds.had.co.nz/workflow-projects.html.

3.2.3 Advanced features

We briefly point out some other advanced features of RStudio projects. These are covered in more detail in the following Chapters.

- **R Package Template.** As presented in Section 3.2.1, when creating a new project we can choose different templates. These templates automatically structure the project according to specific needs (e.g., creating a Shiny App or a Bookdown). In particular, one very interesting template is the *R Package Template*.

  The R Package Template is used to create... guess what? R packages. This template introduces some advanced features of R that become very handy when following a functional style approach. For example, we can manage our project package dependencies, easily load all our functions, document them, and create unit tests. We could go all the way and create a proper R package out of our project that other users can install. This requires some extra work and it may not be worth the effort, but of course, this will depend on the specific project aims.

  Anyway, the R package template is very useful when writing our functions. For this reason, we discuss further details about the R package template in Chapter [TODO: add ref] when discussing the functional style approach.

- **Git.** We can use version control systems such as Git to track changes on our RStudio projects. We can decide to create a git repository when creating our new project (see Section 3.2.1) or to associate the project to an existing repository. This is really a huge step forward in the quality of our workflow and it is absolutely worth the initial pain. All the required information to get familiar with Git and how to integrate the git workflow within RStudio projects is presented in Chapter [TODO: add ref].

- **`renv`.** To allow reproducibility of the result, everyone must use the same project dependencies. This includes not only the specific software and relative packages but also their specific version number. The R packages ecosystem changes quite rapidly, new packages are released every month and already available packages are updated from time to time. This means that in a year or two, our code may fail due to some changes in the underlying dependencies. To avoid these issues, it is important to ensure that the same package versions are always used.

We could list the required packages in a file and their versions manually or find a way to automate this process. As always, in R there is a package for almost everything and in this case, the answer is `renv`. `renv` allows us to manage all the R packages dependencies of our projects in a very smooth workflow. We can include `renv` in our project when creating a new project (see Section 3.2.1) or add it later. In Chapter [TODO: add ref], we introduce all the details about integrating the `renv` workflow in our projects.

 Documentation-Box

Bibtex

- Standard bibtex syntax for bibliography files
  https://www.overleaf.com/learn/latex/Bibliography_management_with_biblatex#The_bibliography_file

Markdown Syntax

- Markdown Guide
  https://www.markdownguide.org/

License

- Projects with no license
  https://opensource.stackexchange.com/q/1720
- Differences between GNU and MIT Licenses
  https://www.quora.com/What-are-the-key-differences-between-the-GNU-General-Public-license-and-the-MIT-License
- Creative Commons license
  https://creativecommons.org/about/cclicenses/
- Open Science Framework documentation
  https://help.osf.io/hc/en-us/articles/360019739014-Licensing
- GitHub documentation
  https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/licensing-a-repository
- GitHub license chooser
  https://choosealicense.com/
- Creative Commons website
  https://creativecommons.org/
- GitHub Setting a License guide
  https://docs.github.com/en/communities/setting-up-your-project-for-healthy-contributions/adding-a-license-to-a-repository

Paths

- Paths in Python
  https://medium.com/@ageitgey/python-3-quick-tip-the-easy-way-to-deal-with-file-paths-on-windows-mac-and-linux-11a072b58d5f

RStudio Projects

- RStudio projects general introduction
  https://r4ds.had.co.nz/workflow-projects.html

4 Data

In the previous chapters, we learned to share our materials through the Open Science Framework (OSF) and to organize all our files in a well structured and documented project.

In this chapter, we focus on data. In particular, in Section 4.1 we provide main guidelines regarding how to organize data inside our project and, in Section 4.2, we discuss main issues related to sharing data.

4.1 Organizing Data

Ideally, we want to store all the data relevant to our project in a dedicated directory (e.g., `data/`). These files should be considered `read-only` and, in the case of data that required some preprocessing, it is recommended to include also the original raw version of the data or provide a link to retrieve it.

However, storing the data and organising it efficiently is just part of the job. We also need to provide a detailed description with all the information to allow other colleagues (aka the future us) to understand what the data contain. We do not want to save and share useless files full of 0-1 values, right?

Let's discuss how to organize and document our data together with other useful recommendations.

4.1.1 Data Structure

There are many data formats depending on the specific data type (e.g., questionnaires, brain imaging, ERP study, conversations, etc.) and there are often no common standards.

Therefore, how to manage and organize data can vary according to the specific case and needs. However, as a general case, we can consider tabular data, that is, data organized in columns and rows.

Tabular data can be structured according to the *wide format* or the *long format*.

- **Wide Format.** Each variable value is recorded in a separate column.

Table 4.1: Wide Format

Name	Sex	Age	Pre	Post
Alice	F	24	...	...
Bob	M	21	...	...
Carl	M	23	...	...

- **Long Format.** Multiple variables can be recorded using a column to indicate the name of the measured variable and another column for the measured values.

Table 4.2: Long Format

Name	Sex	Age	Time	Value
Alice	F	24	Pre	...
Alice	F	24	Post	...
Bob	M	21	Pre	...
Bob	M	21	Post	...
Carl	M	23	Pre	...
Carl	M	23	Post	...

Depending on our specific needs, we may choose a wide or a long format to represent our data. Usually, the long format is the preferred data format, as it is required in most analyses. However, there is an important drawback when using the long format, that is, memory inefficiency.

In general, in a long format dataset, values are unique for each row only for a few columns, whereas for all the other columns the same values are (unnecessarily) repeated multiple times. The Long Format is an inefficient way to store our data.

In the case of small datasets, this does not make a big difference and we can safely continue to store the data using the long format as is the preferred data format in most analyses. When we need to deal with large datasets, however, memory may become a relevant issue. In these cases, we need to find a more efficient way to store our data. For example, we can use the relational model to organize our data.

- **Relational Model.** Instead of a single large table, the relational model organizes the data more efficiently by using multiple tables. Now, how can we use multiple

tables to organize our data more efficiently? To clarify this process, let's consider the data presented in the table above. We have information about participants (i.e., Name, Sex, and Age) and two measures (i.e., Pre and Post) on some outcomes of interest in our study. Using the long data format, participants' information is (unnecessarily) repeated multiple times. Following a relational model, we can create two separate tables, one for the participants' information and the other for the study measures. In this way, we do not repeat the same information multiple times optimizing the memory required to store our data.

Table 4.3: Subjects

ID	Name	Sex	Age
1	Alice	F	24
2	Bob	M	21
3	Carl	M	23

Table 4.4: Study

ID	Subject_ID	Time	Value
1	1	Pre	...
2	1	Post	...
3	2	Pre	...
4	2	Post	...
5	3	Pre	...
6	3	Post	...

But, how should we read these tables? Note that each table has a column `ID` with the unique identifier for each row. These IDs are used to define the relational structure between different tables. For example, in our case, the table Study has a column `Subject_ID` that indicates the subject ID for that specific row. Matching the `Subject_ID` value in the Study table with the `ID` column in the Subjects table, we can retrieve all the participants' information from a specific row.

This matching operation is usually defined as a **JOIN** operation and allows us to reconstruct our data in the desired format starting from separate tables.

If you have used SQL before, you are already familiar with relational databases and join operations. At first, these concepts may seem very complicated. In reality, as soon as we familiarize ourselves with all the technical terms, everything becomes very easy and intuitive. So do not be scared by buzz words such as *"primary-key"*, *"foreign-key"*, *"left-join"*, *"inner-join"*, or other jargon. You will quickly master all of them.

Most programming languages provide dedicated libraries and tutorials to execute all

these operations. For example, if working with R consider https://r4ds.had.co.nz/relational-data.html.

4.1.2 Documentation

A dataset without documentation is like a sky without stars... we can not see anything. We need to describe our data by providing details not only regarding all the variables but also about the data collection process or other information that could be relevant to properly interpret the data.

To document the data, we do not need anything fancy, a simple Markdown file is more than enough. An advantage of using Markdown files is that most online repositories automatically render Markdown files when navigating from the browser.

So, considering the data in the previous example, we could create the following `data-README.md`.

```
#-----     data-README.md    ----#

# Data README

## General Info

Details about the study/project, authors, License, or other relevant
information.

Description of the data collection process or links to the paper/external
documentation for further details.

## Details

The dataset `my-study.csv` is formed by n rows and k columns:

- `Name`. Character variable indicating the subject name
- `Sex`. Factor variable indicating the subject gender (levels are `"F"`
        for females and `"M"` for males)
- `Age`. Numeric variable indicating subject age (in years)
- `Time`. Factor variable indicating measure time (levels are `"Pre"` and
         `"Post"`)
- `Value`. Numeric variable indicating the outcome measure
- ...
```

There are no strict rules about what we should include or not in our data README file. We are free to structure it according to our needs. As a generic recommendation, however, we should specify data general information about:

- **Study/Project.** Reference to the specific study or project.

- **Authors.** List of the authors with contact details of the corresponding author or the maintainer of the project.
- **License.** Specify under which license the data are released (see Section 4.2.3).
- **Data Collection Process.** Description of relevant aspects of the data collection or external links for further details.

Ideally, we should provide all the required details to allow other colleagues to understand the context in which data were collected and to evaluate their relevance for possible other uses (e.g., meta-analyses). We can also provide links to raw data if these are available elsewhere.

Considering the information about the dataset, we should provide the dimension of the dataset and a list with the details for each variable. In particular, we should indicate:

- **Variable Name.** The name of the variable as it is coded.
- **Variable Type.** The specific variable type (i.e., string, integer, numeric, logic, other)
- **Variable Meaning.** The description of what the values indicate.
- **Variable Values.** Information about the variable values. This could include a list of all the labels and their meaning (or the number of levels) for categorical variables; the range of allowed values for numeric variables; the unit of measurement or a description of how the variable was computed if relevant.
- **Missing Values.** Specify how missing values are coded or any other value used to indicate special conditions.

4.1.3 Data Good Practices

Finally, let's consider two other general aspects regarding data that are important to take into account:

- **File Format.** There are many possible file formats. However, an important distinction concerns proprietary format and open format (definitions adapted from the Open Data Handbook, see https://opendatahandbook.org).

  Proprietary Format. Those formats owned and controlled by a specific company. These data formats require specific dedicated software to be read reliably and usually, we need to pay to use them. Note that the description of these formats may be confidential or unpublished, and can be changed by the company at any time. Common examples are XLS and XLSX formats used by Microsoft Excel.

  Open Format. All those file formats that can be reliably used by at least one free/open-source software tool. A file in an open format guarantees that there are no restrictions or monetary fee to correctly read and use it. Common examples are the `RDA` and `RDS` formats used internally by R.

  We should avoid proprietary formats and always use open formats, instead. In particular, we should not use open formats related to specific software (even if is

open-source), but prefer general text formats (e.g., CSV) that can be used easily by many tools.

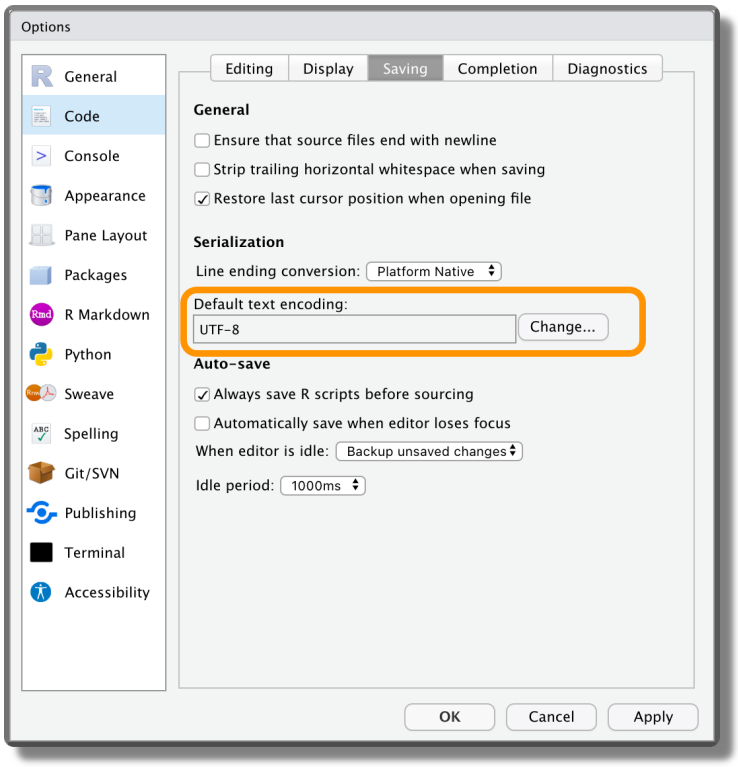
- **Machine Readable.** Data should be stored in a format that allows other researchers to easily use them. People saving tabular data in a PDF file would deserve to be punished in hell. Do they expect us to copy and paste all values by hand?

- **Character Encoding.** All characters (i.e., letters, numbers and symbols) are stored in our machine as sequence bits (i.e., the fundamental unit of information that can assume binary values 0 or 1). Encoding refers to the specific process of converting different characters into their numerical representation. While decoding is the reverse operation.

  There are different standards for different alphabets, so if we open a text file using the wrong decoding, characters can be confused with each other. Many issues typically arise with accented letters.

  UTF-8 is becoming the standard method for encoding files, as it allows to encode efficiently all characters from any alphabet. We should always save our data and any other file using the UTF-8 encoding. If for some reason we need to use a different encoding, remember to explicitly state the used encoding. This will save our colleagues from lots of trouble.

  From RStudio, we can set UTF-8 as default text encoding from *"Tools -> Global Options -> Code -> Saving"*.



- **The Rawer the Better.** We should share data in a format as raw as possible allowing other colleagues to reproduce the analysis from the very beginning.

4.2 Sharing Data

The most important part of our studies is not our conclusions but our data. Of course, this is a very provocative claim, but there is some truth in it. Making data available allows other researchers to build upon previous work leading to the efficient incremental development of the scientific field. For these reasons, we should always share our data.

Imagine we came up with a new idea. If we have access to many datasets relevant to the topic, we could immediately test our hypotheses. This would allow us to refine our hypotheses and properly design a new experiment to clarify possible issues. Wouldn't this be an ideal process?

Of course, one of the main resistance to sharing data is that data collection is extremely costly in terms of time, effort and money. Therefore, we may feel reluctant to share something that cost us so much and from which others can take all the merits if they come up with a new finding. Unfortunately, in today's hyper-competitive *"publish or perish"* world, the spotlight only shines on who comes with the new ideas neglecting the fundamental contribution of those who collected the data that made all this possible. Hopefully, in the future, the merit of both will be recognized.

Regarding this topic, there is a famous historical anecdote. Almost anyone knows or at least has heard Kepler's name. Johannes Kepler (1571 – 1630) was a German astronomer and mathematician who discovered that the orbits of planets around the Sun are elliptical and not circular as previously thought. This was an exceptional claim that earned him eternal fame with Kepler's laws of planetary motion (https://en.wikipedia.org/wiki/Kepler%27s_laws_of_planetary_motion). However, fewer people know the name of Brahe. Tycho Brahe (1546 - 1601) was a Danish astronomer, known for his accurate and comprehensive astronomical observations. Kepler became Brahe's assistant and thanks to Brahe's observational data was able to develop his theory about planetary motion. Of course, any researcher would like to become the next Kepler with an eternal law in our name, but who knows, we might as well as be remembered for our data. Do not waste the opportunity of being the next Brahe!

So, after this historical excursus, let's go back to the present and discuss relevant aspects of sharing data.

4.2.1 When, Where, Who?

Let's consider some common questions about sharing data:

- **When to Share?.** From a user point of view, probably the best timing is when the reference article goes to press. In this way, we try to get the best out of our valuable data but we also make them available so other researchers can build upon our work.

  We can also share data and materials in a private mode, granting access to selected people (e.g., reviewers) before the publication, and unlocking the public access in a second moment.

  However, some restrictions can arise from the legal clauses of the research grants or other legal aspects. If we are not allowed to share the data, we should at least justify the reasons (e.g., privacy issues, national security secrets).

- **Where to Share?.** We need to store materials in an online repository and provide the link to the repository and related research papers that are based on these materials.

  Ideally, we will collect everything in a single place. However, we could also store them in multiple places, to get the advantage of services with appropriate features for specialized types of data or materials. In this case, we should choose a central hub and provide links to other repositories. Of course, this increases the effort required to organize and maintain the project.

  There are many services that allow us to share our material and provide many useful features. For example:

  –  (https://osf.io)

  -  (https://github.com)
  -  (https://gitlab.com)
  -  (https://dataverse.org)
  -  (https://nyu.databrary.org)
  -  (https://www.icpsr.umich.edu)

  Moreover, there could be dedicated services for specific scientific fields or data types.

- **With whom to share?** Data need not be necessarily made available to everybody to meet open-science standards.

  Many online services allow us to select with whom to share our data. For example, Databrary allows different Release Levels for sharing data: Private (data are available only to the research team); authorized users (data are available to authorized researchers and their affiliates); public (data are available openly to anyone).

4.2.2 Legal Aspects

Specific legal restrictions related to data sharing change from state to state. Therefore, here we only discuss the general recommendation and we should always ensure to abide by our local legal restrictions.

- **Informed Consent and Permission to Share.** Participants should be informed about the study (e.g. purposes, approval by an ethics committee), what the risks are, and their rights (e.g. to leave the study). We need both, participants' consent to participate in the research study and participants' permission to share their research data.

  Note that these are two different things and the best practice for permission to share data is to seek it after the completion of research activities. This ensures that participants are fully aware of the study's procedures and what they are being asked to share.

- **Data Subject to Privacy Rules.** When sharing data we need to pay special attention to:

  - **Personal data** is information making a person identifiable.

  - **Sensitive data** are personal data revealing racial or ethnic origin, religious and philosophical beliefs, political opinions, labour union membership, health, sex life and sexual orientation, and genetic and biometric data.
  - **Legal data** are another type of personal data on which put attention
  - **Other data** such as private conversations, geolocalization, etc., also require special attention.

- **Data Anonymization.** Pseudonymisation or removing any information that could be used to identify participants is necessary before sharing the data.

- **Geographical Restrictions.** [TODO: check] The GDPR requires that all data collected on citizens must be either stored in the EU, so it is subject to European privacy laws, or within a jurisdiction that has similar levels of protection.

4.2.3 License

As discussed in Chapter 3.1.1.6, specifying a license is important to clarify under which conditions other colleagues can copy, share, and use our project. Without a license, our project is under exclusive copyright by default. Therefore, we always need to add a license to our project.

The same applies to data. In addition to the *Creative Commons Licenses* presented in Chapter 3.1.1.6, there are also other licenses specific for data/databases published by the **Open Data Commons**, part of the Open Knowledge Foundation.

A database right is a sui generis property right, that exists to recognise the investment that is made in compiling a database, even when this does not involve the "creative" aspect that is reflected by copyright (from Wikipedia; https://en.wikipedia.org/wiki/Database_right).

The main Open Data Commons Licenses are

- **ODC-BY.** Open Data Commons Attribution License requiring Attribution.
- **ODbL.** Open Data Commons Open Database License requiring Attribution and Share-alike.
- **PDDL.** Open Data Commons Public Domain Dedication and License dedicated to the Public Domain (all rights waived).
- **DbCL.** Database Content License requiring Attribution/Share-alike.

Note that Open Data Commons licenses distinguish between *"database"* and *"content"*. Why distinguish? Image a database of images...When licensing data, you need to know if the content of the database is homogeneous in terms of the license, or not, and to license accordingly.

[TODO: ?? expand]

For all the details about licenses dedicated to open data and how to apply them, see https://opendefinition.org/licenses/.

4.2.4 Metadata

Finally, to really be open, data must be findable as well. All our effort will be wasted if no one can find our data. However, just sharing the links of an online repository could not be enough. We need to make our data findable by research engines for datasets such as Google Dataset Search (https://datasetsearch.research.google.com).

To do that, our data needs to be formatted according to a machine-readable format. Moreover, we need to provide the required metadata. Metadata is information about the data going with main data tables. In particular, we have:

- **Data Dictionary:** a document detailing the information provided by data (e.g. the type of data, the author);
- **Codebook:** if necessary, a document describing decoding rules for encoded data (e.g. 0=female / 1=male, Likert descriptors).

These documents are similar to what we described in Section 4.1.2. However, before we created human-readable reports, now we need to structure these files in a machine-readable format.

Two data formats that are machine-readable (but also fairly human-readable) are the *eXtensible Markup Language* (XML) and the *JavaScript Object Notation* (JSON) formats.

These topics are beyond the aim of the present chapter. However, the *"Getting Started Creating Data Dictionaries: How to Create a Shareable Data Set"* tutorial by Buchanan et al. (2021) provides a detailed guide to creating dictionaries and codebooks for sharing machine-readable data.

In particular, they cover the use of:

- **JSON-Linked Data** (JSON-LD), a format designed specifically for metadata.
- **Schema.org** (https://schema.org/), a collaborative, community activity with a mission to create, maintain, and promote schemas for structured data on the Internet.

The combination of JSON-LD and Schema.org allows the creation of machine-readable data optimized for search engines.

Documentation-Box

Relational Data

- Relational Data in R
  https://r4ds.had.co.nz/relational-data.htm

Open Data

- Open Data Handbook
  https://opendatahandbook.org/
- Google Dataset Search
  https://datasetsearch.research.google.com/

Repository Platforms

- The Open Science Framework (OSF)
  https://osf.io
- GitHub
  https://github.com
- GitLab
  https://gitlab.com
- The Dataverse Project
  https://dataverse.org
- Databrary
  https://nyu.databrary.org
- Inter-university Consortium for Political and Social Research (ICPSR)
  https://www.icpsr.umich.edu

License Open Data

- Database Rights
  https://en.wikipedia.org/wiki/Database_right
- Conformant License
  https://opendefinition.org/licenses/

Metadata

- *"Getting Started Creating Data Dictionaries: How to Create a Shareable Data Set"* Buchanan et al. (2021)
  https://doi.org/10.1177/2515245920928007
- Schema.org
  https://schema.org/

5 Coding

In the previous chapters, we learned to share our materials using the Open Science Framework (OSF) and to organize all our files in a well structured and documented repository. Moreover we learned recommended practices to organize and share our data.

However, if we want our analysis to be reproducible, we need to write code, actually, we need to write good code. In this chapter, we provide some main guidelines about coding. In particular, in Section 5.1, we describe general coding good practices and introduce the functional style approach. Note that, although examples are in R, these general recommendations are valid for all programming languages. In Section 5.2, we further discuss specific elements related to R and we introduce more advanced R packages that can be useful when developing code in R.

5.1 Coding Style

Often, researchers' first experience with programming occurs during some statistical courses where they are used as a tool to run statistical analyses. In these scenarios, all our attention is usually directed to the statistical burden and limited details are provided about coding per sé. Therefore, we rarely receive any training about programming or coding good practices and we end up learning and writing code in a quite anarchic way.

Usually, the most common approach is to create a very long single script where, by trials-and-errors and copy-and-paste, we collect all our lines of code in a chaotic way hoping to obtain some reasonable result. This is normal during the first stages of an analysis, when we are just exploring our data, trying different statistical approaches, and coming up with new ideas. As the analyses get more complex, however, we will easily

lose control of what we are doing introducing several errors. At this point, reviewing and debugging the code will be much more difficult. Moreover, it would be really hard, if not impossible, to replicate the results. We need to follow a more structured approach to avoid these issues.

In Chapter [TODO: add ref], we discuss how to organize the scripts and manage the analysis workflow to enhance results reproducibility and code maintainability. In this Chapter, instead, we focus on how to write good code.

But, what does it mean to write *"good code"*? We can think of at least three important characteristics that define a good code:

1. **It Works.** Of course, this is a quite obvious prerequisite, no one wants a code that does not run.
2. **Readable.** We want to write code that can be easily read and understood by other colleagues. Remember that that colleague will likely be the future us.
3. **Easy to Maintain.** We want to organize the code so that we can easily fix bugs and introduce changes when developing our project.

In Section 5.1.1, we describe the general good practices to write readable code. In Section 5.1.2, we introduce the functional style approach to allow us to develop and maintain the code required for the analysis more efficiently. Finally, in Section 5.1.3, we briefly discuss some more advanced topics that are important in programming.

5.1.1 General Good Practices

> *"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."* – Martin Fowler, "Refactoring: Improving the Design of Existing Code"

> *"Good coding style is like using correct punctuation. You can manage without it, but it sure makes things easier to read."* – Hadley Wickham

If you do not agree with the above quotations, try to read the two following chunks of code.

```
y<-c(1,0,1,1,0,1,1);p<-sum(y)/length(y);if(p>=.6){"Passed"}else{"Failed"}
## [1] "Passed"
```

```
# Load subj test answers
exam_answers <- c(1,0,1,1,0,1,1) # 0 = wrong; 1 = correct

# Get exam score as proportion of correct answers
exam_score <- sum(exam_answers) / length(exam_answers)
```

```r
# Set exam pass threshold [0,1]
threshold <- .6

# Check if subj passed the exam
if (exam_score >= threshold) {
  "Passed"
} else {
  "Failed"
}
## [1] "Passed"
```

Which one have you found easier to read and understand? Unless you are a *Terminator* sent from the future to assassinate John Connor, we are sure you had barely a clue of what was going on in the first chuck. On the contrary, you could easily read and understand the second chunk, as if you were reading a plain English text. This is a simple example showing how machines do not need pretty well-formatted and documented code, but the programmers do.

Some programming languages have specific syntax rules that we need to strictly abide by to avoid errors (e.g., the indentation in Python is required to mark code blocks). Other programming languages are more flexible and do not follow strict rules (e.g., the indentation in R is for readability only). However, there are some general good practices common to all programming languages that facilitate code readability and understanding. In the next sections, we discuss some of the main guidelines.

5.1.1.1 Variable Names

> *"There are only two hard things in Computer Science: cache invalidation and naming things."* – Phil Karlton

Choosing appropriate object and variable names is important to facilitate code readability. Names should be:

- **Auto-descriptive.** The content of an object or of a variable should be obvious from their own names. Use meaningful names that clearly describe the object or variable content avoiding generic names. For example,

```r
y # generic name without useful information
```

```r
exam_answers # clear descriptive name
```

- **Right Length.** Object and variable names should be neither too long nor too short. Use the minimum number of words required to describe the object avoiding incomprehensible abbreviations. Usually 2 or 3 words are enough. For example,

```
average_outcome_score_of_the_control_gorup # too long
avg_scr_ctr # difficult to guess the abbreviation meaning
```

```
avg_score_control # clear descriptive name
```

Moreover, we should not be scared of using longer names if these are required to properly describe an object or a variable. Most IDEs (i.e., integrated development environments such as RStudio or Visual Studio Code) have auto-complete features to help us easily type longer names. At the same time, this is not a good excuse to create 12-word-long names. Remember, names should not be longer than what is strictly necessary.

Objects' and variables' names can not include spaces. Therefore, to combine multiple words, we need to adopt one of the following naming styles:

- **camelCase -** The first letter of the object or variable name is lowercase and the beginning of each new word is indicated by capitalizing its first letter. For example,

```
myObjectName       # camelCase
```

- **PascalCase -** Similar to camelCase, but also the first letter of the object or variable name is capitalized. For example,

```
MyObjectName       # PascalCase
```

- **snake_case -** All words are lower-case and they are combined using the character `"_"`. For example,

```
my_object_name     # snake_case
```

- **snake.case (deprecated) -** An alternative to the traditional snake_case, is to use the character `"."` instead of `"_"`. However, this is deprecated as in many programming languages the character `"."` is a reserved special character (see Section [TODO: add ref]). For example,

```
my.object.name     # snake.case
```

Usually, every programming language has its specific preferred style, but it does not really matter which style we choose. The important thing is to choose one style and stick with it naming consistently all objects and variables.

Finally, we should avoid any name that could lead to possible errors or conflicts. For example, we should

- Avoid special characters (accented characters or other symbols), use only basic Latin characters.
- Avoid naming objects or variables using common function names (e.g., `data`, `list`, or `sum`).
- Avoid using similar names for multiple objects that could easily be confused (e.g., `lm_fit`, `lm_Fit`).

Details-Box: Temporary Variables

Some extra tips concern the names of temporary variables that are commonly used in `for` loops or other data manipulation processes.

- **Position Index `i`.** To refer to some position indexes, the variable `i` is commonly used. Alternatively, we could use other custom names to indicate explicitly the specific selection operation. For example, we could use `row_i` or `col_i` to indicate that we refer to row index or column index respectively.

```
# Cities names
cities <- c("Amsterdam", "Berlin", "Cardiff", "Dublin")

# Loop by position index
for (i in seq_len(length(cities))) {
  cat(cities[i], " ")
}
## Amsterdam  Berlin  Cardiff  Dublin
```

- **Element `k`.** To refer to an element inside an object we can use the variable `k`. Alternatively, a common approach is to use the plural and the singular names to refer to the collection of all elements and the single element respectively (or an indicative name that represents a part of a unit, for example, `slice` for `pizza`).

```
# Loop by element
for (city in cities) {
  cat(city, " ")
}
## Amsterdam  Berlin  Cardiff  Dublin
```

These are not mandatory rules but just general recommendations. Using consistently distinct variables to refer to position indexes or single elements of a collec-

tion (e.g., `i` and `k`, respectively) facilitates the review of the code and allows us to identify possible errors more easily.

5.1.1.2 Spacing and Indentation

Again, some programming languages have specific syntax rules about spacing and indentation (e.g., Python), whereas other programming languages are more flexible (e.g., R). However, it is always recommended to use appropriate and consistent spacing and indentation to facilitate readability. As general guidelines:

- **Line Length.** Limit the maximum length of each line of code to 80 characters. Most IDEs display a vertical line in the editor to indicate the recommended margin.
- **Spacing.** Use spaces around operators and after commas separating function arguments. Spaces are free, so we should always use them to enhance readability.

```r
x<-sum(c(1:10,99),rnorm(5,mean=3,1))

if(test>=5&test<=10)print("...")
```

```r
x <- sum(c(1:10, 99), rnorm(n = 5, mean = 3, sd = 1))

if (test >= 5 & test <= 10) print("...")
```

- **Alignment.** Break long function or object definitions into multiple lines and align arguments to facilitate readability.

```r
my_very_long_list<-list(first_argument="something-very-long",
second_argument=c("many","objects"),third_argument=c(1,2,3,4,5))
```

```r
my_very_long_list <- list(
  first_argument = "something-very-long",
  second_argument = c("many", "objects"),
  third_argument = c(1, 2, 3, 4, 5)
  )
```

- **Indentation.** Always indent code blocks to facilitate understanding of the code structure. This is particularly important for nested conditional code blocks and loops. However, too complex nested code hinders readability. In this case, we may prefer to rewrite our code reducing the nesting structure and improving readability.

```
for (...) {   # Outer loop
...
for (...) {   # Inner loop
...
if (...) {    # Conditional
...
}}}
```

```
for (...) {   # Outer loop
  ...

  for (...) {   # Inner loop
    ...

    if (...) {    # Conditional

      ...
    }
  }
}
```

To indent the code, we can use spaces or Tabs. For a nice debate about this choice see https://thenewstack.io/spaces-vs-tabs-a-20-year-debate-and-now-this-what-the-hell-is-wrong-with-go (do not forget to watch the linked video as well). However, if we mix together Tabs and spaces this will lead to errors in programming languages that require precise indentation. This issue is very difficult to debug as Tabs and spaces look invisible. To avoid this problem, most editors allow the user to automatically substitute Tabs with a fixed number of spaces.

5.1.1.3 Comments

Comments are ignored by the program, but they are extremely valuable for colleagues reading our code. Thus, we should always include appropriate comments in our code. The future us will be very grateful.

Comments are used to provide useful information about the code in plain language. For example, we could describe the aim and the logic behind the next block of code, explain the reasons for specific choices, clarify the meaning of some particular uncommon code syntax and functions used, or provide links to external documentation.

Note that comments should not simply replicate the code in plain language, but they should rather explain the meaning of the code by providing additional information.

```
# Set x to 10
x <- 10
```

```
# Define maximum answer number
x <- 10
```

Remember, good comments should explain the why and not the what. If we can not understand what the code is doing by simply reading it, we should probably consider re-writing it.

Finally, comments can also be used to divide and organize the code scripts into sections. We further discuss how to organize scripts used to run the analysis in Chapter [TODO: add ref].

5.1.1.4 Other Tips

Here we list other general recommendations to facilitate code readability and maintainability.

- **Use Named Arguments.** Most programming languages allow defining function parameters according to their specific order in the function call or by specifying the parameter name. When possible, we should always define parameter values by specifying the parameter names. In this way, we enhance readability and limit the possibility of making errors.

```
x <- seq(0, 10, 2)
```

```
x <- seq(from = 0, to = 10, by = 2)
```

- **Avoid Deep Nesting.** Complex nested code structures hinder readability and are more complex to follow. In these cases, we may prefer to rewrite our code reducing the nesting structure to improve readability and maintainability.

```
check_value <- function(x){

  if (x > 0) {
    if (x > 100) {
      return("x is a positive large value")
    } else {
      return("x is a positive value")
    }
  } else {
    if (x < - 100) {
      return("x is a negative small value")
```

```
    } else {
      return("x is a negative value")
    }
  }

}
```

```
check_value <- function(x){

  if(x < - 100) return("x is a negative small value")
  if(x < 0) return("x is a negative value")
  if(x < 100) return("x is a positive value")

  return("x is a positive large value")
}
```

- **KISS (Keep It Simple Stupid).** As we get more proficient with a programming language, we usually start to use more advanced functions and rely on (not so obvious) language-specific behaviours or other special tricks. This allows us to write a few lines of compact code instead of lengthy chunks of code, but, as a result, readability is severely compromised. There is a trade-off between readability and code length.

  We should always aim to write elegant and readable code. This is different from trying to write code as short as possible. This is not a competition where we need to show off our coding skills. If we are not required to deal with specific constraints (e.g., time or memory efficiency), it is better to write a few more lines of simple code rather than squeezing everything into a single obscure line of code.

  In particular, we should not rely on weird language-specific behaviours or unclear tricks, but rather we should try to make everything as explicit as possible. Simple and clear code is always easier to read and maintain.

Remember that writing good code requires time and experience. We can only get better by... writing code.

5.1.2 Functional Style

When writing code, it is very likely that in many occurrences we need to apply the same set of commands multiple times. For example, suppose we need to standardize our variables. We would write the required commands to standardize the first variables. Next, each time we need to standardize a new variable, we will need to rewrite the same code all over again or we copy and pasted the previous code making the required changes. We would end up with something similar to the following lines of code.

```
# Standardize variables
x1_std <- (x1 - mean(x1)) / sd(x1)
x2_std <- (x2 - mean(x2)) / sd(x2)
x3_std <- (x3 - mean(x3)) / sd(x3)
```

Rewriting the same code over and over again or, even worse, copying and pasting the same chunk of code are very inefficient and error-prone practices. In particular, suppose we need to modify the code to solve a problem or to fix a typo. Any change would require us to revise the entire script and to modify each instance of the code. Again, this is a very inefficient and error-prone practice.

To overcome this issue, we can follow a completely different approach by creating our custom functions. Considering the previous example, we can define, possibly in a separate script, the function `std_var()` that allows us to standardize a variable. Next, after we have loaded our newly created function, we can call it every time we need it. Following this approach, we would obtain something similar to the code below.

```
#----    my-functions.R    ----#

# Define custom function
std_var <- function(x){

  res <- (x - mean(x)) / sd(x)

  return(res)
}

#----    my-analysis-script.R    ----#

# Apply custom function
x1_std <- std_var(x1)
x2_std <- std_var(x2)
x3_std <- std_var(x3)
```

Now, if we need to make some change to our custom function, we can simply modify its definition and any change will be automatically applied to each instance of the function in our code. This allows us to easily develop the code efficiently and limit the possibility of introducing errors (really common when copying and pasting).

Following this approach, we obey the ***DRY*** (Don't Repeat Yourself) principle that aims at reducing repetitions in the code. Each time we find ourselves repeating the same code logic, we should not rewrite (or copy and paste) the same lines of code, but instead, we should create a new function and use it. By defining custom functions in a single place and then using them, we enhance code:

- **Maintainability.** If we need to fix an issue or modify some part of the code, we

do not need to run all over our scripts making sure we change all occurrences of the function. Instead, we only need to modify the function definition in a single place. This facilitates enormously code debugging and development.

- **Readability.** Applying the DRY principle, entire chunks of code are substituted by a single function call. In this way, we obtain a much more compact code that, together with the use of meaningful function names, improves readability.
- **Reuse.** The DRY principle encourages the writing of reusable code facilitating the development.

Now it should be clear that writing functions each time we find ourselves repeating some code logic has many advantages. However, we do not have to necessarily wait for code repetitions before writing a function. Even if a specific code logic is present only once, we can always define a wrap function to execute it, improving code readability.

For example, in most analyses, we need to execute some data cleaning or preprocessing. This step usually requires several lines of code and operations that make our analysis script messy and difficult to read.

```
# Data cleaning
my_data <- read_csv("path-to/my-data.csv") %>%
  select(...) %>%
  mutate(...) %>%
  gorup_by(...) %>%
  summarize(...)
```

To avoid this problem, we could define a wrap function in a separate script with all the operations required to clean the data and give it a meaningful name (e.g,. `clean_my_data`). Next, after we have loaded our custom function, we can use it in the analysis script to clean the data, improving readability.

```
#----    my-functions.R    ----#

# Define data cleaning function
clean_my_data <- function(file){
  read_csv(file) %>%
    select(...) %>%
    mutate(...) %>%
    gorup_by(...) %>%
    summarize(...)
}

#----    my-analysis-script.R    ----#

# Data cleaning
my_data <- clean_my_data("path-to/my-data.csv")
```

We followed a ***Functional Style***: break down large problems into smaller pieces and define functions or combinations of functions to solve each piece.

Functional style and DRY principle allow us to develop readable and maintainable code very efficiently. The idea is simple. Instead of having a unique long script with all the analysis code, we define our custom functions to run each step of the analysis in separate scripts. Next, we use these functions in another script to run the analysis. In this way, we keep all the code organized and easy to read and maintain. In the short term, this approach requires more time and may seem overwhelming. In the long term, however, we will be rewarded with all the advantages.

In Chapter [TODO: add ref], we describe possible methods to manage the analysis workflow. In the following sections, we provide general recommendations about writing functions, documentation, and testing.

5.1.2.1 Functions Good Practices

Here we list some of the main recommendations and aspects to take into account when writing functions:

- **Knowing your Beast.** Writing functions is more difficult than simply applying them. When writing functions we need to deal with environments and arguments evaluations (see Section 5.1.3.2), classes and methods (see Section 5.1.3.3), or other advanced aspects. This requires an in-depth knowledge of the specific programming language that we are using. Studying books about a specific programming language or consulting resources available online can help to improve our understanding of all the mechanisms underlying a specific programming language. At first, errors and bugs can be very frustrating, but this is also a great opportunity to improve our programming skills. Surely, Google and Stack Overflow will quickly become much-needed friends.

```
x <- sqrt(2)
x^2 == 2  # WTF (Why is This False?)
## [1] FALSE
```

```
all.equal(x^2, 2)
## [1] TRUE
```

```
# Not intuitive behaviour
round(1.5)
## [1] 2
round(2.5)
## [1] 2
```

- **Function Names.** The same recommendations provided for variable names apply also to function names. Thus, we need to choose meaningful names of appropriate

length adopting a consistent naming style (the `snake_case()` is usually preferred). In addition, function names should be verbs summarizing the function goal.

```
f()
my_data()
```

```
get_my_data()
```

- **Single Responsibility Principle.** Each function should achieve a single goal. Avoid creating complex functions used for multiple different purposes. These functions are more difficult to maintain.
- **Handling Conditions.** When using a function, we may need to handle different conditions. We could do that directly within the function. However, if the code becomes too complicated and it is difficult to understand which part of the code is evaluated, we may prefer to simply write separate functions. There are no absolute correct or wrong approaches, the only important thing is to favour readability and maintainability.

```
solve_condition <- function(x){

  # Initial code
  ...

  if (is_condition_A){
    ...
  } else {
    ...
  }

  # Middle code
  ...

  if (is_condition_B){
    ...
  } else {
    ...
  }

  # Final code
  ...

  return(res)
}
```

```r
solve_condition_A <- function(x){

  # All code related to condition A
  ...

  return(res)
}

solve_condition_B <- function(x){

  # All code related to condition B
  ...

  return(res)
}
```

- **From Small to Big**. Start defining small functions that execute simple tasks and then gradually combine them to obtain more complex functions. In this way, we can enhance the re-usability of smaller functions. However, remember that each function, even the most complex ones, should always achieve a single goal.
- **Avoid Hard-Coded Values.** To enhance function re-usability, we should avoid hard-coded values favouring instead the definition of variables and function arguments that can be easily modified according to the specific needs.

```r
format_perc <- function(x){

  perc_values <- round(x * 100, digits = 2)
  res <- paste0(perc_values, "%")

  return(res)
}
```

```r
format_perc <- function(x, digits = 2){

  perc_values <- round(x * 100, digits = digits)
  res <- paste0(perc_values, "%")

  return(res)
}
```

- **Checking Inputs.** Check if the function's arguments are correctly specified and handle exceptions by providing appropriate outputs or informative error messages.

Checks are less relevant in the case of projects where the code is only used internally to run the analyses (although it is still valuable as it helps us debug and prevents possible unnoticed errors). On the contrary, it is extremely important in the case of services or packages where users are required to provide inputs. Be ready for any kind of madness from humans.

```
safe_division <- function(x, y){

  res <- x / y

  return(res)
}

safe_division(x = 1, y = 0)
## [1] Inf
```

```
safe_division <- function(x, y){

  if(y == 0) stop("you can not divide by zero")

  res <- x / y

  return(res)
}

safe_division(x = 1, y = 0)
## Error in safe_division(x = 1, y = 0): you can not divide by zero
```

Of course, writing checks is time-consuming and therefore we need to decide when it is worth spending some extra effort to ensure that the code is stable.

- **Be Explicit.** How functions return the resulting value depends on the specific programming language. However, a good tip is to always make the return statement explicit to avoid possible misunderstandings.

```
get_mean <- function(x){
  sum(x) / length(x)
}
```

```
get_mean <- function(x){

  res <- sum(x) / length(x)
```

```
  return(res)
}
```

- **KISS.** Again we want to highlight that writing functions is not a competition where to show off our coding skills. We should always aim to write elegant and readable code. This does not mean squeezing everything into a single line of code. Remember, Keep It Simple (Stupid).
- **Files and Folders Organization.** We could collect all the functions within the same script. However, maintaining the code would be very difficult. A possible approach is to collect related or similar functions within the same script and name the script accordingly (e.g., `data-preprocessing.R`, `models.R`, `plots.R`, `utils.R`). Alternatively, each function can be saved in a separate script named by the function name (this is usually done for complex long functions). Finally, we can collect all our scripts in a single folder within our project.
- **WET (write everything twice).** This is the opposite of the DRY principle. We just want to highlight that it is not the end of the world if we do not strictly follow the DRY principle (or any other rule). We should always take the most reasonable approach in any specific situation without blindly following some guidelines.

[TODO: find better examples?]

Remember that writing good functions requires time and experience. We can only get better by... writing functions.

5.1.2.2 Documentation

Writing the code is only a small part of the work in creating a new function. Every time we define a new function, we should also provide appropriate documentation and create unit tests (see Section 5.1.2.3).

Function documentation is used to describe what the function is supposed to do, provides details about the function arguments and outputs, presents function special features, and provides some examples. We can document a function by writing multiple lines of comments right before the function definition or at the beginning of the function body.

Ideally, the documentation of each function should include:

- **Title.** One line summarizing the function goal.
- **Description.** A detailed description of what the function does and how it should be used. In addition, we can create multiple sections to discuss the function's special features or describe how the function handles particular cases. Note that we can also provide links to external relevant documentation.
- **Arguments.** A list with all the function arguments. For each argument, provide details about the expected data type (e.g., string, numeric vector, list, specific

object class) and describe what the parameter is used for. We should also discuss the parameter default values, possible different options and effects.
- **Outputs.** Describe the function output specifying the data type (e.g., string, numeric vector, list, specific object class) and what the output represents (important in the case of a function returning multiple elements).
- **Examples.** Most of the time an example is worth a thousand words. Providing simple examples, we clarify what the function does and how it should be used. Moreover, We can also present specific examples showing function special features or particular use cases.

Thus, for example, we could create the following documentation.

```
#----    format_perc    ----

# Format Values as Percentages
#
# Given a numeric vector, return a string vector with the values formatted
# as percentage (e.g., "12.5%"). The argument `digits` allows specifying
# the rounding number of decimal places.
#
# Arguments:
# - x : Numeric vector of values to format.
# - digits: Integer indicating the rounding number of decimal places
#           (default 2)
#
# Output:
# A string vector with values formatted as percentages (e.g., "12.5%").
#
# Examples:
# format_perc(c(.749, .251))
# format_perc(c(.749, .251), digits = 0)

format_perc <- function(x, digits = 2){

  perc_values <- round(x * 100, digits = digits)
  res <- paste0(perc_values, "%")

  return(res)
}
```

Let's discuss some general aspects of writing documentation:

- **Generating documentation.** Most programming languages have specific rules to create documentation that can be automatically parsed and made available in

the function help pages or autocompletion hints. Alternatively, dedicated packages are usually provided to facilitate documentation generation. Therefore, it is worth checking the documentation best practices of our preferred programming language and sticking to them.

- **Time consuming.** Creating function documentation is time-consuming, and for this reason unfortunately it is often neglected. However, without documentation, we severely compromise the maintainability of our code.
- **Documenting applications and packages.** In the case of applications or packages, where our functions are expected to be used by others, documentation is the most important aspect. No one will use our application or package if there are no instructions. In this case, we really need to put some extra effort into creating excellent functions' documentation, vignettes, and online resources to introduce our application or package and provide all the details.

  Moreover, in the case of open-source projects, documentation should not be limited to the exported functions (i.e., functions directly accessed by the users), but it should include internal functions as well (i.e., utility functions used inside the app or package not directly accessed by the users). In fact, documenting all functions is required to facilitate the project maintenance and development by multiple contributors.
- **Documenting analyses.** In the case of projects where the code is only used to run the analyses, documentation may seem less relevant. This is not true. Even if we are the only ones working on the code, documentation is always recommended as it facilitates code maintainability. Although we do not need the same level of detail, spending a few extra hours documenting our code is always worth it. The future us will be very grateful for this.

5.1.2.3 Unit Tests

We may think that after writing the functions and documenting them we are done. Well... no. We still miss unit tests. ***Unit Tests*** are automated tests used to check whether our code works as expected.

For example, consider the following custom function to compute the mean.

```
#----    get_mean    ----
get_mean <- function(x){

  res <- sum(x) / length(x)

  return(res)
}
```

We can write some tests to evaluate whether the function works correctly.

```
#----    Unit Tests    ----

# Test 1
stopifnot(
  get_mean(1:10) == 5.5
  )

# Test 2
stopifnot(
  get_mean(c(2,4,6,8)) == mean(c(2,4,6,8))
  )
```

Let's discuss some general aspects of unit tests:

- **Which and how many tests?** For the same function, we can write many unit tests to evaluate its behaviour in different conditions. For example, we can check that for some fixed inputs the function returns the expected outputs, but we can also check whether the function manages exceptions according to expectations (e.g., returning specific values, error messages or warnings). Ideally, we should write enough tests to cover all possible scenarios.
- **Organizing tests.** Usually, we collect all related unit tests into a separate script and we save all scripts used for unit tests in a dedicated folder (e.g., `tests/`), ready to be run.
- **Manage tests.** Note that most programming languages have specific packages and functions that allow us to create unit tests and automatically run them. Therefore, we should check unit tests' best practices of our preferred programming language and stick to them.
- **Time consuming.** Writing unit tests takes a lot of time. Therefore, we may wonder if all this is worth all the effort. Well, the short answer is YES. Unit tests are the only thing that allows you to keep control over our code during the development. If we have only a couple of functions, we can easily deal without unit tests checking on our own that everything works as expected. But what happens if, instead, we have many dozens of functions and functions are used inside other functions? How can we be sure that a small change will not have an unexpected effect somewhere in our code leading to problematic errors? Well, the answer is unit tests. If we write unit tests, we can automatically check that everything works as expected without the worry of breaking the code during the development.
- **Testing applications and packages.** Unit tests are mandatory when developing an application or a package. In this case, we should pay particular attention to testing our code against any kind of madness users are capable of. Think the unthinkable, human *"creativity"* is endless. Only in this way we can build human-proof code.
- **Testing analysis.** When the code is used only to run an analysis, instead, unit tests may seem less relevant. In this case, functions are applied to fixed data and

thus we do not have to deal with unexpected conditions. Nevertheless, unit tests are still important to ensure that small changes in the code do not lead to unexpected problems. In the case of analysis, we could define unit tests based on a small portion of the data checking that the same results are obtained. In this way, we can develop our project keeping everything under control.

- **Fail to fail.** We highlight that the biggest problem is not when the code fails with an error. In this case, the issue is clear and we can work to solve it. The biggest problem is when the code runs without errors but, for some unexpected reasons, we do not obtain the correct results. Unfortunately, there are no simple solutions to this problem. Only in-depth knowledge of the specific programming language we are using and its specificities can help us prevent these issues.

Now, we have understood the importance of documenting and testing our functions to enhance code maintainability. In an ideal world, each line of code would be documented and tested. But, of course, this happens only in the ideal world and the reality is very far from this. Most of the time, documentation is limited and tests are only a dream. When choosing what to do, we should evaluate the trade-off between short-term effort and long-term advantages. In small projects, all this may be recommended but not necessary. In long term projects when maintainability is a real issue, however, we should put some real effort into documenting and testing. Again, the future us will be very grateful.

5.1.3 Advanced

In this section, we introduce some more advanced programming aspects that we may have to deal with when defining functions. These topics are complex and highly dependent on the specific programming language. Therefore we do not aim to provide a detailed description of each argument. Instead, we want to offer a general introduction to these topics providing simple definitions that can help us begin to familiarize ourselves with these advanced concepts.

5.1.3.1 Performance

In some projects or analyses, we may need to run some computational heavy tasks (e.g., simulations). In these cases, performance becomes a fundamental aspect and our code needs not only to be readable and maintainable but also efficient. Here we discuss some general aspects to take into account when we need efficient code in terms of speed.

- **For Loops.** For loops are used to apply the same set of instructions over all the elements of a given object. Unfortunately, for loops have a bad reputation of being slow. In reality, for loops are not slow per se. What makes for loops slow are usually bad coding practices. In particular, the most common issue is failing to pre-allocate the objects used inside the loop (for example to save the results).

  Let's consider a case where we need to execute a function (`add_one()`) over each element of our vector. A common but very bad practice is to grow objects inside the

loop. For example, in the function below, we are saving the newly obtained value by combining it with the previously obtained results. This is an extremely inefficient operation as it requires copying the whole vector of results at each iteration. As the length of the vector increases, the program will be slower and slower.

```
bad_loop <- function(x){

  res <- NULL

  for (i in seq_along(x)){

    value <- add_one(x[i])

    res <- c(res, value) # copy entire vector at each iteration
  }

  return(res)
}
```

Some programming languages provide specific functions to allow "adding" an element to an object without copying all its content. In these cases, we should take care in choosing the right functions. However, a commonly recommended approach is to pre-allocate objects used inside the loop. This simply means we need to create objects of the required size before we start the loop.

For example, in our case, first, we initialize the vector `res` of length equal to the number of iterations outside of the loop. Next, we store the obtained values at each iteration inside the vector.

```
good_loop <- function(x){

  # Initialize vector of the required length
  res <- vector(mode = "numeric", length = length(x))

  for (i in seq_along(x)){

    value <- do_stuff(x[i])

    res[i] <- value # assign single value
  }

  return(res)
}
```

Differences in performance will be greater as the number of iterations increases. Let's compare the two loops over 1000 iterations. The difference is incredible.

```
x <- 1:1e4 # vector with 1000 elements

# Bad loop
microbenchmark::microbenchmark(bad_loop(x))
## Unit: milliseconds
##         expr      min       lq    mean   median       uq      max neval
##  bad_loop(x) 148.9659 157.2734 166.442 161.2933 169.2989 255.4373   100

# Good loop
microbenchmark::microbenchmark(good_loop(x))
## Unit: milliseconds
##          expr      min      lq     mean   median       uq      max neval
##  good_loop(x) 4.382927 4.57817 6.053194 4.783873 5.016338 106.0503   100
```

Another important tip to improve loop performance is to limit computations at each iteration to what is strictly necessary. All elements that are constant between iterations should be defined outside the loop.

So, do not be afraid of using loops. They are not slow if we write them correctly.

- **Vectorized Operators.** We have just seen that for loops are used to perform operations on the elements of an object. Most programming languages, however, also provide specific functions that atomically apply operations over all the elements of an object. These are called *Vectorized Operators*.

  Let's consider the simple case of adding two vectors of the same length. Without vectorized operators, we would need to write a for loop as in the below function.

```
add_vectors <- function(x1, x2){

  res <- vector(mode = "numeric", length = length(x1))

  # Add element by element
  for (i in seq_along(x1)){
    res[i] <- x1[i] + x2[i]
  }

  return(res)
}
```

  Let's see how this for loop compares to the analogue vectorized operator.

```r
# vectors with 1000 elements
x1 <- 1:1e4
x2 <- 1:1e4

# Element by element operation
microbenchmark::microbenchmark(add_vectors(x1, x2))
## Unit: microseconds
##                 expr     min       lq     mean   median       uq      max
##  add_vectors(x1, x2) 777.917 814.8215 926.6311 841.1075 933.425 6385.063
##  neval
##    100

# Vectorized operation
# - In R the `+` operator is vectorized
microbenchmark::microbenchmark(x1 + x2)
## Unit: microseconds
##     expr    min     lq     mean  median     uq    max neval
##  x1 + x2 44.668 46.505 49.36991 47.4465 49.005 113.46   100
```

The difference is incredible. Note that this is not because for loops are slow, but rather because vectorized operators are super fast. In fact, vectorized operations are based on really efficient code usually written in compiled languages and run in parallel (see next point). This is what makes vectorized operators so fast and efficient.

So, if we want to improve performance, we should always use vectorized operators when available.

- **Compiled and Interpreted Languages.** To run a program, the source code written in a given programming language needs to be translated into machine code that can be executed by the processor. How this translation occurs differs between compiled and interpreted programming languages.

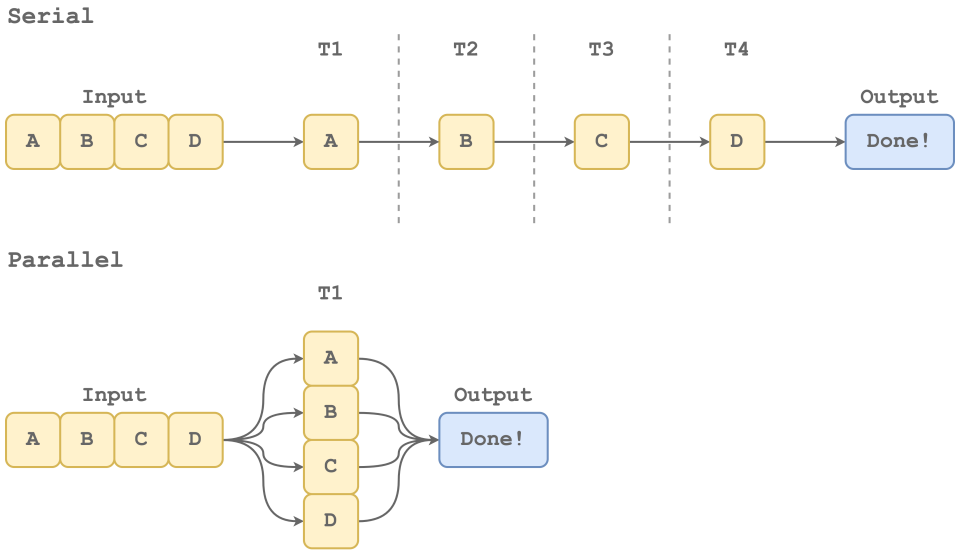
  In *Compiled Languages* (e.g., C or C++), the source code is translated using a compiler before we can execute it. The compilation process is slow and it is required each time we make changes to the source code. Once compiled, however, the code can be simply loaded and executed in a very fast and efficient way.

  In *Interpreted Languages* (e.g., R or Python), the source code is translated at execution time by an interpreter. This allows us to modify the source code at any time and immediately run it. However, the resulting code is slower and less efficient.

  So, interpreted languages are much more flexible and ideal when we write and execute code iteratively, but they are slower. On the contrary, compiled languages are very fast and efficient but they require to be compiled first. Therefore, when performance is important, we should use compiled code. However, this does not

mean that we necessarily have to write code in compiled languages, we can simply check if there are available libraries that implement compiled code for our needs. In fact, many interpreted programming languages provide libraries based on compiled code to execute specific tasks very efficiently.

- **Parallel and Serial Processing.** Depending on the specific task, we can improve performance by running it in parallel. In *Serial Processing*, a single task is executed at a time. On the contrary, in *Parallel Processing*, multiple tasks are executed at the same time.



Parallel processing, allows us to take advantage of the multiple processors available on our machine to execute multiple tasks simultaneously. If our program involves the execution or repetitive independent computations, parallel processing can help us to step up in terms of performance. However, parallelization is an advanced topic that needs to be applied appropriately. In fact, there are many aspects to take into account. For example, not all tasks can be parallelized and the overall costs of parallelizing processes may be higher than the benefits.

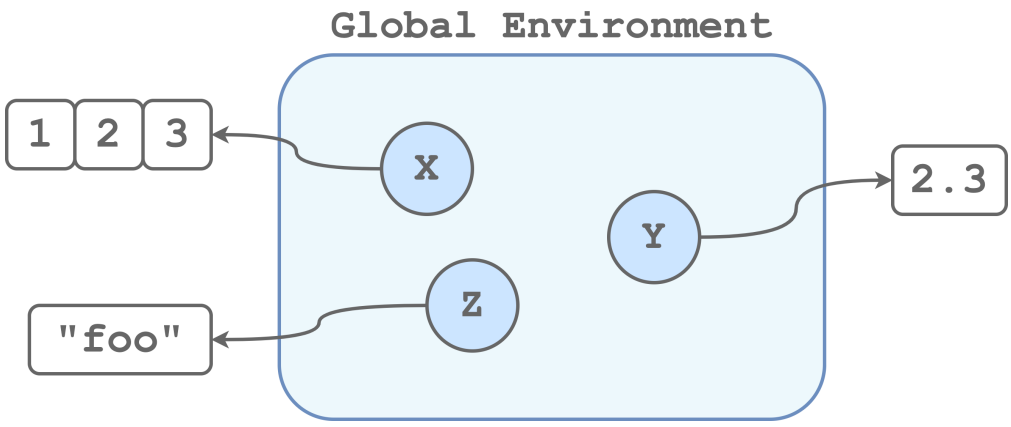
So, parallelization is a wonderful world that can help us to reach incredible levels of performance but we need to use it with care.

To summarize, when performance is an issue, we should check that our code is written efficiently. In particular, we should always use vectorized operators if available and follow best practices when writing for loops. Next, if we really need to push the limits, we can consider compiled code and parallelization. These are very advanced topics that require specific knowledge. Fortunately, however, many dedicated libraries allow us to implement these solutions more easily. Get ready to break the benchmark!

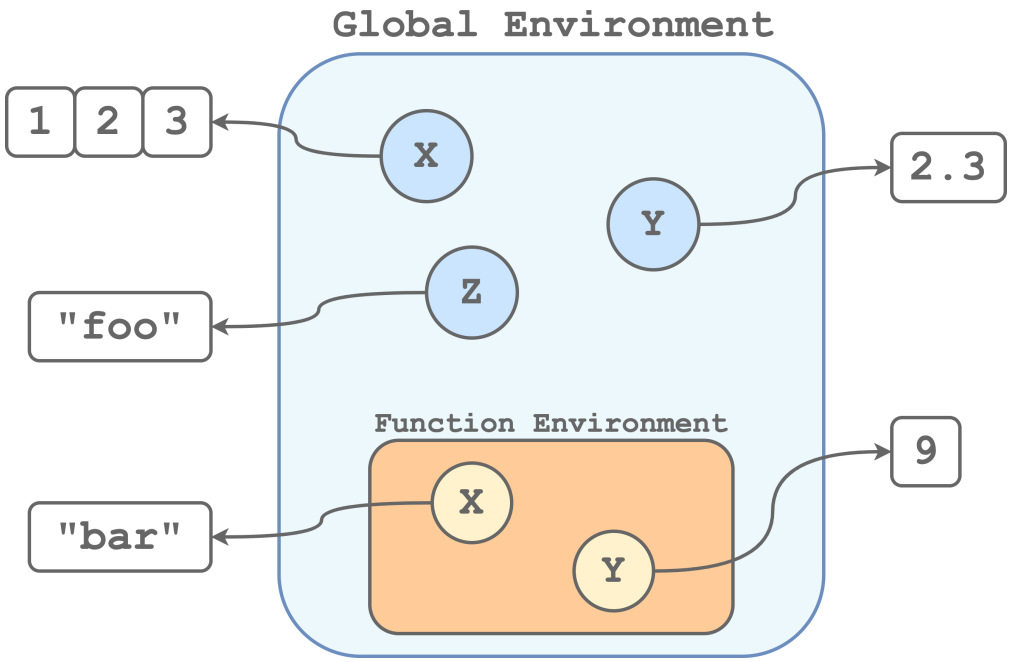
5.1.3.2 Environments

Another important aspect that we need to master when writing functions, is how a specific programming language evaluates and accesses variables from within a function. Again, we do not aim to provide a detailed discussion of this argument. Instead, we just want to introduce these concepts allowing us to familiarize ourselves with these relevant aspects that need to be understood more in-depth by studying programming language dedicated resources.

- **Environments.** An environment is a virtual space where all objects, variables, and functions we created during our session are available. More technically, an environment is a collection that associates a set of names with a set of values. Usually, the main environment of our session is called *"Global Environment"*. Note that inside an environment names must be unique. So, for example, we can represent the global environment in the following way.



When we execute a function, commands are executed from inside a new environment. In this way, we avoid conflicts between objects with the same name in the Global Environment and the function environment. For example, in the following case, we have a variable `X` pointing to a three-element vector in the global environment and another variable named `X` in the function environment pointing to a string.



So each time we run a function, commands are executed inside a newly created environment with its own set of objects. Note that the function environments are actually inside the global environment and therefore they are also referred to as *child-environment* and *parent-environment* respectively. If we call a function, within another function, we would obtain a function environment inside another function environment. We can think about it like a Russian doll.

- **Global Variables.** Global variables are objects defined in the parent environment. Global variables can be accessed from within the child-environment. For example, consider the following case.

```
global_var <- "I am Global!"

my_fun <- function(){
  return(global_var)
}

my_fun()
## [1] "I am Global!"
```

We created `global_var` in the global environment. Next, we defined the `my_fun()` that simply prints the object `global_var`. Note that, although there is no object `global_var` defined in the function, we do not get an error. Instead, the function looks in the parent environment for the variable and we obtain its value.

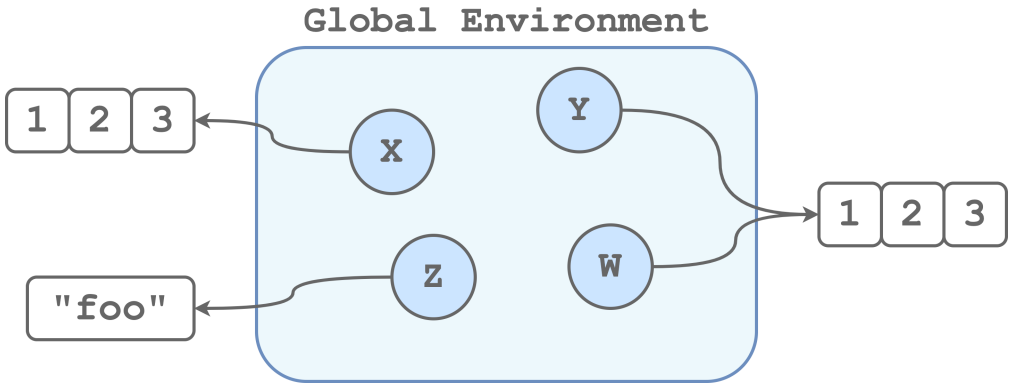
Note that we can not modify global variables from within a function as any attempt will simply create a local variable.

```
global_var <- "I am Global!"

my_fun <- function(){
  global_var <- "I am local"

  return(global_var)
}

my_fun()
## [1] "I am local"
global_var
## [1] "I am Global!"
```

There are specific commands used to modify global variables from within a function, but this is usually a deprecated practice because we may affect other functions that depend on those variables.

Global variables can be used to specify constants and settings that affect the whole analysis. A common practice is to capitalize global variables to distinguish them from the local variables. However, global variables should be used with care preferring to explicitly pass function arguments instead.

- **Aliasing.** Occasionally, we create an object as a copy of another object, for example, `x = y`. This apparent simple command can actually lead to very different behaviours depending on the specific programming language. In fact, some programming languages distinguish between *copying* (i.e., creating a new variable that points to an **independent** copy of the same object) and *aliasing* (i.e., creating a new variable that points to the **same** object). In the first case, we would obtain two independent objects, whereas, in the second case, we obtain two variables pointing to the same object as presented in the figure below.



If we are not aware of this difference, we could easily end up with serious problems. Let's consider the following example in R (aliasing is not allowed) and in Python (aliasing is allowed).

```
#----    R Code    ----

# Create objects
x = c(1, 2, 3)
y = c(1, 2, 3)
w = y

w[3] <- 999 # change a value

# Check values
x
## [1] 1 2 3
y
## [1] 1 2 3
w
## [1]   1   2 999
```

In R, changes to an element of `w` do not affect y.

```
#----    Python Code    ----

# Create objects
x = [1,2,3]
y = [1,2,3]
w = y

w[2] = 999 # change a value

# Check values
x
## [1, 2, 3]
y
## [1, 2, 999]
w
## [1, 2, 999]
```

In Python, changes to an element of `w` do also affect y. This example is not intended to scare anyone but simply to highlight the importance of having in-depth knowledge and understanding of the programming language we are using.

5.1.3.3 Classes and Methods

At some point in programming, we will need to deal with classes and methods. But what do these two strange words mean? Let's try to clarify these concepts.

- **Class.** Each object we create belongs to a specific family of objects depending on their characteristics. We call this family a *"class"*. More precisely, a class is a template that defines which are the specific characteristics of the objects belonging to that class. This template is used to create objects of a given class and we say that an object is an instance of that class. Of course, we can create multiple instances (i.e., multiple objects) of the same class.
- **Methods.** Each class has their own methods, that is, a set of actions objects of a specific class can execute or functions we can apply to manipulate the object itself.

So, why are classes and methods so important? Classes and methods allow us to organize our code efficiently and enhance reusability. For example, if we find ourselves relying on some specific data structure in our program, we can create a dedicated class. In this way, we can improve the control over the program by breaking down the code into small units and by specifying different methods depending on the object class.

Now, classes and methods are typical of the Object-Oriented Programming approach rather than the Functional Programming approach. Let's briefly introduce these two approaches.

- **Object-Oriented Programming.** According to the object-oriented programming approach, we define *object classes* to represent everything we need in our program. Note that these object classes include not only how we create the specific objects but also the code of all the methods (i.e., procedures and actions) we can use to manipulate these objects. These methods are a characteristic of the object class itself and we can define different methods for different object classes.
- **Functional Programming.** According to the functional programming approach, we create programs by applying and composing only *pure functions*. Similarly to mathematical functions, pure functions are deterministic, that is, given a fixed set of inputs they return the same output. Thus, pure functions do not produce side effects nor are affected by other external variables or states. We can think of pure functional programming as an extreme version of the functional style, introduced in Section 5.1.2, where there are no objects but only functions.

Less extreme applications of functional programming allow object classes. In this case, methods are not characteristics of the object itself but are functions defined separately from the object. To clarify this difference, suppose we have an object `todo_list` with a list of tasks we need to complete today and we have a method `whats_next()` that returns which is the next task we need to complete. In an object-oriented programming approach, the method is directly invoked from the object, whereas, in a functional programming approach, we would apply the method as a function to the object.

```
# Object Oriented Programming
todo_list.whats_next()

## Write the paper

# Functional Programming
whats_next(todo_list)

## Have a break
```

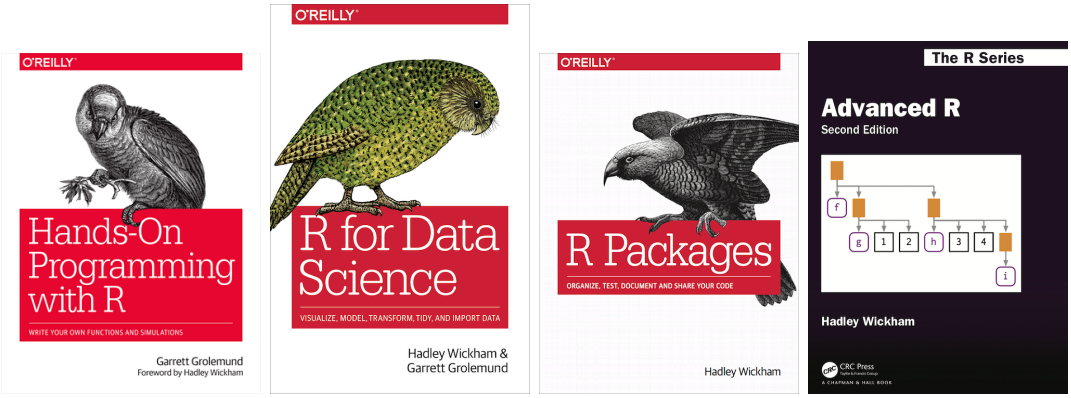
Note that in most object-oriented programming languages, methods are accessed using the dot character (`"."`). For this reason, we should always separate words in object names using `snake_case` and not `snake.case`.

Finally, the two approaches are not mutually exclusive and actually, most programming languages support both approaches, usually leading to a mixed flavour of object classes and functions working together. However, different programming languages could favour one of the two approaches. For example, Python is commonly considered an object-oriented programming language and R a functional programming language, although both support both approaches.

5.2 R Coding

In this section, we discuss further recommendations regarding writing code in R. Now, learning how to program in R is a huge topic that would require an entire new book on its own (or probably more than one book). Therefore, we prefer to provide references to useful resources and packages.

There are many books available on-line covering all the aspects of the R programming language. In particular, we highlight the following books (freely available on-line) ordered from beginner to advanced topics:



- **Hands-On Programming with R (https://rstudio-education.github.io/hopr).** This book covers the basics of R (i.e., data types, functions, programs, and

loops). This book is different from all the resources of the *"learning statistics with R"* kind, as it focuses on R from a programming perspective rather than applying it to run statistical analyses. Therefore, it is perfect to build fundamental knowledge about basic programming concepts that are otherwise overlooked in other more applied books. Remember R is not simply a statistical software, but it is a real programming language.

- **R for Data Science (https://r4ds.had.co.nz).** The tidyverse bible. We address the tidyverse vs Base R discussion in the Box below. However, no one can deny the importance of tidyverse which has led to a small revolution in R creating a wonderful ecosystem of packages. This book covers the process of wrangling, visualising, and exploring data using the tidyverse packages. However, along with the chapters, it also discusses many general important aspects of programming that we commonly have to deal with (i.e., regular expressions and relational data). Therefore, although it is more of an applied book, it helps us to deal with many common issues when working on real data projects.

- **R Packages (https://r-pkgs.org).** When writing functions, we start to deal with many subtle aspects of R. The best way to start understanding what is going on behind the scenes is to start developing our own packages. This book covers all the details and mechanisms of R packages and it will become our best friend if we want to publish a package on CRAN. Of course, we do not always need to create an actual stand-alone package. However, using the R package project template allows us to get the advantage of many useful features (e.g., documentation and unit tests) that can help us develop our projects. In Section 5.2.2, we further discuss these aspects.

- **Advanced R (https://adv-r.hadley.nz).** Finally the *"one book to rule them all"*. This book covers all the black magic and secrets of R. All the topics are very advanced and discussed in detail from a programming perspective. Usually, we end up reading parts of this book when facing strange bugs or issues. If you have never heard about lexical scoping, lazy evaluation, functional, quasiquotatio, and quosure, well... you will have lots of fun.

In the next sections, we briefly discuss coding good practices in R and how we can develop projects according to a functional style approach.

Details-Box: Tidyverse VS Base R

Regarding the tidyverse vs Base R discussion, we want to share our simple opinion. We love tidyverse. This new ecosystem of packages allows us to write readable

code in a very efficient way. However, tidyverse develops very quickly and many functions or arguments may become deprecated or even removed in the future. This is not an issue per se, but it can make it hard to maintain projects in the long term. So what should we use tidyverse or Base R? Our answer is...depends on the specific project aims.

- **Analyses Projects.** In the case of projects related to specific analyses, we recommend using tidyverse. Wrangling, visualising, and exploring data in the tidyverse is very easy and efficient (and fun!). To overcome the issues related to the frequent changes in the tidyverse, we recommend using the `renv` R package to manage the specific package versions (see Chapter [TODO: add ref]). In this way, we can have all the fun of tidyverse without worrying about maintainability in the long term.
- **Apps and Packages.** In the case of projects that aim to create apps or packages, we recommend using Base R. In this case, we may have less control over the specific package versions installed by the users. Therefore, we prefer to build our code with as few dependencies as possible, relying only on stable packages that rarely change. In this way, we limit issues of maintainability in the long term.

5.2.1 Coding Style

The same general good practices described in Section 5.1.1 apply also to R. In addition, there are many *"unofficial"* coding style recommendations specific to R. We should always stick to the language-specific style guidelines. In some cases, however, there are no strict rules and thus we can create our style according to our needs and personal preferences. When creating our personal style, remember that we want a consistent styling that enhances code readability.

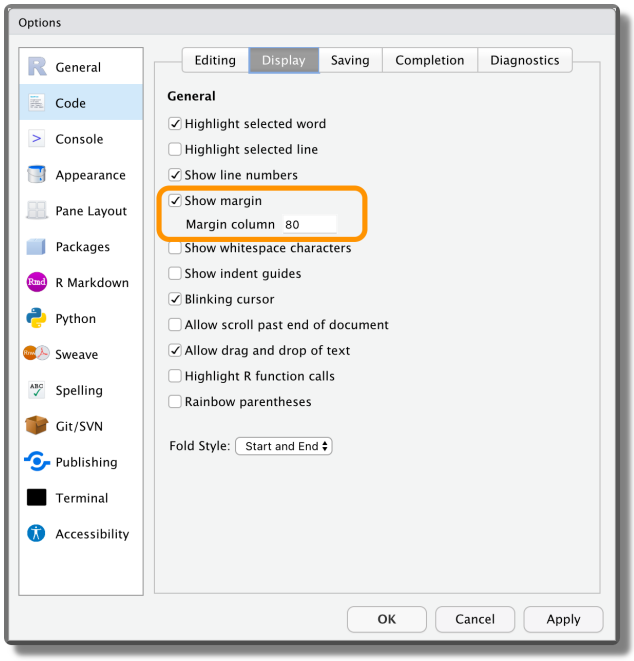
For more details about R coding style, consider the two following resources:

- A R Coding Style Guide by Iegor Rudnytskyi (https://irudnyts.github.io/r-coding-style-guide).
- The tidyverse style guide (https://style.tidyverse.org).

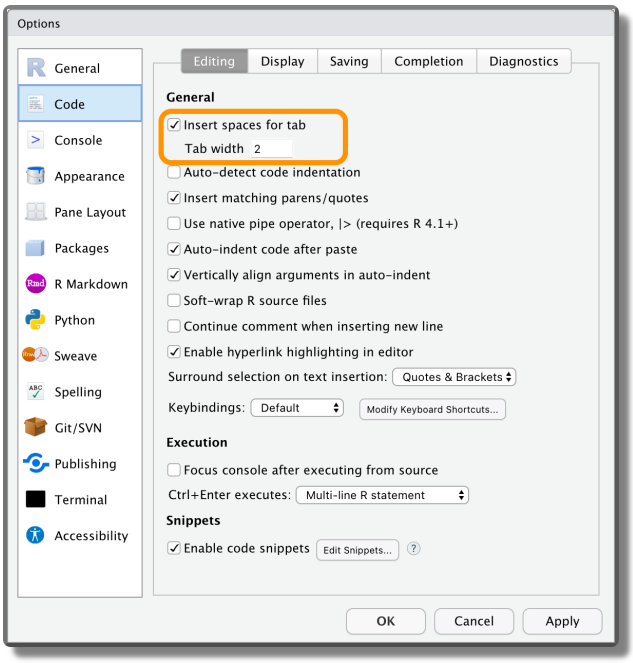
Here we review only a few aspects:

- **Names.** The preferred naming style for both variables and functions is `snake_case`.
- **Assign Function.** In R we can assign values to an object using the symbols `<-` or `=`. We agree with the tidyverse preference of using `<-` for assignment, as it explicitly indicates the direction of the assignment (`x = y`, are we assigning `y` to `x` or the contrary? With `x <- y` there are no doubts). However, both are fine, just pick one and stick with it.

- **Line length.** In RStudio, we can display the margin selecting *"Show margin"* from *"Tools -> Global Options -> Code -> Display"* and specifying the desired margin column width (default 80).



- **Indentation.** In RStudio, we can substitute Tabs with a fixed number of spaces. Select *"Insert spaces for tab"* from *"Tools -> Global Options -> Code -> Editing"* specifying the desired number of spaces (usually 2 or 4).



- **Logical Values.** Always write `TRUE` and `FALSE` logical values instead of the respective abbreviations `T` and `F`. `TRUE` and `FALSE` are reserved words, whereas `T` and `F` are not. This means that we can overwrite their values, leading to possible issues in the code.

```
# Check value
TRUE == T
## [1] TRUE

# Change values
TRUE <- "Hello"
## Error in TRUE <- "Hello": invalid (do_set) left-hand side to assignment

T <- "World"
T
## [1] "World"

TRUE == T
## [1] FALSE

# If you want to be evil
T <- FALSE
FALSE == T
## [1] TRUE
```

- **RStudio Keyboard Shortcuts**. RStudio provides many useful keyboard shortcuts to execute specific actions https://support.rstudio.com/hc/en-us/articles/200711853-Keyboard-Shortcuts-in-the-RStudio-IDE. Familiarizing with them can facilitate our life while coding. For example:

  - Clear console: `Ctrl+L`
  - Delete Line: `Ctrl+D` (macOS `Cmd+D`)
  - Insert assignment operator: `Alt+-` (macOS `Option+-`)
  - Comment/uncomment current line/selection: `Ctrl+Shift+C` (macOS `Cmd+Shift+C`)
  - Reindent lines: `Ctrl+I` (macOS `Cmd+I`)
  - Reformat Selection: `Ctrl+Shift+A` (macOS `Cmd+Shift+A`)
  - Reflow Comment: `Ctrl+Shift+/` (macOS `Cmd+Shift+/`)
  - Show help for function at cursor: `F1`
  - Show source code for function at cursor: `F2` ([TODO: check on my mac is `command + click` function]).
  - Find in all project files: `Ctrl+Shift+F` (macOS `Cmd+Shift+F`)
  - Check Spelling: F7

5.2.2 R Package Project

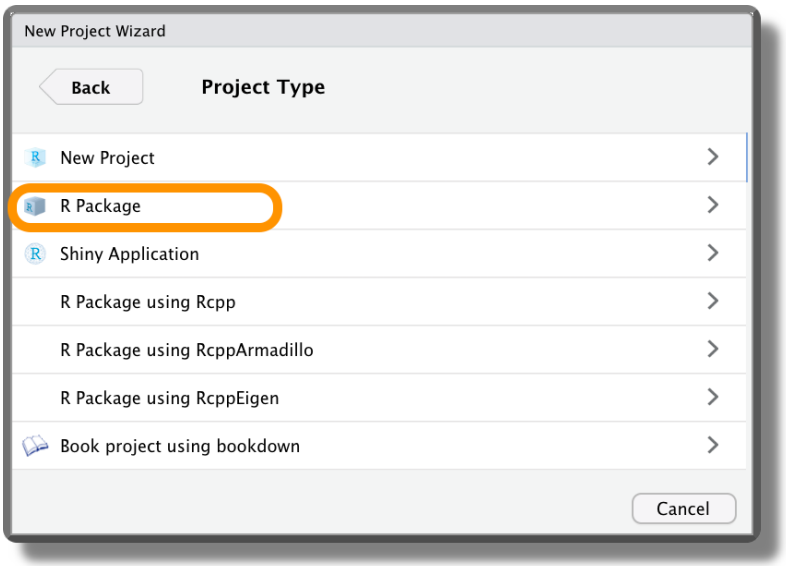
Adopting a functional style approach, we will create lots of functions and use them in the analyses. To organize our project files, we can save all the scripts containing the function definitions in a separate directory (e.g., `R/`) and source all them at the beginning of the main script used to run the analyses. This approach is absolutely fine. However, if we want to optimize our workflow, we should consider organizing our project as it was an R package.

Using the structure of R Packages, we can take advantage of specific features that facilitate our lives during the development process. Note that creating (and publishing) an actual R package requires dealing with many advanced aspects of R and this whole process may be overwhelming for our projects. However, we do not need to create a real R package to take advantage of the development tools. Simply by organizing our project according to the R Package Project template, we can already use all the features that help us manage, document, and test our functions.

In the next sections, we introduce the basic aspects of the R Package Project template. Our aim is to simply provide an introduction highlighting all the advantages to encourage learning more. For a detailed discussion of all aspects, we highly recommend the R Packages book (https://r-pkgs.org).

5.2.2.1 R Package Structure

To create a project using the structure of R Packages, we simply need to select *"R Package"* as Project Type when creating a new project. Alternatively, we can use the function `devtools::create()` indicating the path.



The basic structure of an R package project is presented below.

```
- <pkg-name>/
    |-- .Rbuildignore
    |-- DESCRIPTION
    |-- NAMESPACE
    |-- <pkg-name>.Rproj
    |-- man/
    |-- R/
    |-- tests/
```

In particular, we have:

- **`.Rbuildignore`**, a special file used to list the files we do not want to include in the package. If we are not interested in creating a proper package, we can ignore it.
- **`DESCRIPTION`**, this file provides metadata about our package (see Section 5.2.2.3). This is a very important file used to recognize our project as an R package, we should never delete it.
- **`NAMESPACE`**, a special file used to declare the functions that are exported and imported by our package the files we do not want to include in the package. If we are not interested in creating a proper package, we can ignore it.
- **`<pkg-name>.Rproj`**, the usual `.Rproj` file of each RStudio project.
- **`man/`**, a directory containing the functions documentation (see Section 5.2.2.4).
- **`R/`**, a directory containing all the function scripts.
- **`tests/`**, a directory containing all the unit tests (see Section 5.2.2.5).

5.2.2.2 The `devtools` Workflow

So, what is special about the R Package Project template? Well, thanks to this structure we can take advantage of the workflow introduced by the `devtools` R package. The `devtools` R package provides many functions to automatically manage common tasks during the development. In particular, the main functions are:

- **`devtools::load_all()`.** Automatically load all the functions found in the `R/` directory.
- **`devtools::document()`.** Automatically generate the function documentation in the `man/` directory.
- **`devtools::test()`.** Automatically run all unit tests in the `tests/` directory.

These functions allow us to automatically execute all the most common actions during the development. In particular all these operations have dedicated keyboard shortcuts in RStudio:

- Load All (devtools): `Ctrl+Shift+L` (macOS `Cmd+Shift+L`)
- Document Package: `Ctrl+Shift+D` (macOS `Cmd+Shift+D`)
- Test Package: `Ctrl+Shift+T` (macOS `Cmd+Shift+T`)

Using these keyboard shortcuts, the whole development process becomes very easy and smooth. We define our new functions and immediately load them so we can keep on working on our project. Moreover, whenever it is required, we can create the function documentation and check that everything is fine by running unit tests.

The R Packages book (https://r-pkgs.org) describes all the details about this workflow. It could take some time and effort to familiarize ourselves with this process but the advantages are enormous.

5.2.2.3 DESCRIPTION

The `DESCRIPTION` is a special file with all the metadata about our package and it is used to recognize our project as an R package. Thus, we should never delete it.

A `DESCRIPTION` looks like this,

```
#-----    DESCRIPTION    ----#

Package: <pkg-name>
Title: One line description of the package
Version: the package version number
Authors@R:  # authors list
    c(person(given = "name",
             family = "surname",
             role = "aut",  # cre = creator and maintainer; aut = other authors;
```

```
              email = "name@email.com"),
      ...)
Description: A detailed description of the package
Depends: R (>= 3.5) # Specify required R version
License: GPL-3      # Our  prefered license
Encoding: UTF-8
Imports: # list of required packages
Suggests: # list of suggested packages
Config/testthat/edition: 3
RoxygenNote: 7.1.2
VignetteBuilder: knitr
URL:     # add useful links to online resources or documentation
```

The `DESCRIPTION` file is particularly important if we are creating an R package. However, it can be used for any project to collect metadata, list project dependencies, or add other useful information.

Note that the `DESCRIPTION` file follows specific syntax rules. To know more about the `DESCRIPTION` file, see https://r-pkgs.org/description.html.

5.2.2.4 Documentation with `roxygen2`

The `roxygen2` R package allows us to create functions documentation simply by adding comments with all the required information right before the function source code definition. `roxygen2` will process our source code and comments to produce the function documentation files in the `man/` directory.

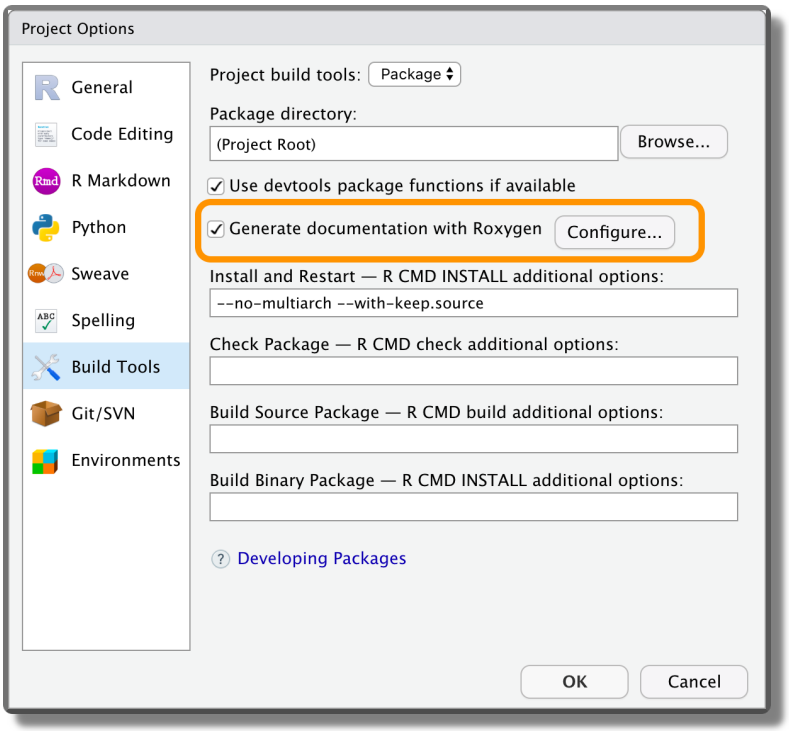
`roxygen2` assumes a specific structure and uses specific tags to correctly produce the different parts of the documentation. Below is a simple example of documenting a function using `roxygen2`. Note the use of `#'` instead of `#` to create the comments and the special tags (`@<tag-name>`) used to specify the different documentation components.

```
#----    format_perc    ----

#' Format Values as Percentages
#'
#' Given a numeric vector, return a string vector with the values
#' formatted as percentage (e.g.,"12.5%"). The argument `digits` allows
#' to specify the rounding number of decimal places.
#'
#' @param x Numeric vector of values to format.
#' @param digits Integer indicating the rounding number of decimal places
```

```r
#'                (default 2)
#'
#' @return A string vector with values formatted as percentages
#'  (e.g.,"12.5%").
#'
#' @examples
#' format_perc(c(.749, .251))
#' format_perc(c(.749, .251), digits = 0)
#'

format_perc <- function(x, digits = 2){

  perc_values <- round(x * 100, digits = digits)
  res <- paste0(perc_values, "%")

  return(res)
}
```

We can use the keyboard shortcut `Ctrl+Alt+Shift+R` (macOS `Cmd+Option+Shift+R`) to automatically insert the Roxygen skeleton for the documentation.

To create the function documentation using `roxygen2`, we need to select the *"Generate documentation with Roxygen"* box from the *"Project Options"* > *"Build Tools"* (remember Project Options and not Global Options).



Next, we can run `devtools::document()` (or `Ctrl+Shift+D`/ macOS `Cmd+Shift+D`) to automatically create the function documentations. Now, we can use the common help function `?<function-name>` (or `help(<function-name>)`) to navigate the help page of newly created functions.

To learn all the details about documenting functions using `roxygen2`, consider:

- `roxygen2` official documentation (https://roxygen2.r-lib.org).
- R Packages dedicated chapter (https://r-pkgs.org/man.html).

5.2.2.5 Unit Tests with `testthat`

The `testthat` R package allows us to create and run unit tests for our functions. In particular, `testthat` provides dedicated functions to easily describe what we expect a function to do, including catching errors, warnings, and messages.

To create the unit tests using `testthat`, we need to use dedicated functions following specific folders and file structures. Below is a simple example of a unit test using `testthat`.

```
#----    testing format_perc    ----

test_that("check format_perc returns the correct values", {
```

```
  # numbers
  expect_match(format_perc(.12), "12%")
  expect_match(format_perc(.1234, digits = 1), "12.3%")

  # string
  expect_error(format_perc("hello"))

})
```

Once the tests are ready, we can automatically run all the unit tests using the function `devtools::test()` (or `Ctrl+Shift+T`/ macOS `Cmd+Shift+T`).

To learn all the details about unit tests using `testthat`, consider:

- `testthat` official documentation (https://testthat.r-lib.org/).
- R Packages dedicated chapter (https://r-pkgs.org/tests.html).

Documentation-Box

R Coding

- Hands-On Programming with R
  https://rstudio-education.github.io/hopr
- R for Data Science
  https://r4ds.had.co.nz
- R Packages
  https://r-pkgs.org
- Advanced R
  https://adv-r.hadley.nz

R Style

- General coding style
  https://code.tutsplus.com/tutorials/top-15-best-practices-for-writing-super-readable-code--net-8118
- A R Coding Style Guide by Iegor Rudnytskyi
  https://irudnyts.github.io/r-coding-style-guide
- The tidyverse style guide
  https://style.tidyverse.org
- RStudio Keyboard Shortcuts
  https://support.rstudio.com/hc/en-us/articles/200711853-Keyboard-Shortcuts-in-the-RStudio-IDE

R packages

- DESCRIPTION file
  https://r-pkgs.org/description.html

`roxygen2`

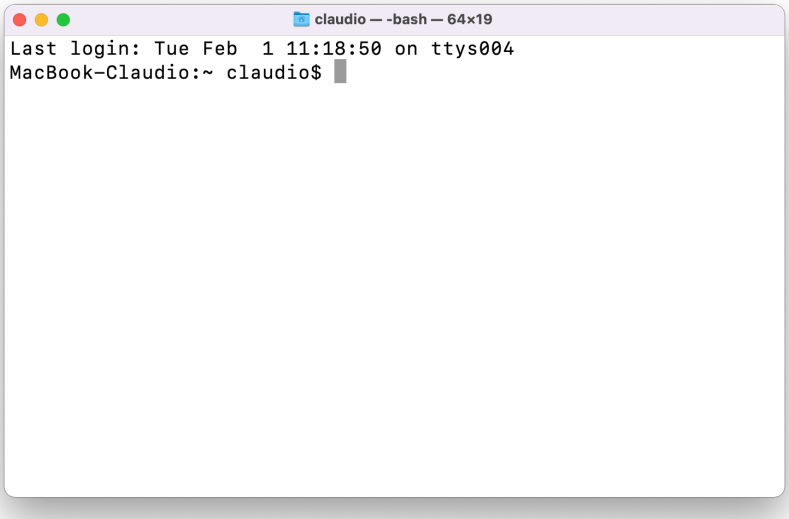
- `roxygen2` official documentation
  https://roxygen2.r-lib.org
- R Packages dedicated chapter
  https://r-pkgs.org/man.html

`testthat`

- `testthat` official documentation
  https://testthat.r-lib.org/
- R Packages dedicated chapter
  https://r-pkgs.org/tests.html

6
Terminal

In the era of the *metaverse*, *User Experience Design*, and beautiful *Graphic User Interfaces*, the terminal with its text-based command-line interface looks like a prehistoric tool from some old 80s sci-fi film.



We may wonder why we still need such an old school tool? Well, the terminal is a very powerful tool. By using the terminal we can easily manage our files and execute complex

operations very efficiently. Moreover, although most software provides some graphical user interfaces, advanced functionalities may only be available through the command-line interface. The bottom line is, *"when the going gets tough the terminal gets opening"*.

Therefore, although it may seem overwhelming at first, we need to learn the basics of the terminal to later use more advanced tools that are introduced in the following Chapters. In particular, the terminal is required to use Git (see Chapter **??**) and Docker (see Chapter **??**).

In this chapter, we provide a minimal guide to the terminal introducing the main concepts and basic commands to manage and manipulate files. In particular, we refer to the Bash command language used on Unix systems (see Section 6.1.1).

This is just a minimal introduction to familiarise less-experienced users with the terminal. A complete overview of the terminal is beyond the aims of this chapter. However, we encourage everyone to spend more time learning how to properly work with the terminal and the Bash command language. This will help a lot to improve our skills as a programmer. We highly recommend these two tutorials (please, read them!):

- **Terminal Tutorial** https://ryanstutorials.net/linuxtutorial/
- **Bash Scripting Tutorial** https://ryanstutorials.net/bash-scripting-tutorial/

Warning-Box: Create Backups

Remember that the terminal is a very powerful tool and... *"with great power comes great responsibility"*. The terminal allows us to access important files and settings on which our machine relies on.
Fortunately, most fundamental stuff requires admin permissions, but still, we can easily mess up things on our machine and some actions may not be reversible. Therefore, any time we use the terminal we should be very careful about the commands we run and aware of the possible consequences
A good tip is to always keep some up-to-date backups of our machine, just in case we mess things up.

6.1 What is a Terminal?

The terms command-line, terminal, and shell are often used interchangeably to refer to the same thing: *"using text-based commands to interact with the operative system"*.

To be precise, however, they are not exactly the same thing. Let's define them:

- **Command Line Interface (CLI).** A CLI is a text-based interface that allows users to interact with a program by typing command lines. The program executes the command and possible responses are returned in a text-based format. For example, both Python and R can be used through CLI. On the other hand, a **Graphical User Interface (GUI)** is a point-and-click interface that allows users

to interact with a program through menus, icons and buttons. Both have pros and cons, so they are used according to the different needs. In particular, CLIs are very efficient to automate tasks by using scripts. For more details, see https://www.computerhope.com/issues/ch000619.htm.

- **Terminal (or Console).** The graphical window with a command-line interface allows us to interact with the shell.
- **Shell.** The command line interpreter that processes the commands, communicates with the Operative System and returns the results.

For more details, see https://www.geeksforgeeks.org/difference-between-terminal-console-shell-and-command-line/.

Although these subtle differences, also in this book the terms command line, terminal, and shell are used interchangeably. Thus, for example, when we say "using the terminal" we mean "typing commands processed by a shell command-line interpreter in a terminal window."

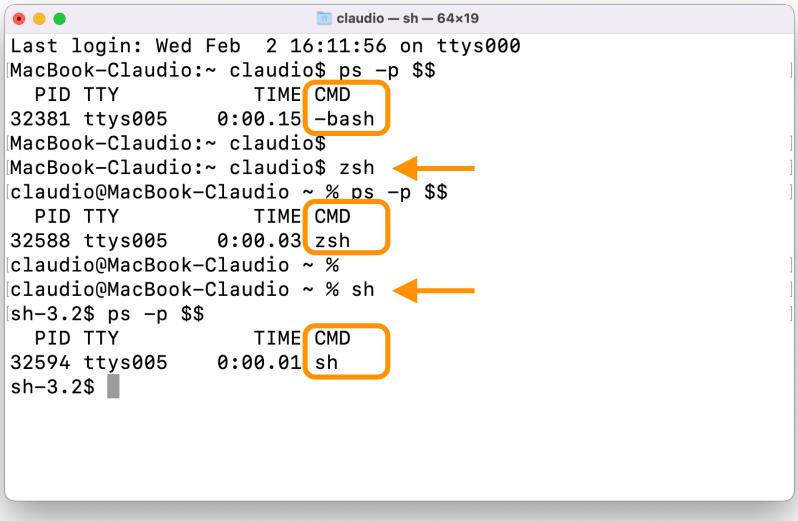
6.1.1 Different Shells

Different shell command line interpreters are available depending on the Operative System. The most popular shells are reported in Table 6.1. For more details about the different shells, see https://www.servertribe.com/difference-between-cmd-vs-powershell-vs-bash.

Table 6.1: Shell Interpreter according to Operative System

Shell Interpreter
Windows
Command Prompt (CMD)
PowerShell
...
Unix System (macOS and Linux)
sh (Bourne shell)
Bash (Bourne again shell)
Zsh (Z shell)
...

We can install multiple shells on the same machine. When we open the terminal, the default shell is automatically used. We can change the terminal default shell by changing the system settings. Alternatively, we can simply change the shell in the current session by typing the desired shell name. For example, ignoring all the jargon that will be explained in Section 6.3, you can see how starting from the default shell `bash`, we can use the commands `zsh` and `sh` to change the current shell (arrows indicate the commands and the current shell is indicated in the rectangles).



Although most shells work similarly, each one has its unique commands and specific features. In particular, Unix shells and Windows shells are based on very different frameworks leading to important differences. Learning one of the two would give us only a limited intuition of how the other works. For an example of command differences, see https://www.geeksforgeeks.org/linux-vs-windows-commands/. Therefore, we need to choose which shell command language to learn first.

We decided to use Bash for two main reasons. First, Bash is now available on all main Operative Systems (macOS, Linux, and Windows). In fact, Windows recently introduced the *Windows Subsystem for Linux (WLS)* that allows us to run Linux directly on Windows meaning that we can use any Unix based shell (see Section 6.2.1). Second, Bash is one of the most diffused and supported shell command languages and it is the default shell in most Linux distributions. As most web servers and online services are Linux-based, learning Bash would allow us to easily work with all these advanced tools (e.g., Docker).

Details-Box: Programming Language vs Command Language

Bash language and other shell languages, in general, are referred to as command languages rather than programming languages. Why this difference?
Shell languages are considered *super-languages* used to communicate with the Operative System. They are intended to interact with everything and execute any task by managing calls to other programs. This is their real power, the possibility to create complex applications using different programs.
Hypothetically, shell languages can be used on their own to implement any arbitrary algorithm. However, they usually lack features to facilitate this job. You

would need to implement everything yourself or relay to call some other external program.
For more detail, see https://stackoverflow.com/questions/28693737/is-bash-a-programming-language.

6.2 Install Bash

Let's see how to install Bash depending on the Operative System.

6.2.1 On Windows

With the introduction of the *Windows Subsystem for Linux (WLS)* (now at its second version), Windows supports Linux natively. This means we can now install Linux distributions directly on Windows allowing us to use any Unix based shell. See official documentation at https://docs.microsoft.com/en-us/windows/wsl/about.

The WLS install procedure depends on your Windows version. Check your Windows version following instructions at https://support.microsoft.com/en-us/windows/which-version-of-windows-operating-system-am-i-running-628bec99-476a-2c13-5296-9dd081cdd808.

- **Command-Line Procedure**, for Windows 10 version 2004 or higher (Build 19041 and higher) or Windows 11 follow instructions at https://docs.microsoft.com/en-us/windows/wsl/install.
- **Manual Procedure**, for older builds of Windows follow instructions at https://docs.microsoft.com/en-us/windows/wsl/install-manual.

The instructions will guide you through the installation of WLS (version 1 or 2 depending on your Windows version) with a specific Linux distribution. If you follow the manual procedure, choose Ubuntu as the Linux distribution (this is already the default in the command line install procedure).

Done?... Congrats! You have just installed Linux on a Windows machine. Now we can launch a Bash terminal session simply by opening Ubuntu as we would do with any other application. Note that we can also start the Bash shell from other terminals by simply typing `bash`.

Now let's briefly clarify a few important things without going into details. We have installed both Windows and Linux on our machine but they actually *"live"* in two different places. They can communicate with each other, but we have to be careful about what we do. Windows and Linux are completely different Operative Systems and they manage files differently (they use different files metadata). Therefore, if we modify Linux files from Windows, this could result in corrupted or damaged files. Fortunately, there is a safe way to do that:

- **Accessing Linux file system from Windows** safely via `\\wsl$\<DistroName>\` (e.g. `\\wsl$\Ubuntu\home\<username>\<folder>`)

Note that the reverse process is not a problem, we can access Windows files from Linux without issues. To do that:

- **Accessing Windows file system from Linux** via `/mnt/<drive>/<path>` (e.g. `/mnt/c/Users/<username>/Desktop`)

For more details, see https://devblogs.microsoft.com/commandline/do-not-change-linux-files-using-windows-apps-and-tools/ and https://devblogs.microsoft.com/commandline/whats-new-for-wsl-in-windows-10-version-1903/.

Instructions-Box: Shells and Terminals on Windows

The most common shells on Windows are Command Prompt (CMD) and PowerShell. These are installed in Windows by default and they come with their own dedicated terminal application. By opening one of the two terminal applications, a new terminal window is opened with the specific shell interpreter.

Admin

Occasionally we may need to open the terminal with administrator privileges. This means opening the terminal with permission to make major changes to the system. To do that we need to right-click on the specific terminal application and select *"Run as administrator"*.

Windows Terminal

Windows recently introduced Windows Terminal, a modern terminal application for using shells like Command Prompt, PowerShell, and Windows Subsystem for Linux (WSL). Windows Terminal has many features and custom settings to facilitate our work (see https://docs.microsoft.com/en-us/windows/terminal/).
We highly recommend using Windows Terminal as your terminal. To install Windows Terminal and specify default shell settings, see https://docs.microsoft.com/en-us/windows/terminal/install.

6.2.2 On macOS

In macOS, the Bash shell is already installed. From macOS 10.15 Catalina, however, the default shell for new users will be Zsh. Zsh behaves very similarly to Bash so both are fine. Nevertheless, if you want to use the Bash shell, simply run the command `bash` in the terminal.

To find the Terminal app, press `command + space` and type Terminal in the search field. Alternatively, we can find the Terminal app with Finder in the `Applications/Utilities` folder. See https://support.apple.com/en-in/guide/terminal/apd5265185d-f365-44cb-8b09-71a064a42125/mac.

Once the Terminal is open, run the following command `xcode-select -install`. This will install different tools that are useful when working with the Terminal (e.g., git). For more details, see https://www.freecodecamp.org/news/install-xcode-command-line-tools/.

Instructions-Box: Brew the Missing Package Manager for macOS

Brew is a free and open-source package manager that allows us to easily install software and applications (it is the corresponding of `apt` in Linux distributions). The advantages of using Brew are that all the dependencies and environmental settings are automatically managed, saving us from lots of troubles. Moreover, using Brew we can safely update or remove software and applications with a single command. We highly encourage you to start using Brew to install all software and applications.
To install Brew, follow the instructions at https://mac.install.guide/homebrew/3.html. Note that for macOS versions older than Catalina, instructions are slightly different. In particular, the command to run for older versions is

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/
```

Moreover, on Apple Silicon machines depending on your default Shell (Zsh or Bash), you are required to edit different profile files (`.zprofile` or `.bash_profile`).

Default Shell: Zsh vs Bash

With macOS 10.15 Catalina, the default shell on macOS changed from Bash to Zsh. We could think that this is due to some improved features of Zsh over Bash. This is partially true, but there is also another part of the story.
The available version of Bash on macOS is `3.2` from 2007 while the currently available version is `5.1`. Why is there this big difference? Well, `3.2` was the last release under GPLv2 whereas subsequent releases moved to the GPLv3 which is incompatible with Apple's policies. For more details, see https://scriptingosx.com/2019/06/moving-to-zsh/.
We can install the updated version of Bash using Brew and set the default shell according to our preference. To do that follow instructions at https://itnext.io/upgrading-bash-on-macos-7138bd1066ba. Note that depending on the CPU

(Intel vs ARM) the path would be different (`/usr/local/bin/<shell-name>` vs `/opt/homebrew/bin/<shell-name>`; see https://apple.stackexchange.com/a/434278/356551).

6.2.3 On Linux

If you are using the Ubuntu Linux distribution, you are already using Bash. Actually, most Linux distributions use Bash. However, if this is not the case, you can simply install it from the command line.

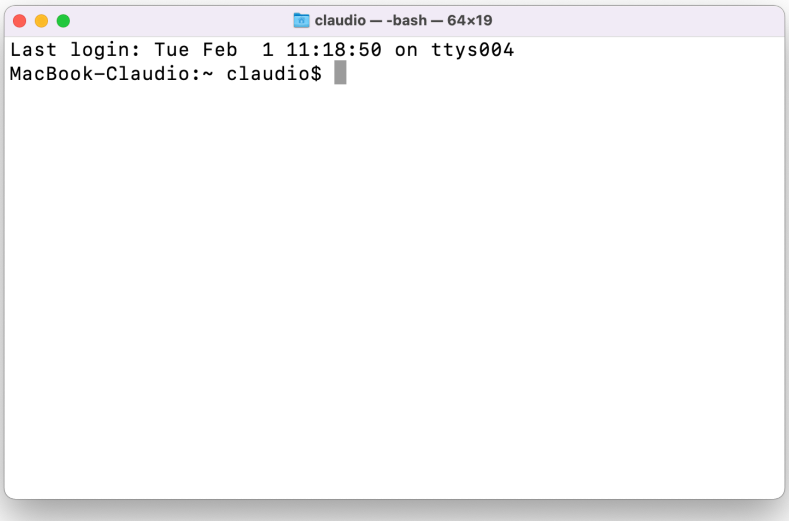
6.3 Get Started

Now that we all have bash (or another shell) installed, let's run our first commands. Here we only explain how to move between directories and execute simple file manipulations. For complete tutorials, we highly recommend:

- **Terminal Tutorial** https://ryanstutorials.net/linuxtutorial/
- **Bash Scripting Tutorial** https://ryanstutorials.net/bash-scripting-tutorial/

6.3.1 Prompt and Commands

When we open the Terminal, a similar short text message will appear at the start of the command line.



This is the *"prompt"* indicating that the system is ready for the next command. The prompt displays some information depending on the actual shell used and the specific settings. In this case, we have:

```
HOST_NAME:CURRENT_DIRECTORY USER_NAME$
```

To run a command, we simply type the desired command and then we press `Enter`. Commands usually have the following structure:

```
Command [Options] Argument1 Argument2 ...
```

- **`Command`** is the specific command name.
- **`Options`** are used to obtain some command specific behaviour. Options are optional (in Bash documentation this is usually indicated by square brackets "`[ ]`"). Options usually are single letters preceded by a single dash "`-`" (e.g., `-a` or `-l`) or they can be words preceded by a double-dash "`--`" (e.g., `--all` or `--recursive`). Sometimes, options have both forms, the single letter and the single word (e.g., `--directory` is the same as `-d`). Although the single word is more readable, the single letter form is usually preferred as less verbose and it is possible to concatenate multiple options (e.g., `-ad` stands for `-a -d` or `--all --directory`). Note that options are also called *"flags"* (in reality there is a subtle difference, an option can itself take an argument whereas a flag does not).
- **`Argument*`** are the command specific arguments. Arguments are separated by space, if an argument is formed by multiple words we need to wrap it inside quotes (e.g., `my argument` is considered as two separate arguments, `"my argument"` is considered as a single argument). Note that we don't always need to specify all the command arguments as some arguments may have default values and others may be optional.

For example, in the command `ls -la Desktop/my-project`, `ls` is the specific command, `-la` are two options, and `Desktop/my-project` is the command argument.

Bash has many implemented commands. However, we may need other software to execute some specific operations. To do that we simply specify the required software followed by the desired command and its options and argument

```
Software Command [Options] Argument1 Argument2 ...
```

For example, in the command `docker build -t my-image:1.0.0 Desktop/my-project`, `docker` is the software (see Chapter [TODO: add ref]), `build` is the specific command, `-t my-image:1.0.0` is an option, and `Desktop/my-project` is the command argument.

Warning-Box: Avoid Spaces in Names

As we already have pointed out, command arguments are separated by spaces. This is particularly relevant in the case of paths. If file or directory names include spaces, we need to use quotes.
As discussed in Chapter 3.1.2, a good tip is to avoid spaces in file or directory names. We can use the single dash ("-") or underscore ("_") character to concatenate multiple words.
Another tip is to avoid special characters (e.g., accented characters or other symbols) they only create trouble.

6.3.2 Navigating the File System

Oh yes! Finally, after all these words let's get into some actions. Firstly, as we no longer have our mouse and nice GUIs, we need to learn how to navigate within our machine from the terminal.

To check our current position, use the command `pwd` (print working directory). The repose is printed on the Terminal. In this case, we are in our home directory.

```
$ pwd
/Users/myname
```

To see all the files available in the current directory, use the command `ls` (list directory contents). By default, `ls` prints only the names.

```
$ ls
Applications                    Documents                   Library
Desktop                                                     Downloads
```

To get more information, specify the option `-l` (long listing format). Another useful option is `-a` to also list hidden files (i.e., file or folder that starts with a dot character usually used to specify preferences and settings).

```
ls -l
total 0
drwx------@   4 myname  staff   128 Oct 25  2018 Applications
drwx------@  27 myname  staff   864 Feb  3 14:41 Desktop
drwx------@  10 myname  staff   320 Jan 17 09:41 Documents
drwx------@  10 myname  staff   320 Feb  3 09:56 Downloads
drwx------@ 104 myname  staff  3328 Jan 13 21:30 Library
drwxr-xr-x+   5 myname  staff   160 Sep  7  2018 Public
```

The initial `d` indicates that these are all directories, next we get information about permissions and ownerships (see Chapter [TODO: add ref]), the file size in bits (use

options `-h` for human-readable file sizes), file modification time, and finally the actual file name.

Let's say we now want to see what is inside the `Desktop/` directory. To do that, we simply specify it as an argument of `ls`.

```
$ ls -l Desktop/
drwxr-xr-x@ 17 claudio  staff    23544 Dec 17 17:09 Courses
drwxr-xr-x@ 17 claudio  staff    14590 Dec 17 17:09 Presentations
-rw-r--r--   1 claudio  staff     5928 Feb  3 14:41 Repot.pdf
-rw-r--r--@  1 claudio  staff      160 Jan 18 11:10 TODO-list.txt
```

To move to another directory, use the command `cd` (change directory) specifying the desired location. We move `Courses/Open-Science` and check the current working directory.

```
$ cd Desktop/Courses/Open-Science
$ pwd
/Users/myname/Desktop/Courses/Open-Science
```

To move back to the parent directory, we use the syntax `../`. In this case, if we want to return to the `Desktop` directory, we need to move back to two levels (`../../`).

```
$ cd ../../
$ pwd
/Users/myname/Desktop
```

Details-Box: Directory Structure

As we start using the terminal, we discover how all directories and files are actually organized in our computer and how they are managed by the operative system.
Modern graphical user interfaces with icons and buttons give a very misleading representation of how a computer is organized. We may think that the Desktop is the entrance of our computer and from there we can reach all the files simply by point-and-click actions.
Actually, the computer is organized into a **Hierarchical Directory Structure**. At the lowest levels, we find all the system files which we can access only with special permissions. At the upper level, we find all the files concerning the programs and applications installed which generally can be used by multiple users on the same computer. Finally, we find all the directories and files that concern a specific user. The desktop is simply one of the top-level folders and what you see on the screen is simply a graphical user interface that allows us to interact with

the computer.
When working with the terminal it is important to be aware of this hierarchical structure. Moreover, we also need to understand other aspects such as file metadata, ownership, and user permissions (see Chapter Files Permissions [Todo add link]). These aspects define the possible action we can execute with a specific file.

Trick-Box: Autocompletion and Command History

When working using the terminal, there is a lot of typing going on. To facilitate our life, we can use command auto-completion by pressing `Tab` (or double `Tab` to list all available options). This is very useful when writing paths and file names.
Moreover, using the up/down arrow keys, we can navigate the command history and select commands we have already executed making changes if required.

6.3.3 Modifying Files

We learned the basics of how to move ourselves inside the file system. Let's see now how we can manipulate files.

To create a new directory, use the command `mkdir` (make directory) specifying the name (and position). We create the directory `my-project` and we move inside it.

```
$ mkdir my-project
$ cd my-project
$ pwd
/Users/myname/my-project
```

To create a blank file, use the command `touch` specifying the name (and position). Note that this command can also be used to create hidden files (i.e., files that start with a dot character). We create the file `README`.

```
$ touch README
$ ls
README
```

We can check the file content using the command `cat` (concatenate; print file content on the screen) or `less` (visualize the file on the screen allowing to move up and down the page; to quit press `q`). The file is now empty so if we check its content nothing appears.

```
$ cat README
$
```

To add text to the file we can use a text editor (see Section 6.3.4). Alternatively, we can use the command `echo`. By default `echo` prints the desired message on screen, but using the syntax `echo "message" >> file` we can specify to add the message at the end of the desired file (Wow! Isn't this magic?!?).

```
$ echo "Hello World!"
Hello World!
$ echo "Hello World!" >> README
```

Now we can check the file content again using `cat`.

```
$ cat README
Hello World!
```

To move a file or a directory, use the command `mv` (move) specifying the source location and the destination. Let's move the file `Report.pdf` from the Desktop to the `my-project` directory.

```
$ mv ../Report.pdf .
$ ls
README                Report.pdf
```

Note that `.` is used to indicate here (the current location). The command `mv` can also be used to rename files and directories by specifying as destination the same initial directory, but with a different name.

To copy a file, use the command `cp` (copy) specifying the source location and the destination. Note that to copy a directory the option `-r`, which stands for recursive, is required. Let's create a copy of `Report.pdf` named `Report-copy.pdf`.

```
$ cp Report.pdf Report-copy.pdf
$ ls
README                Report-copy.pdf          Report.pdf
```

Finally to remove a file, use the command `rm` (or `rmdir` to remove a directory) specifying the file.

```
$ rm Report-copy.pdf
$ ls
README                Report.pdf
```

So far we described how to do simple operations using the terminal. We may wonder why we should bother with all this fuss when we could easily do the same operations using the common point-and-click interface? Well, this may be true if we need to do a single operation a single time, but where the Terminal shines is automation. If we need to repeat the same operation over multiple files or periodically over time, a few command lines would save us a lot of hours wasted in point-and-click menus.

Hopefully, this brief introduction has shed some light on the potential and utility of using the terminal and made you interested in learning more (or at least less afraid).

For complete tutorials, we highly recommend (if we do it for the third time there is a reason):

- **Terminal Tutorial** https://ryanstutorials.net/linuxtutorial/
- **Bash Scripting Tutorial** https://ryanstutorials.net/bash-scripting-tutorial/

Trick-Box: Exit

If something goes wrong and the session stays idle press `Ctrl + Shift + C` (or `Control + C` on macOS) to interrupt the session. Sometimes (e.g., when using the command `top` or `less`) you only need to press `q` to go back to the interactive session.
If nothing seems responsive, well probably we ended up in vim or nano (two text editors: see Section 6.3.4), do not panic. To quit from vim type `:wq` end press `Enter`. To quit from nano press `Ctrl + X` (or `Control + X` on Mac).

Command Cheatsheet: Bash

Here is a summary of all the Bash commands introduced so far.

```
#----    Navigating    ----#

pwd                         # Print working directory
cd      <directory>         # Change directory
ls      <directory>         # List files
    -l   # Long list with details
    -a   # List also hidden files and directories
    -h   # Return file size in readable units

#----    Modifying    ----#

mkdir   <directory>         # Create directory
touch   <file>              # Create file
mv      <source><dest>      # Move (or rename) file or directory
cp      <source><dest>      # Copy file
    -r   # For directories
rm      <file>              # Remove file
rmdir   <directory>         # Remove directory

#-----   Other    -----#

echo    <message>           # Print message in console
cat     <file>              # Print file in console
less    <file>              # Open file in a screen
```

6.3.4 Text Editors

Working with the Terminal, we realize that we can execute most tasks just using a few plain text files and a bunch of command lines.

To edit plain text files, we can use our preferred IDE (e.g., Visual-Studio-Code or RStudio) or other simple editors available in our OS (e.g., Notepad on Windows or TextEdit on macOS).

Alternatively, some text editors work directly from within the terminal. Two popular editors are:

- **`nano`** a *simple* editor. For a tutorial see https://www.howtogeek.com/howto/42980/the-beginners-guide-to-nano-the-linux-command-line-text-editor/.
- **`vi`** (or its improved version `vim`) is the most powerful editor. For a tutorial see https://ryanstutorials.net/linuxtutorial/vi.php.

These editors are quite different from any other common editor. They are powerful but to properly use them we need to know lots of commands and keyboard shortcuts.

The learning process can be quite challenging (not to say frustrating). Even being able to close them can be considered a great achievement. Consider this funny (but not too unrealistic) quote about vim:

> I've been using vim for about 2 years now, mostly because I can't figure out how to exit it. (source)

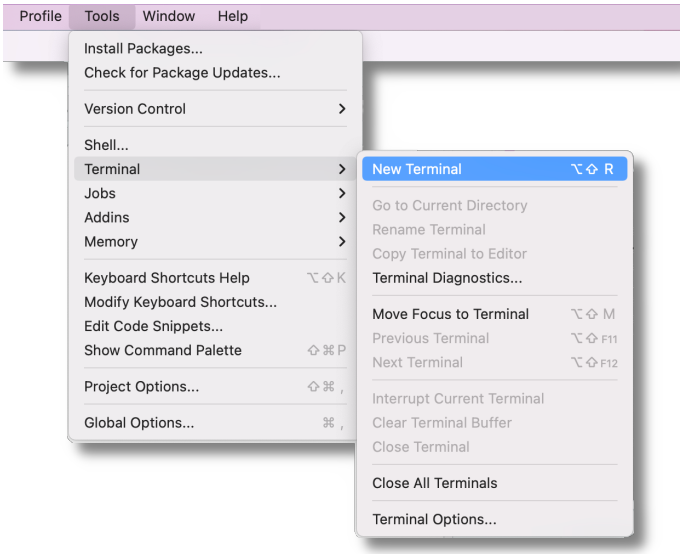
Just in case you went down the rabbit hole:

- To quit from `nano` press `Ctrl + X` (or `Control + X` on Mac)
- To quit from `vim` type `:w` (or `:wq` to save and exit) end press `Enter`.

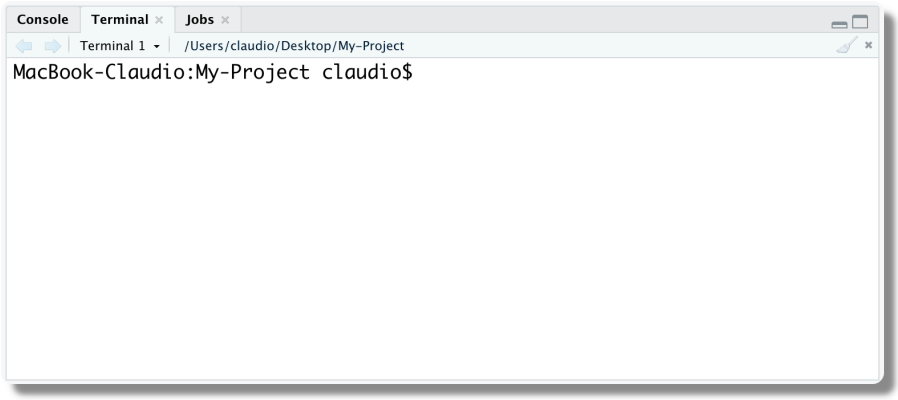
6.4 Terminal in RStudio

Most Integrated Development Environments (IDEs; e.g., RStudio or Visual Studio Code) provide a Terminal window from which we can interact with the system shell.

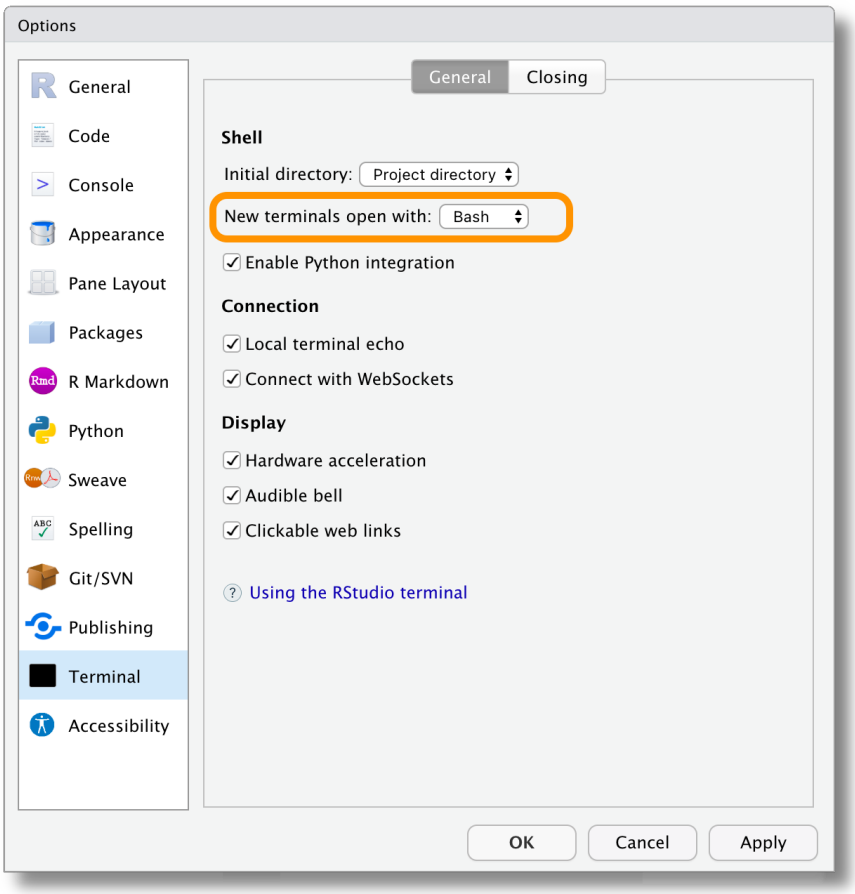
In RStudio, we can open a Terminal window selecting from *Tools* > *Terminal* > *New Terminal*.



The Terminal panel appears next to the Console panel.



We can select the default shell used (plus other custom settings) from the Terminal section in the Global Options.



For more details on the use of the Terminal in RStudio, see https://support.rstudio.com/hc/en-us/articles/115010737148-Using-the-RStudio-Terminal-in-the-RStudio-IDE.

 Documentation-Box

Terminal Tutorials

- Terminal Tutorial
  https://ryanstutorials.net/linuxtutorial/
- Bash Scripting Tutorial
  https://ryanstutorials.net/bash-scripting-tutorial/

Terminal Elements

- Command line vs. GUI
  https://www.computerhope.com/issues/ch000619.htm
- Difference between Terminal, Console, Shell, and Command Line
  https://www.geeksforgeeks.org/difference-between-terminal-console-shell-and-command-line/
- Difference Between CMD Vs Powershell Vs Bash
  https://www.servertribe.com/difference-between-cmd-vs-powershell-vs-bash
- Linux vs Windows Commands
  https://www.geeksforgeeks.org/linux-vs-windows-commands/
- Programming Language vs Command Language
  https://stackoverflow.com/questions/28693737/is-bash-a-programming-language.

Install Bash

Windows

- Windows Subsystem for Linux
  https://docs.microsoft.com/en-us/windows/wsl/about
- Do not change Linux files using Windows apps and tools
  https://devblogs.microsoft.com/commandline/do-not-change-linux-files-using-windows-apps-and-tools/
- What's new for WSL in Windows 10 version 1903?
  https://devblogs.microsoft.com/commandline/whats-new-for-wsl-in-windows-10-version-1903/
- Windows Terminal
  https://docs.microsoft.com/en-us/windows/terminal/

MacOS

- Open the Terminal
  https://support.apple.com/en-in/guide/terminal/apd5265185d-f365-44cb-8b09-71a064a42125/mac
- Install Xcode Command Line Tools
  https://www.freecodecamp.org/news/install-xcode-command-line-tools/
- Homebrew
  https://mac.install.guide/homebrew/3.html
- Moving to zsh
  https://scriptingosx.com/2019/06/moving-to-zsh/
- Upgrading Bash on macOS
  https://itnext.io/upgrading-bash-on-macos-7138bd1066ba

Text Editors

- Nano tutorial
  https://www.howtogeek.com/howto/42980/the-beginners-guide-to-nano-the-linux-command-line-text-editor/
- Vi tutorial
  https://ryanstutorials.net/linuxtutorial/vi.php

RStudio

- Terminal in RStudio
  https://support.rstudio.com/hc/en-us/articles/115010737148-Using-the-RStudio-Terminal-in-the-RStudio-IDE.

References

Buchanan, E. M., Crain, S. E., Cunningham, A. L., Johnson, H. R., Stash, H., Papadatou-Pastou, M., Isager, P. M., Carlsson, R., & Aczel, B. (2021). Getting Started Creating Data Dictionaries: How to Create a Shareable Data Set. *Advances in Methods and Practices in Psychological Science*, *4*(1), 2515245920928007. https://doi.org/10.1177/2515245920928007

Nosek, B. A., & Errington, T. M. (2020). What is replication? *PLOS Biology*, *18*(3), e3000691. https://doi.org/10.1371/journal.pbio.3000691