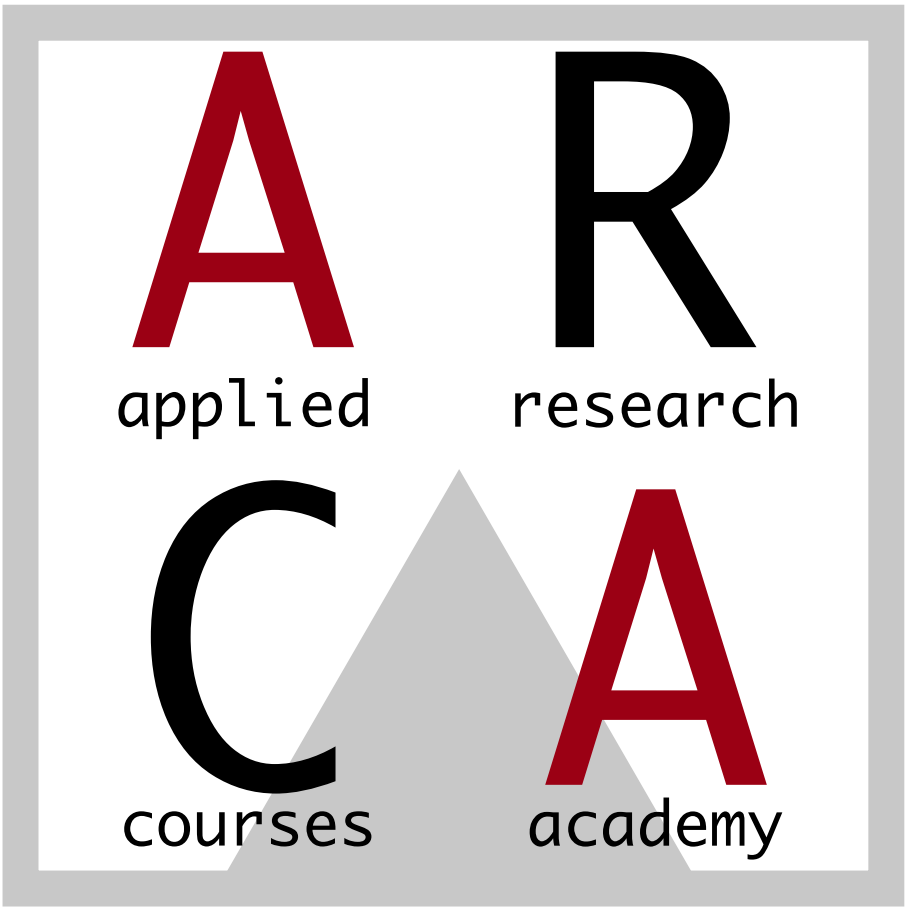
The Open Science Manual

Make Your Scientific Research Accessible and Reproducible

Claudio Zandonella Callegher and Davide Massidda

26 April, 2022 [last-updated]



Contents

Preface

> 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.
>
> "Science as Amateur Software Development" Richard McElreath (2020)
>
> https://youtu.be/zwRdO9_GGhY

Inspired by McElreath's words, this book aims to describe programming good practices and introduce common tools used in software development to guarantee the reproducibility of analysis results. We want to make scientific research an 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 Summary

In the book, we will learn to:

- Share our materials using the **Open Science Framework**
- Organize project files and data in a well structured and documented **Repository**
- Write readable and maintainable code using a **Functional Style** approach
- Use **Git** and **GitHub** for tracking changes and managing collaboration during the development

- Use dedicated tools for managing the **Analysis Workflow** pipeline
- Use dedicated tools for creating **Dynamic Documents**
- Manage project requirements and dependencies using **Docker**

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.

Examples and specific applications are based on the R programming language. However, this book provides recommendations and guidelines useful for any programming language.

About the Authors

During our careers, we both moved into the field of Data Science after a 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 (claudiozandonella@gmail.com). 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 (d.massidda@kode-solutions.net).

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.

View book source at GitHub repository https://github.com/arca-dpss/manual-open-science.

Cite

For attribution, please cite this work as:

Zandonella Callegher, C., & Massidda, D. (2022). The Open Science Manual: Make Your Scientific Research Accessible and Reproducible. https://arca-dpss.github.io/manual-open-science/

BibTeX citation:

```
@book{zandonellaMassiddaOpenScience2022,
  title = {The Open Science Manual: Make Your Scientific Research Accessible and Reproducibl
  author = {Zandonella Callegher, Claudio and Massidda, Davide},
  date = {2022},
  url = {https://arca-dpss.github.io/manual-open-science/}
}
```

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.
>
> "Science as Amateur Software Development" Richard McElreath (2020)
>
> https://youtu.be/zwRdO9_GGhY

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 present book aims to describe programming good practices and introduce common tools used in software development to guarantee the reproducibility of 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 7, we introduce Git software for tracking changes in any file during the development of our project.
- In Chapter 8, we introduce GitHub for managing collaboration using remote repositories.
- In Chapter 9, we discuss how to manage the analysis workflow to enhance results reproducibility and code maintainability.

- In Chapter 10 [work in progress], we introduce the main tools to create dynamic documents that integrate narrative text and code describing the advantages.
- In Chapter 11, we discuss how to manage our project requirements and dependencies (software and package versions) to enhance results reproducibility.
- In Chapter 12, we introduce Docker and the container technology that allows us to create and share an isolated, controlled, standardized environment for our project.
- In Chapter 13, we introduce the Rocker Project which provides Docker Containers for the R Environment.

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.

1.2.4 Info Boxes

Inside the book, there are special Info-Boxes that provide further details.

Tip-Box: My title

Tip-Boxes are used to provide insight into specific topics.

Warning-Box: My title

Warning-Boxes are used to provide important warnings.

Instructions-Box: My title

Instructions-Boxes are used to provide detailed instructions.

Details-Box: My title

Details-Boxes are used to provide further details about advanced topics.

Trick-Box: My title

Trick-Boxes are used to describe special useful tricks.

Command Cheatsheet: My title

Command Cheatsheets are used to summarize commands of a specific software.

Moreover, at the end of each chapter, we list all useful links to external documentation in a dedicated box.

Documentation-Box

Documentation-Boxes are used to collect all useful links to external documentation.

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 colleagues to easily navigate around all the files and reproduce the analysis. Remember, our best colleague 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 4.1.2). 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 4, 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 5.1.2. 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 colleagues to easily navigate the code and understand the purpose of each function.

In Chapter 5, we discuss in detail the functional style approach, considering general good practices to write tidy, documented, and efficient code. In Chapter 9, 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 9, 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 10, 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 11).
  - *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.

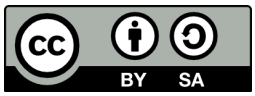
In the case of **software** or **code releasing** the most popular open-source license are:

- **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 4.2.3, 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 8.2.1, 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 operating 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 operating system; see *"Details-Box: The 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: The Garden of Forking Paths

The syntax to define file paths differs according to the operating 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 3.2.2; 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 operating 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 5 and Chapter 9, we describe how to adopt the functional style approach and how to manage the analysis workflow, respectively. In Chapter 10, 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 7 and Chapter 8, 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

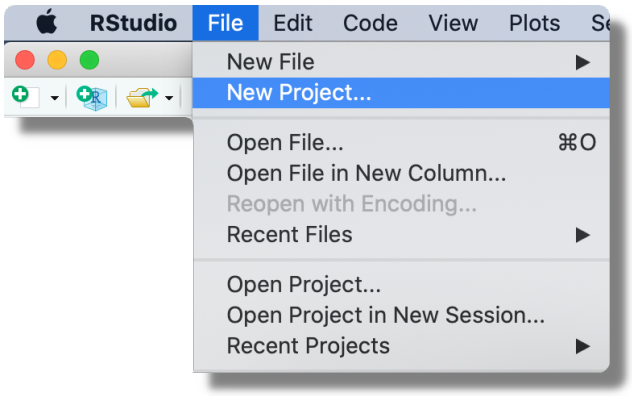
RStudio has built-in support for projects that allows us to create independent RStudio sessions with their own settings, environment, and history. Let's see how to create a project directly from RStudio and discuss some specific features.

3.2.1 Creating a New Project

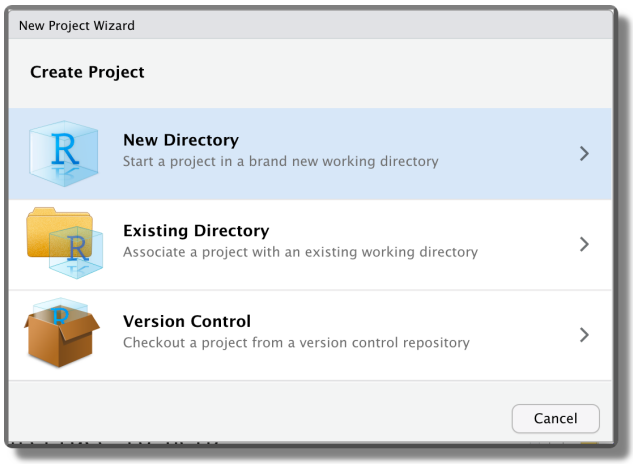
To create a new RStudio project:

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

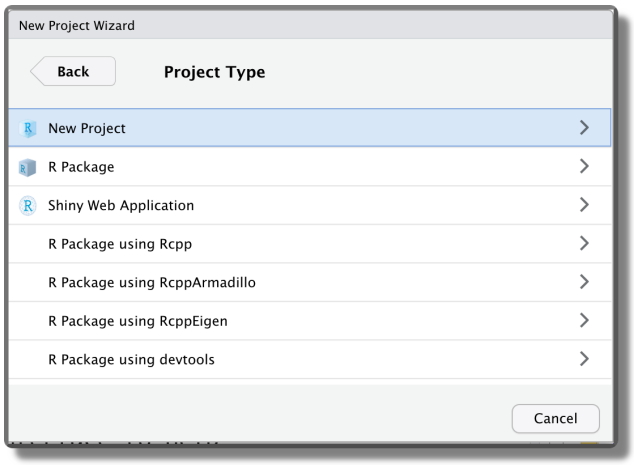
"Open Project..." or *"Recent Projects"*, respectively.



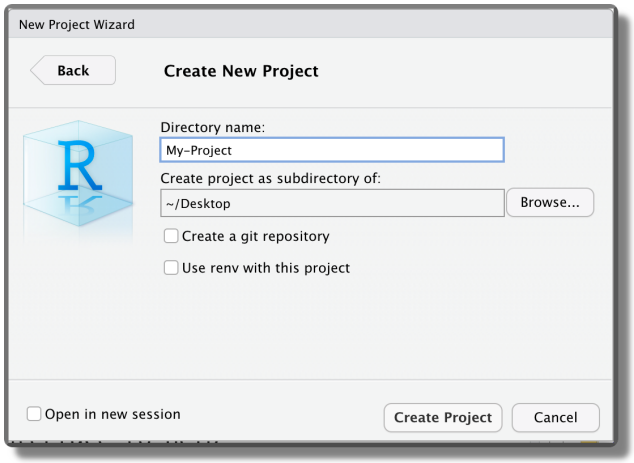
2. When creating a new project, we can decide between starting from a new directory (*"New Directory"*; the most common approach), associating a project with an already existing directory (*"Existing Directory"*), or associating a project to an online repository (*"Version Control"*; for more information see Chapter 7 and Chapter 8).



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"*.

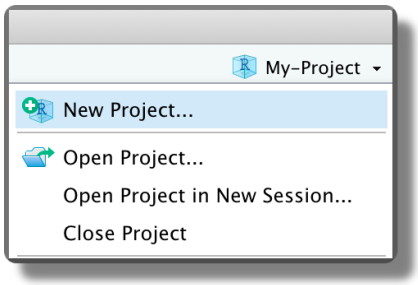


4. Finally, we can indicate the location where to create the project directory and specify the sdirectory name. This will be used also as the project name. Note that two more options are available *"Create a git repository"* and *"Use renv with this project"*. We discuss these options in Chapter 7.6 and Chapter 11.3, 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 

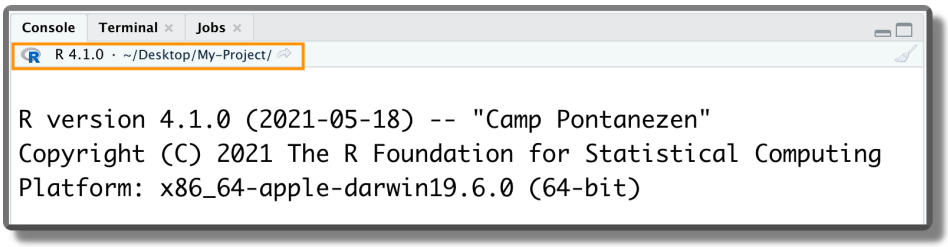
. The current project is also indicated in the top right corner. Click on it to see other project options and to close the project (*"Close Project"*; or from the top bar menu *"File > Close Project"*).



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()`.



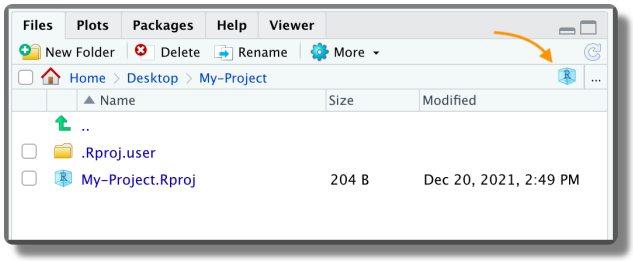
  As discussed in Section 3.1.3.1, this is a very useful feature because now we no longer have to bother about setting the working directory manually (we can finally forget about the `setwd()` command). Moreover, we can refer to any file using relative paths considering as reference the project root.

Details-Box: File Paths in R

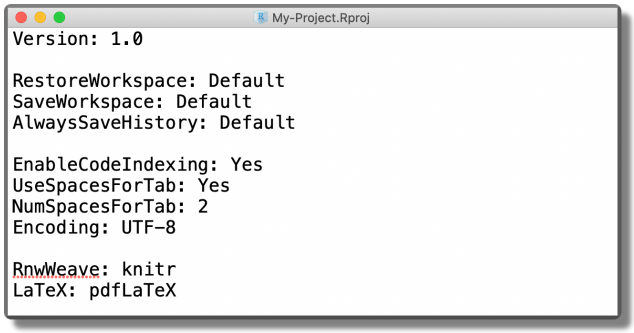
In R, we can specify file path using forward slash `"/"` or backslash `"\"` independently of the operating system we are currently using. However, the backslash has a special meaning in R as it is used as *escape character*. Thus, to specify a file path using backslash we have to double them (e.g., "my-project\\data\\my-data.csv").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

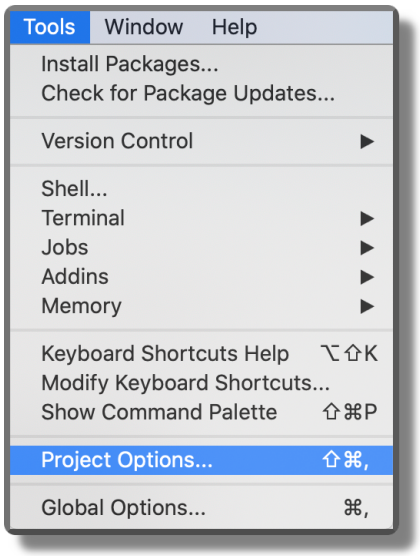
- **`<project-name>.Rproj` File.** The default Project template creates an empty directory with a single file named `<project-name>.Rproj` (plus some hidden files). Clicking on the `<project-name>.Rproj` file from the file manager (not from RStudio), we can open the selected project in a new RStudio 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.



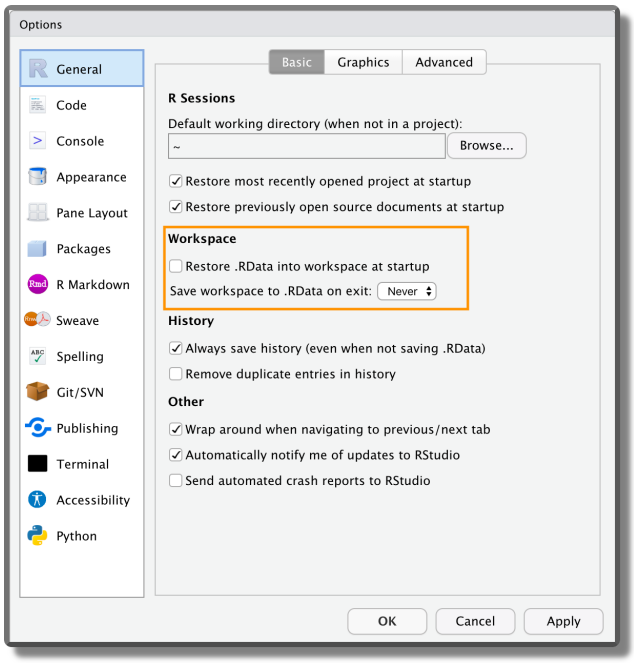
The `<project-name>.Rproj` file is simply a text file with the project settings. Using a text editor, we can see the actual content.



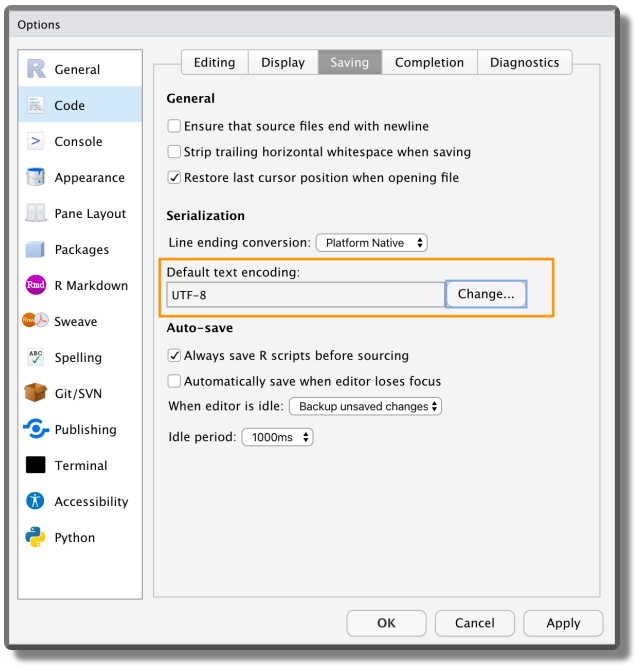
- **Project Settings.** We can specify specific settings for each project. From the top bar menu, select *"Tools > Project Options"* and specify the required options according to our needs.



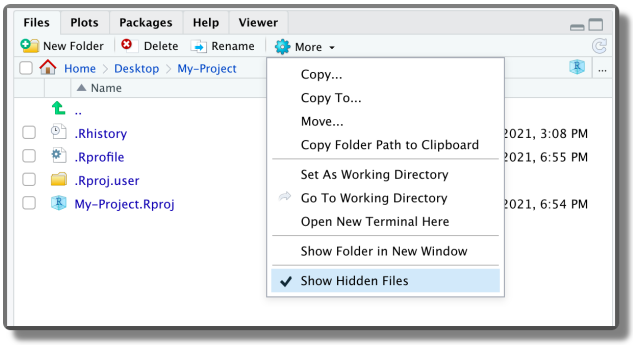
Some recommended settings should be applied as default to all projects. To do that, select from the top bar menu *"Tools > Global Options"*. From the panel *"General"* in the *"Workspace"* section, ensure the box *"Restore..."* is **not** selected and the option *"Save..."* is set to **Never** (see figure 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.



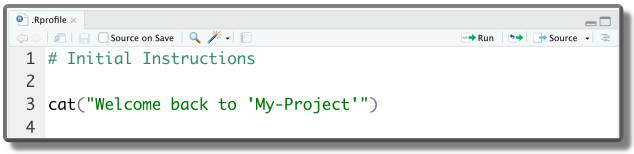
Another important setting is the encoding. From the panel *"Code"* ensure that the default text encoding is set to *"UTF-8"*. Encoding defines how the characters are represented by the computer. This could be problematic for different alphabets or special characters (e.g., accented characters). The UTF-8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



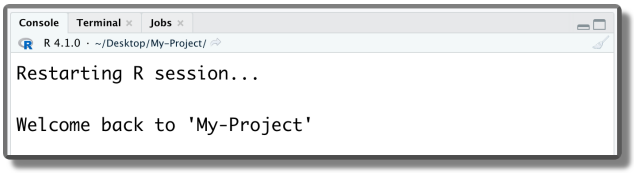
- **`.Rprofile` File.** This file is a special script that is automatically executed at the beginning of each session (or when the session is restarted). This file is not available by default but we can create it by naming an R script as `.Rprofile` (**without extension!**) [TODO: check in windows https://stackoverflow.com/questions/28664852/saving-a-file-as-rprofile-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 5.2.2 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 7.6.

- **`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 7.6, 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.

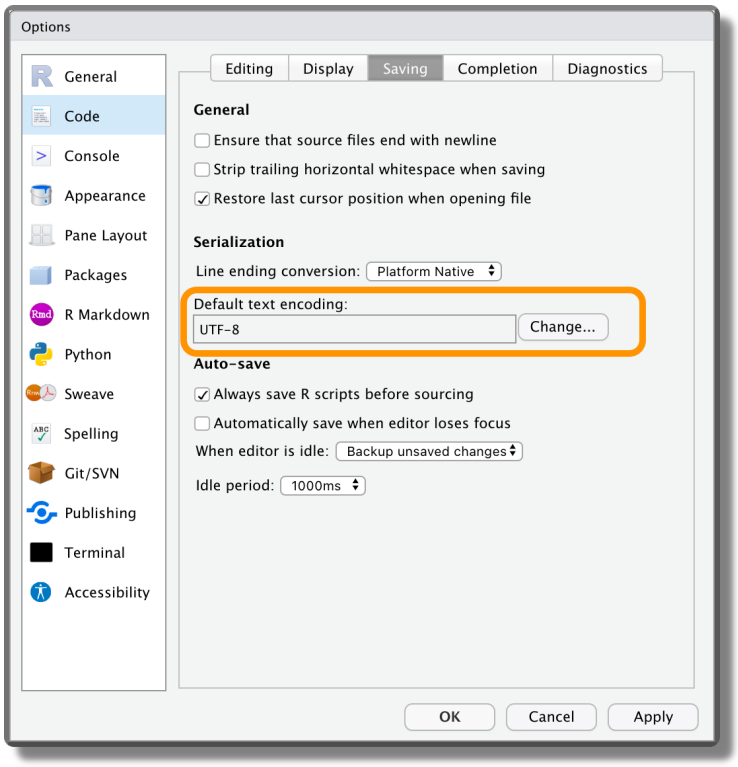
- **Machine Readable.** Data should be stored in a format that allows other researchers to easily use them. People saving tabular data in a PDF file would deserve to be punished in hell. Do they expect us to copy and paste all values by hand?

- **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 9, 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 5.1.3.3). 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

collection (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.

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

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

```
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.

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

```
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 9.

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 9, 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.

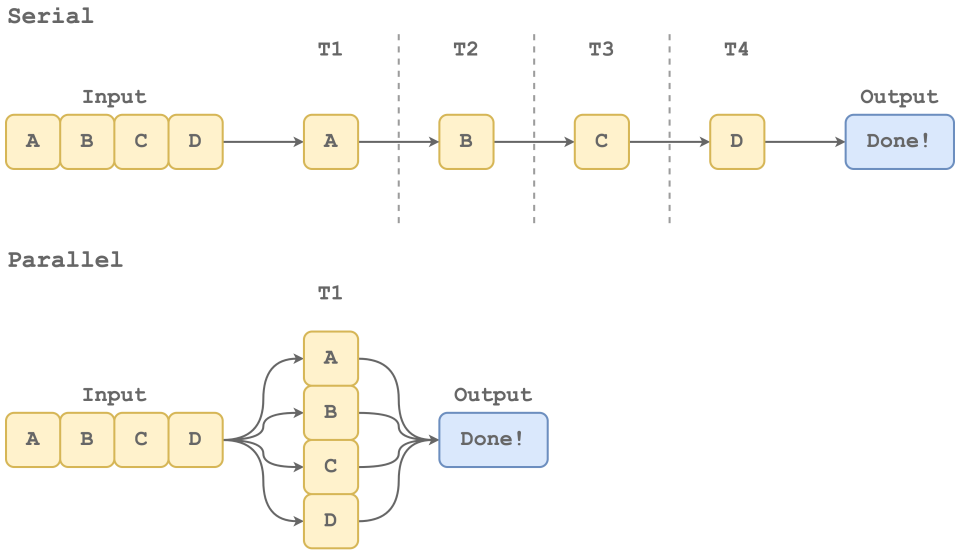
  In *Compiled Languages* (e.g., C or C++), the source code is translated using a compiler before we can execute it. The compilation process is slow and it is required each time we make changes to the source code. Once compiled, however, the code can be simply loaded and executed in a very fast and efficient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

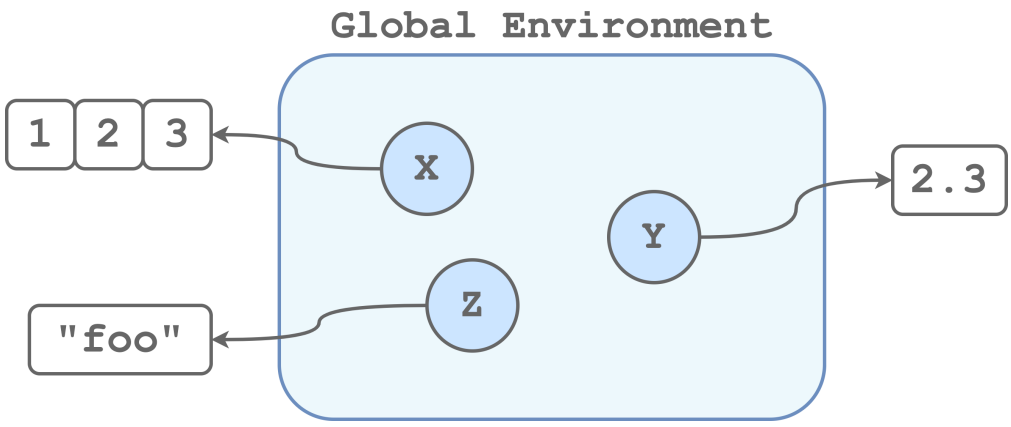
So, parallelization is a wonderful world that can help us to reach incredible levels of performance but we need to use it with care.

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!

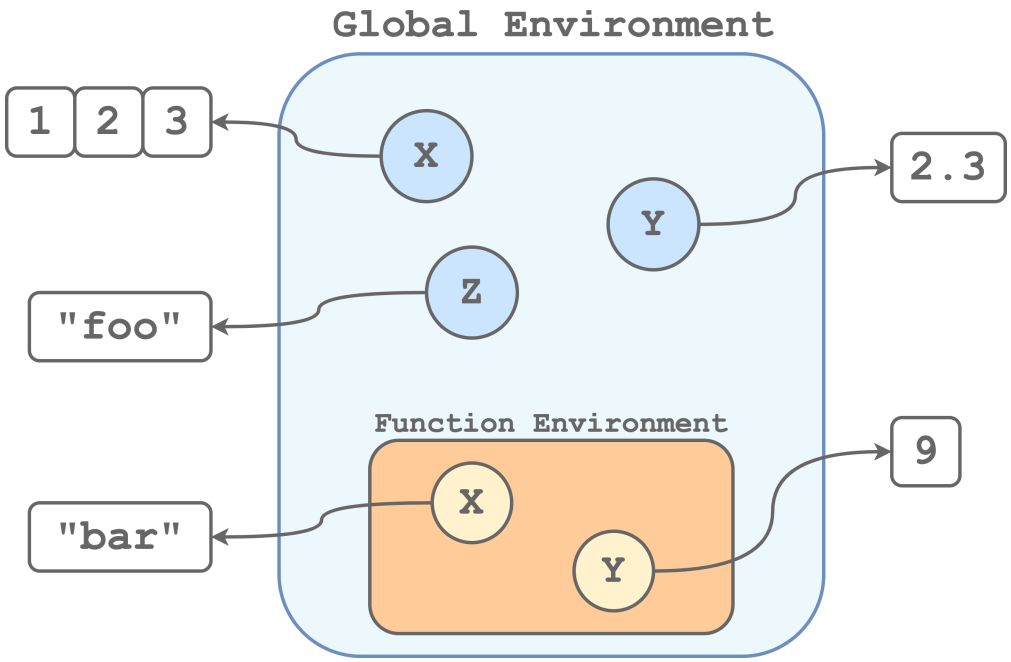
5.1.3.2 Environments

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.

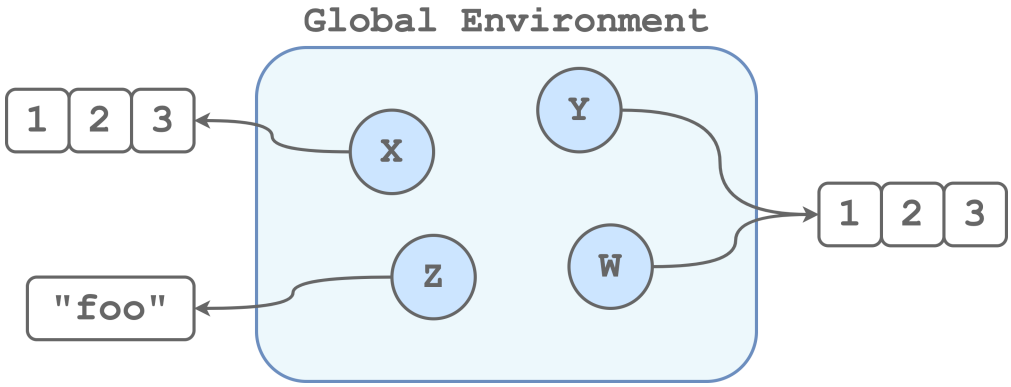
Note that we can not modify global variables from within a function as any attempt will simply create a local variable.

```
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
```

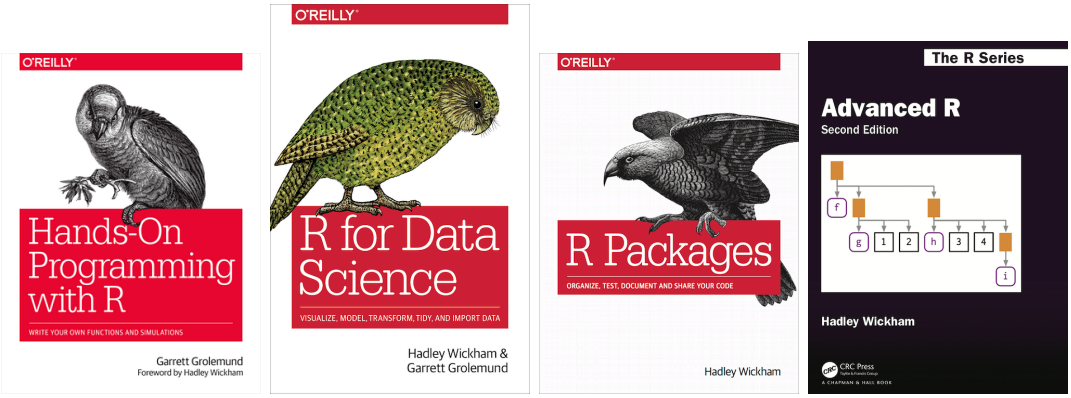
Note that in most object-oriented programming languages, methods are accessed using the dot character (`"."`). For this reason, we should always separate words in object names using `snake_case` and not `snake.case`.

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 11.3). 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.

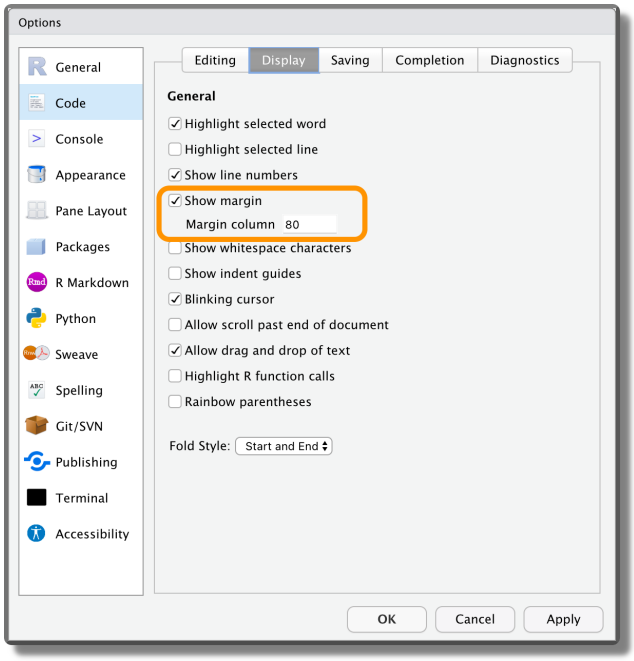
For more details about R coding style, consider the two following resources:

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

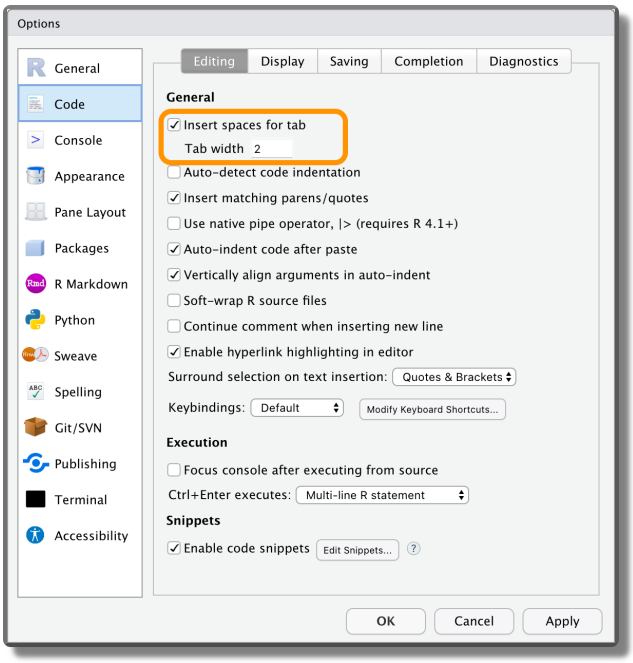
Here we review only a few aspects:

- **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

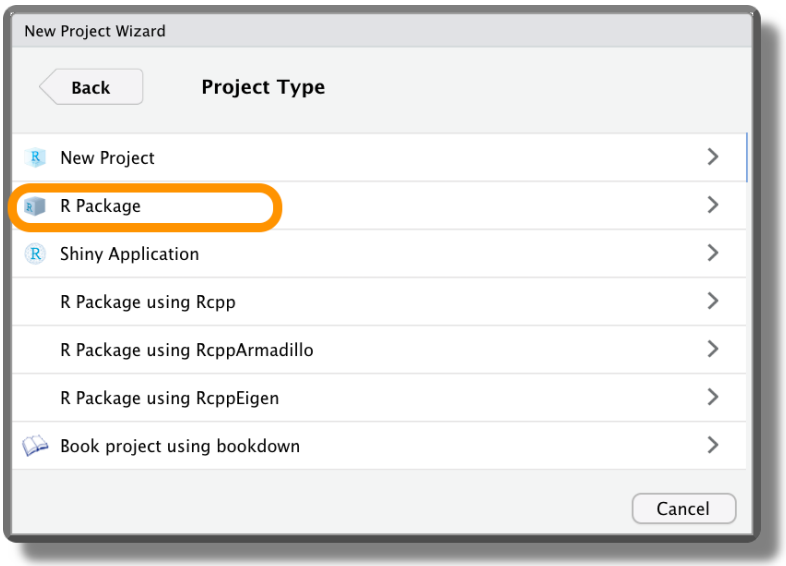
Adopting a functional style approach, we will create lots of functions and use them in the analyses. To organize our project files, we can save all the scripts containing the function definitions in a separate directory (e.g., `R/`) and source all them at the beginning of the main script used to run 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 (must be capital `"R"`).
- **`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 (Wickham, Hester, et al., 2021) 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 (Wickham, Danenberg, et al., 2021) 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.

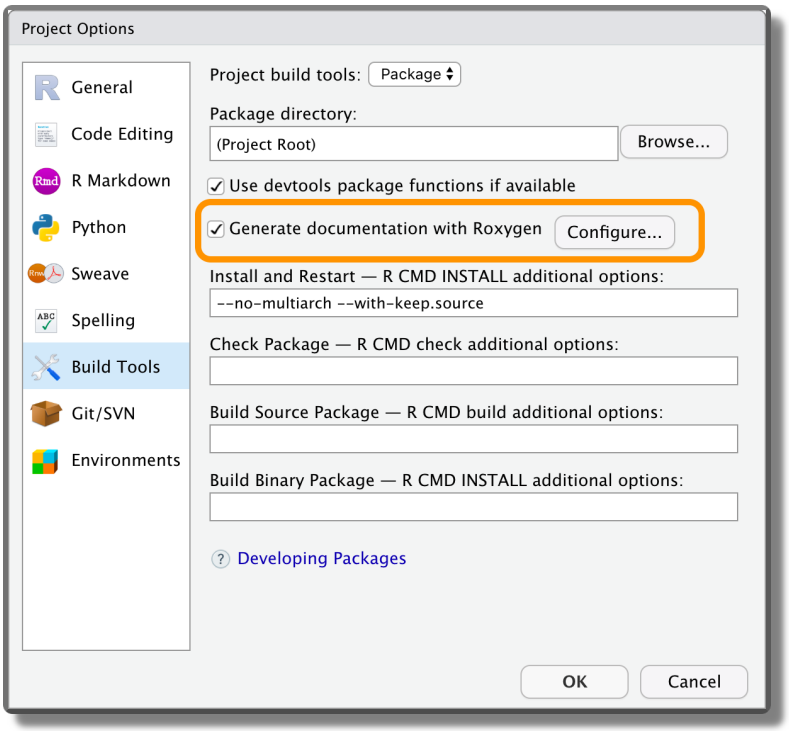
`roxygen2` assumes a specific structure and uses specific tags to correctly produce the different parts of the documentation. Below is a simple example of documenting a function using `roxygen2`. Note the use of `#'` instead of `#` to create the comments and the special tags (`@<tag-name>`) used to specify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

```r
#' @param digits Integer indicating the rounding number of decimal places
#'               (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 (Wickham, 2022) 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

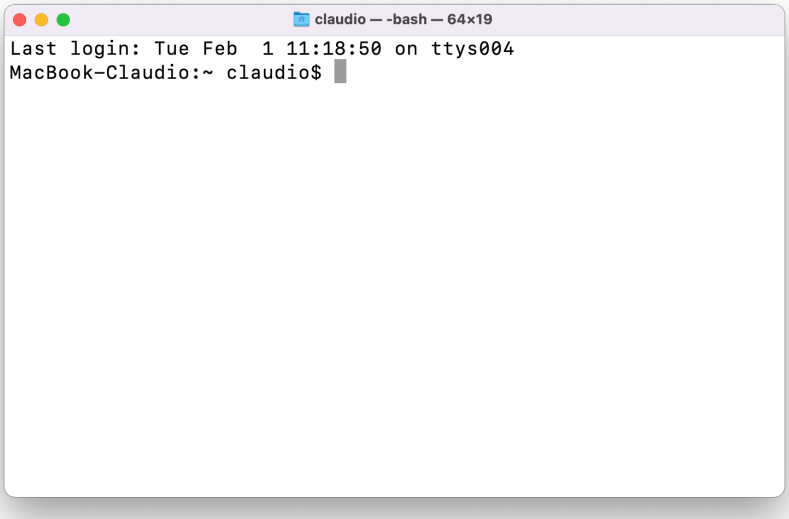
`testthat`

- `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 7) and Docker (see Chapter 12).

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 operating 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 Operating 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."

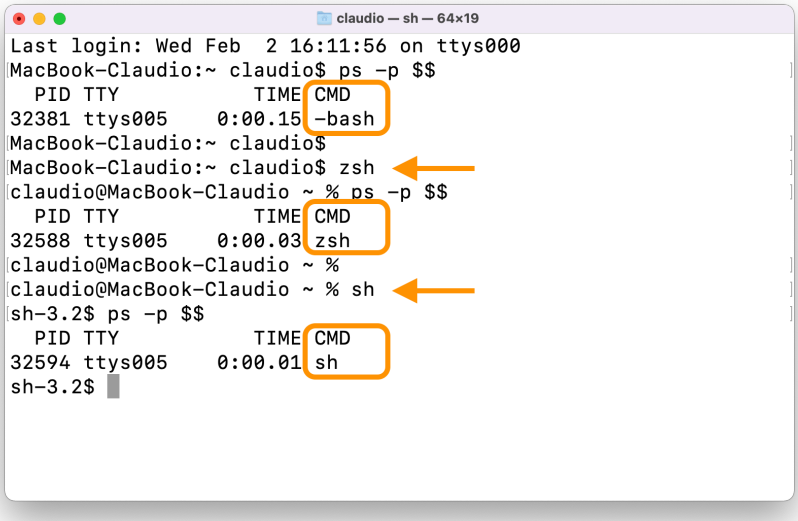
6.1.1 Different Shells

Different shell command line interpreters are available depending on the Operating 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 Operating 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 Operating 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 Operating 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 Operating 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 Operating 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.

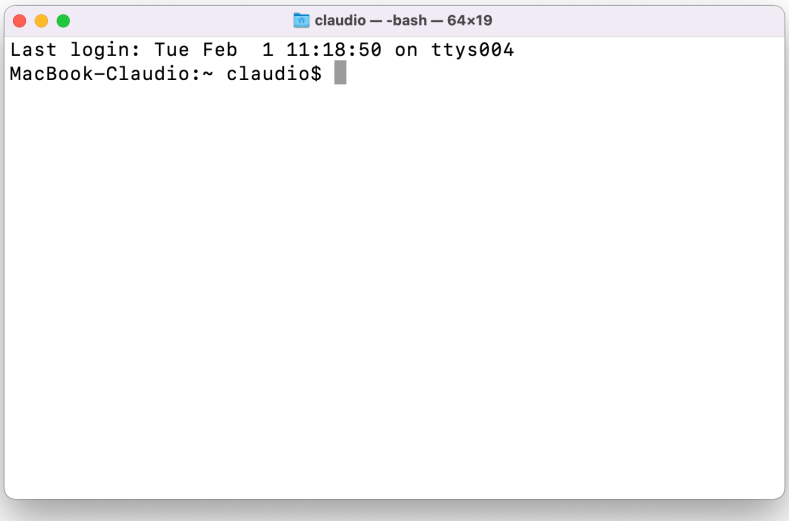
6.3 Get Started

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 12), `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 Directories

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 12.2.3.3), 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 operating 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 hierar-

chical structure. Moreover, we also need to understand other aspects such as file metadata, ownership, and user permissions (see Chapter 12.2.3.3). 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 machine directory structure. 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)

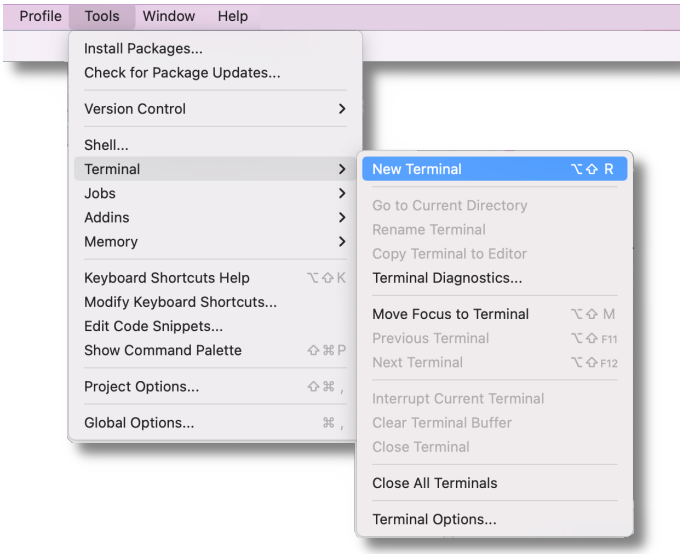
Just in case you went down the rabbit hole:

- 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`.

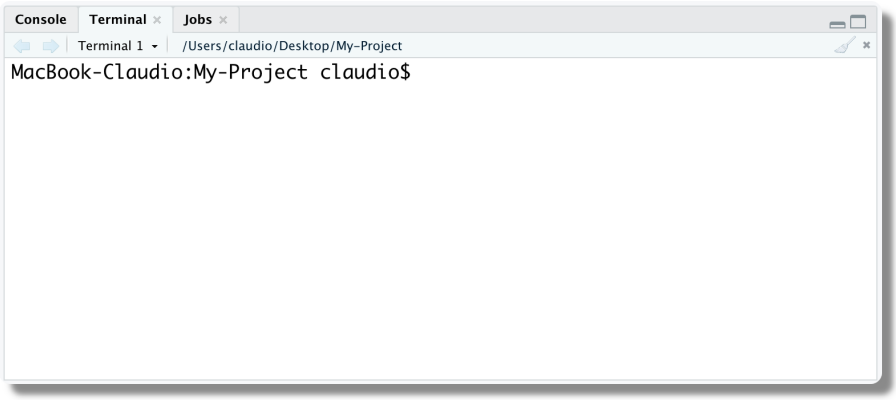
6.4 Terminal in RStudio

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.

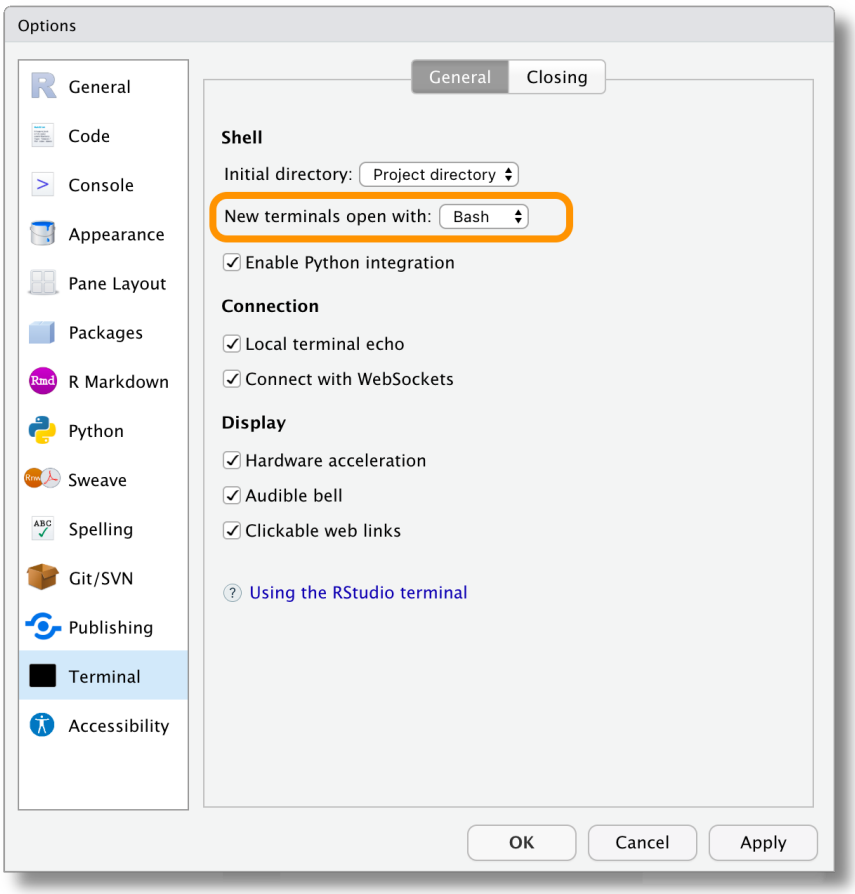
In RStudio, we can open a Terminal window selecting from *Tools* > *Terminal* > *New Terminal*.



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.

7 Git

In the previous chapters, we learned to organize all our files and data in a well structured and documented repository. Moreover we learned how to write readable and maintainable code.

However, as the number of files and lines of code increase, it is more and more difficult to maintain full control of our project. We need some tools to help us managing this process. In this chapter, we introduce Git, a software for tracking changes in any file during the development of our project. In Chapter 8, instead, we introduce GitHub, a dedicated Git hosting services that allow us to collaborate with other colleagues using shared remote repositories.

7.1 Version Control

As our projects get more complex, we quickly end up writing thousands of lines of code. During the development, we create several scripts and define many functions; we continuously revise the code, adding new parts, removing others, and making any sort of changes. In particular, when collaborating with other colleagues, this whole process can quickly became very chaotic. We will likely end up with a messy project with multiple copies of the same files (e.g, `my-script-2`, `my-script-5-rev`, `my-script-10b-rev-adjust` and other crazy names). In this scenario, any file conflict or error in the code would results in endless hours trying to figure out the problem and restoring the project to the previous working state could become a real adventure. We need something that helps us managing and tracking our projects during their development.

Fortunately, we do not need to invent anything. Programmers have already faced these issues for years and their answer is *"version control"*. Version control software allow

us to track the files in our project saving all changes and managing collaboration with other colleagues. In particular, version control have the following advantages:

- **Tracking Files.** All changes to the project files (creation, deletion or modification) are recorded keeping track of which were the specific changes, who made them, when, and why. In this way, we obtain a detailed history and complete traceability of the project development. Moreover, we can easily check earlier version of the project and if something goes wrong we can restore the project to any previous status (see Section 7.2.1).
- **Managing Collaboration.** By sharing an online repository, multiple colleagues can contribute to the development of the same project concurrently. They can work independently on their local machine and then add their own contributions to the shared repository. Version control software manage this whole process in a smooth and efficient way, ensuring no conflicts between the different contributions. A conflict occurs when two colleagues modify the same line in a file (see Section 7.2.2).
- **Branching.** During the development, we can create different branches (like the branches of a tree) that diverge from the main line of development. Each branch is independent allowing us to make changes without affecting the other branches or the main line of development. When ready, we can integrate all changes into the main line by merging the desired branch. Branches are commonly used to develop new features, solve issues, or make experiments safely without worrying about braking the code. Only when a only when a stable solution is obtained, we integrate branches into the main line of development (see Section 7.2.3).

Version control software are the solution for managing projects development in an efficient and secure way and *Git* (https://git-scm.com/) is the most popular version control software. Therefore, let's learn how to use Git.

7.2 Git Overview

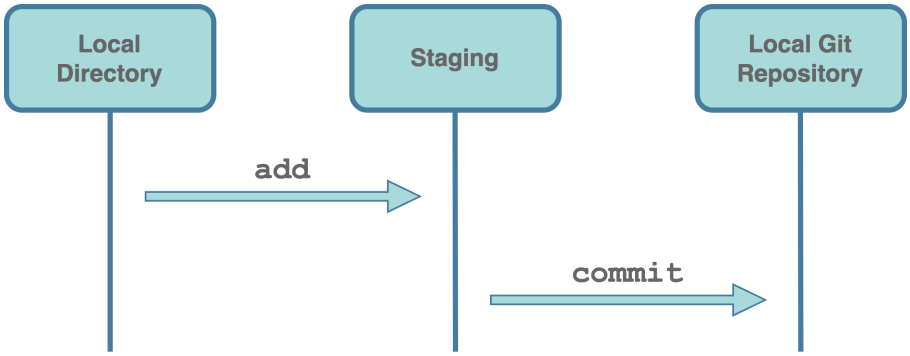
First, we describe the main idea behind the git workflow introducing all the elements, concepts, and technical terms.

7.2.1 Tracking Files

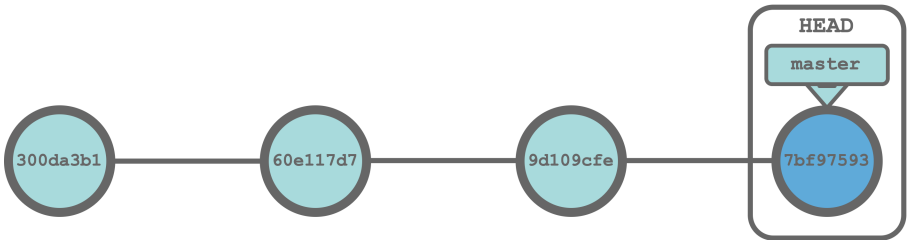
Using Git, we can track the files in our project saving all changes. In Git terms, we define this process as *creating a commit* (or *committing*). Note that creating a commit is much more than simply *saving* in the traditional terms (i.e., overwriting existing files or creating new files). Creating a commit, we keep record of the exact changes (creation, deletion or modification) to each file. We can think of a commit as a snapshot of all the files in our project at that exact point in time.

Creating a commit requires a two step process:

1. **`add`.** First, we need to add the desired files to the *staging area*. We can think of the staging area as a waiting room for the files ready to be committed. This is used to indicate to Git which files from our local directory we want to include in the next commit. In this way, we can group together in the same commit only related files, logically splitting changes into different commits.
2. **`commit`.** Next, we proceed creating the actual commit. Git captures a snapshot of the currently staged files and adds the commit to the timeline in our *local Git repository*. The local Git repository is simply an hidden directory (i.e., `.git/`) within our project used by Git to store all information and files required for version control.

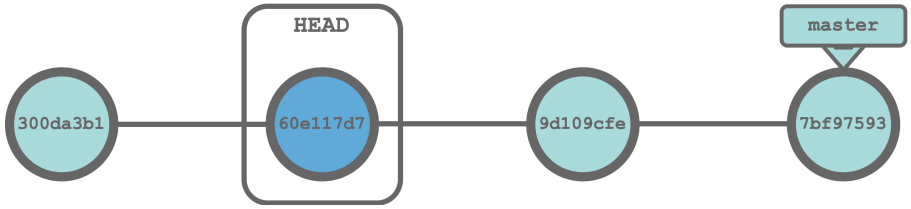


Commit by commit, we create a timeline with the whole history of our project. We can visualize this process as in the following figure, where each circle represents a commit and we called the principal timeline `master` (more on this in Section 7.2.3).



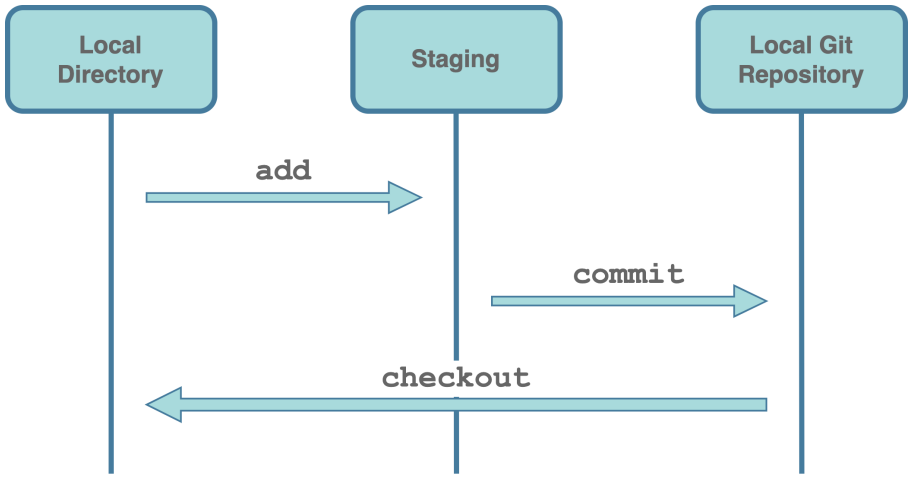
`HEAD` indicates the current commit we are viewing, in our case, the last commit in the timeline. Moreover, each commit is identified by a unique SHA-1 hash ID (i.e., an alpha-numeric sequence). We can use these IDs to move along the project timeline as if we had a time machine.

Imagine we have found some unexpected issues in our code and we want to move back to a previous commit to check if at that point in time everything was fine. We can move our `HEAD` to the desired commit and, as if by magic, all files in our project will be restored to their previous versions at that exact point in time. Of course, we can subsequently move again the `HEAD` to the last commit restoring the files to their present version.



In Git terms, we define the process of moving the `HEAD` to restore a specific commit status as:

- **`checkout`.** We can restore any commit status in our timeline. Git uses the information in the local repository to restore all files in our local directory to their versions at that exact point in time.



7.2.2 Managing Collaboration

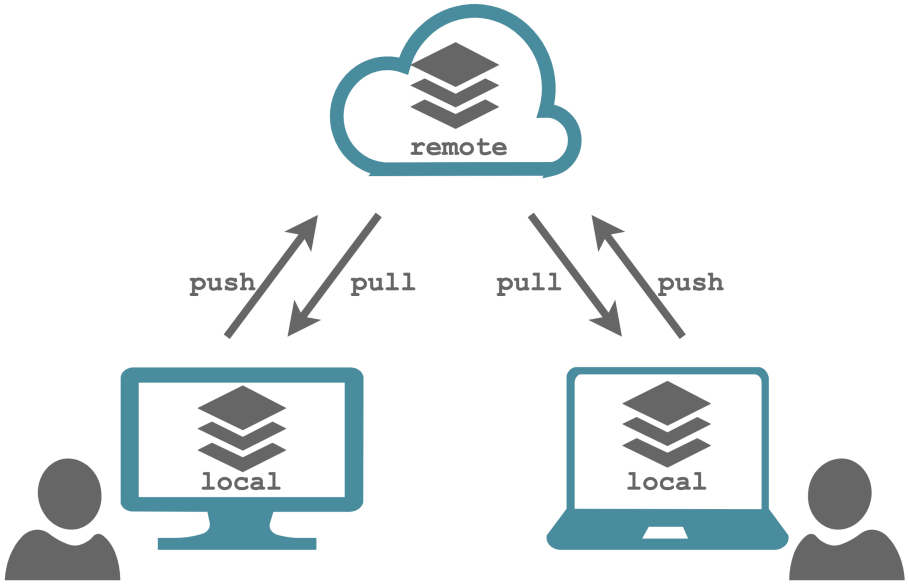
So far we described the workflow of a single user working only on his/her local machine. Even if we work on our own only on our local machine, using Git is always highly recommended as allows us to keep track of our project development. However, when we collaborate with other colleagues, Git show all of its power.

To collaborate with other colleagues, we need to share the Git repository. Therefore, we will have:

- **Remote Git Repository.** The Git repository shared between all colleagues that is used as reference. This is usually available on a online service (e.g., GitHub, GitLab, or Bitbucket; see Chapter 8) or a private server. Note that, for Git to work, we need to share the repositories on dedicated Git hosting services. We can not use common cloud storage services (e.g., Google Drive or Drop Box) as they do not have Git running behind.
- **Local Git Repository.** Each colleagues has his/her own copy of the remote Git repository on his/her own local machine.

Colleagues can work independently on their own local repository as usual, making all sort of changes and create multiple new commits. Once finished, they can share their work with other colleagues by uploading their new commits to the remote repository. Next, other colleagues can updated their local repository by downloading the new changes from the remote repository. In particular, we have two actions:

- **`pull`.** Download the new commits in the local repository from the remote repository.
- **`push`.** Upload the new commits from the local repository to the remote repository.

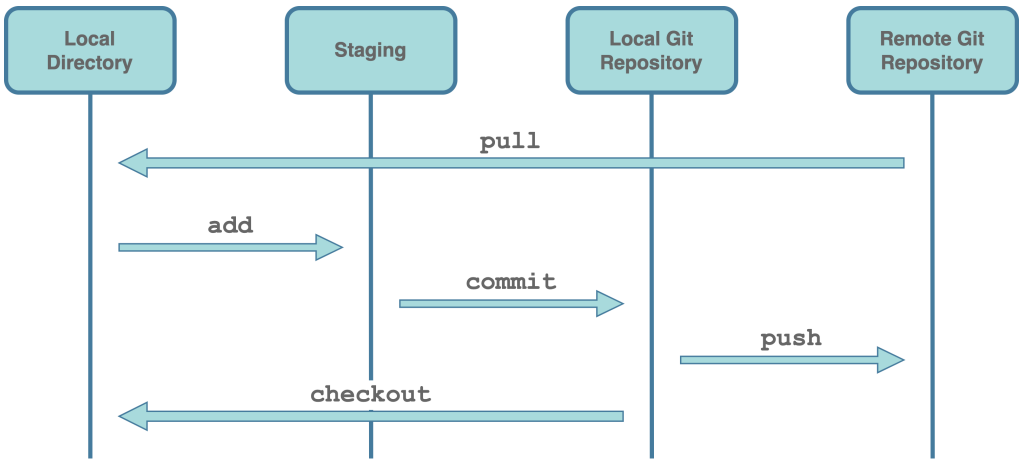


Git automatically manages this whole process in a smooth and efficient way. In particular, Git checks for:

- **Pull First.** Git do not allows us to push our commits if detects in the remote repository new commits that are not available in our local repository. This is usually due to other colleagues pushing their commits. Git force us to integrate these commits by pulling from the remote repository and next push our commits. In this way, Git ensure that there are no conflicts between our commits and the new commits from other colleagues.
- **Solve Conflicts.** A conflict con occur when we integrate changes from other colleagues (or merging branches; see Section 7.2.3) and two different commits modify the same line in a file (or one commit deletes the file and the other edit the same file). In this case, Git is not able to automatically integrate the changes. Therefore, before pushing our commits, Git ask us to explicitly solve any conflict by choosing which of the two versions to keep.

Summarizing, when collaborating with other colleagues on a shared repository, the workflow proceed as follow. First, we pull latest commits from the remote repository to

our local repository. Next, we keep working as usual developing the project and creating new commits in the local repository. Finally, when we reach some stable point in the development, we push our new commits to the remote repository.



Remember,

> ***Always pull first!***

This must be the first thing we do at each session as it is important to always work with the latest project version. If we forget about it, we may end up relying on old code versions introducing several issues.

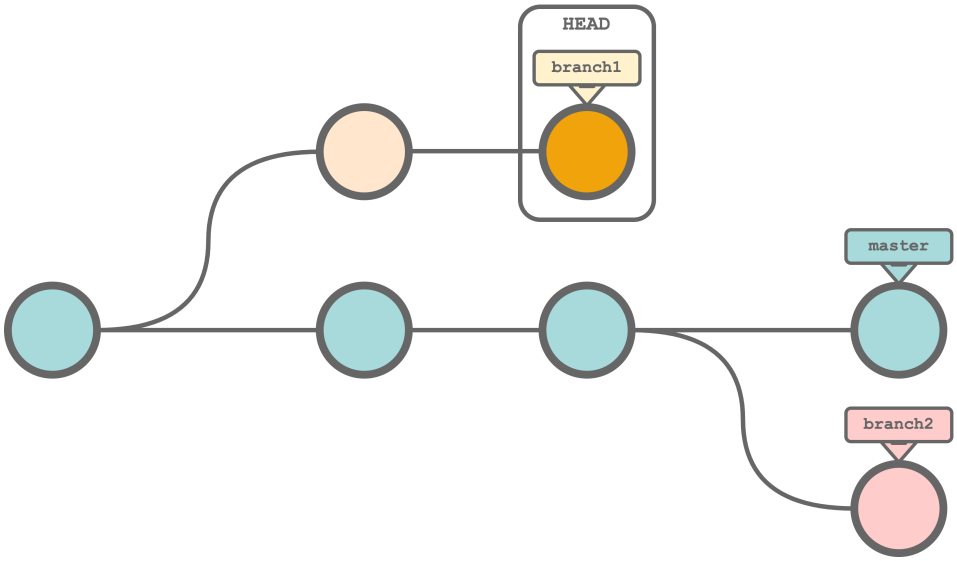
7.2.3 Branching and Merging

Imagine we are working with other colleagues on the development of different features. Ideally, we would be able to develop each feature independently of each other and integrate everything together when ready. In this way, we could avoid continuously running into conflicts or other issues. Alternatively, suppose we got a wonderful idea to improve our project but this would require some drastic changes and we are not sure if this idea would be successful. Ideally, we would like to work on a separate copy of our project allowing us to shift back to the original project if something goes irremediably wrong.

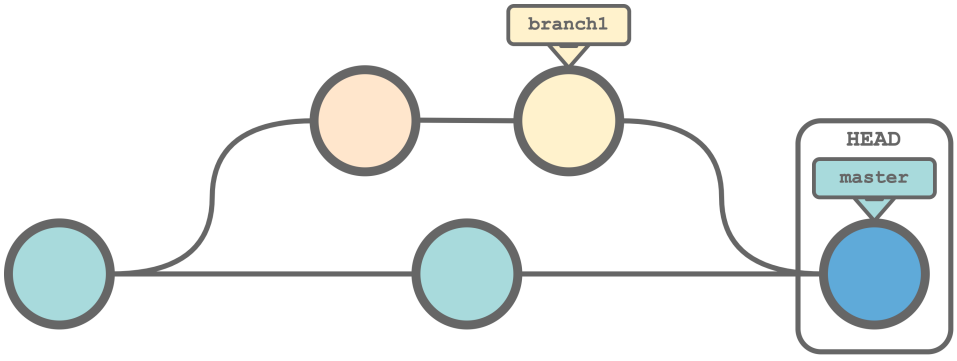
Git has two incredibly useful features designed to facilitate the workflow in both scenarios described above:

- **`branch`.** We can create independent streams of development. In Git terms, these are called *branches* (like the branches of a tree). Branches allows us to work concurrently on different features (or projects ideas) without letting changes in one branch to affect the other branches. We need to assign a name to each branch and a standard convention is to name the principal branch `master` (or `main`; these are just names without any special meaning per se). We can subsequently move our `HEAD` to shift

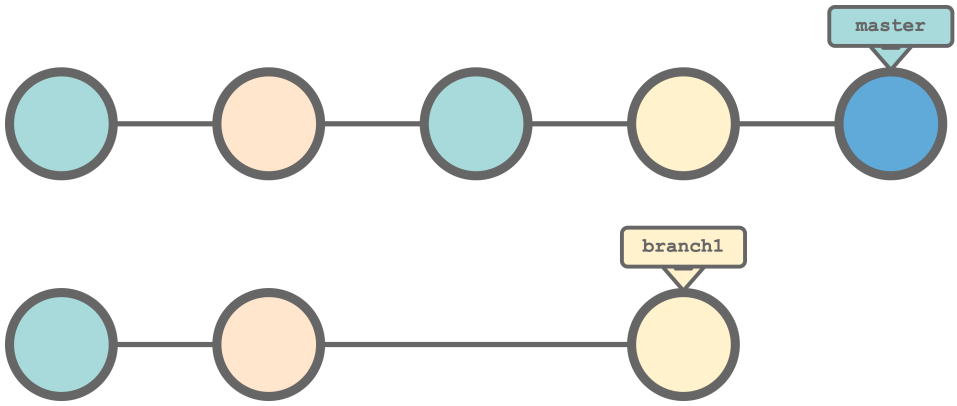
from one branch to another. Files in our local directory will be restored according to the specific branch stream of changes. In the figure below, we can see an example of a Git history with three branches where the `HEAD` is currently pointing to `branch1`.



- **`merge`.** At some point, we may want to integrate changes from different branches. In Git terms, we *merge* one branch into another. In this way, we can develop different features (or projects ideas) independently and merge them when ready. Before merging, Git will check for conflicts between the different branch commits. In case of conflicts, Git will require us to explicitly solve them by choosing which of the two versions to keep. In the figure below, we can see an example of a Git history where the `branch1` is merged back into `master`.



Note that the directionality is important. Branches are not mixed together creating a single branch, but commits in the timeline of one branch are integrated in the timeline of the other branch. Therefore, considering the previous example, we would obtain the following timelines for the two branches.



Finally, even after merging, the two branches still exist and they are independent. This means that we can keep working on `branch1` and new commits will not affect `master`. Only the merging operation allows to integrate commits from one branch to another.

Branching and merging may require some practice and experience before to properly use them. However, they are extremely useful tools to organize and manage projects development workflow both when collaboration with other colleagues or when working on our own. In Section 7.5, we describe suggested practices and more advanced workflows to efficiently organize and use multiple branches during the project development.

7.3 Install Git

We can check whether Git is installed on our machine by running the following command in the terminal.

```
$ git --version
```

The Git version is returned. If Git is not available, we can install it following the Operating System specific instructions. See also https://git-scm.com/downloads.

7.3.1 On Windows

We have two main scenarios:

- **Git Bash.** *Git for Windows* (https://gitforwindows.org/) is a project that provides us the required tools to work with Git on Windows. In particular, it includes *Git Bash* (a Bash terminal to run Git from the command line), *Git GUI* (a graphical user interface for Git), and other integration tools. Install Git for Windows following the instructions at https://git-scm.com/download/win.

- **WLS.** We can use the Windows Subsystem for Linux (see Section 6.2.1) opening the dedicated terminal. Git is already installed in most Linux distributions. if this is not the case (or if we want to update the Git version), we can run the command (for Ubuntu/Debian)

```
$ sudo apt-get update
$ sudo apt-get install git
```

Depending on our preferred solution, we need to use the correct terminal (Git Bash or WLS) to run Git command lines.

7.3.2 On macOS

We have two main scenarios:

- **XCode Command Line Tools.** Git is automatically installed with the XCode Command Line Tools (`xcode-select -install`). If not already available, when running `git --version` command, the system will automatically ask us if we want to install it. However, this is not the latest version of Git.
- **Brew.** If we are using Brew to manage packages (highly recommended), we can install the latest Git version running the command `bash $ brew install git`

7.3.3 On Linux

Git is already installed in most Linux distributions. If this is not the case (or if we want to update the Git version), we can run the command (for Ubuntu/Debian).

```
$ sudo apt-get update
$ sudo apt-get install git
```

7.4 Get Started

Yep! Finally, we are ready to put our hands on Git and enjoy all of it power (and madness). In this introduction to Git, we describe the basic features and commands. For more details and advanced features, we suggest the the following resources:

- *Bitbucket Git Tutorial:* https://www.atlassian.com/git/tutorials
- *Pro Git Book:* https://git-scm.com/book/en/v2
- *Git Official Manual:* https://git-scm.com/doc

Now we can open our terminal and start to play.

7.4.1 Configure Settings

First of all, we need to configure Git providing our username and email. These will be used by Git to identify the author of each commit. To configure the username and email, run the following commands,

```
$ git config --global user.name "<My Name>"
$ git config --global user.email "<user@email.com>"
```

Using the flag `--global`, we configure username and email for all projects for the current user. Configuration settings are stored in `~/.gitconfig`.

```
#----    .gitconfig    ----#
[user]
    name = My Name
    email = user@email.com
```

To check the current settings, run he following command,

```
$ git config --list
```

Perfect! Now Git knows who we are.

Trick-Box: Three Levels of Configuration

We can configure Git settings at three levels:

- **System.** These settings are available for all users and are stored in `/etc/gitconfig`. To set system settings, use the flag `--system`, for example

  ```
  $ git config --system user.name "My Name"
  ```

- **Global.** These settings are available for all projects of the current user and are stored in `~/.gitconfig`. To set global settings, use the flag `--global`, for example

  ```
  $ git config --global user.name "My Name"
  ```

- **Project.** These settings are available only for the current project and are stored in `.git/config` (within the project directory). We do not need flags to set project settings.

```
$ git config user.name "My Name"
```

Of course, project settings over-rides global settings and global settings over-rides system settings (see https://stackoverflow.com/questions/8801729/is-it-possible-to-have-different-git-configuration-for-different-projects/54125961).

7.4.2 Initialize a Git Repository

When starting a project with Git, there are two main scenarios:

- **`init`.** We have an already existing project on our local machine and we want to start using Git for that project. To do that, we run the command `git init` from the project directory.

  ```
  $ cd <path-to/project-directory>
  $ git init
  ```

  This command will create an hidden folder `.git/` within the project directory where Git stores all the information and files required for version control. Moreover, Git also initialize the principal branch using the name `master` (or `main`).

- **`clone`.** The project is already under version control and is available in a remote repository. To obtain a local copy of the repository, we run the command `git clone` indicating the repository URL.

  ```
  $ git clone <repository-URL>
  ```

  This command creates a copy of the remote repository at the current working directory. Note that we can easily clone any public repositories. However, we need specific authorization or authentication protocol (e.g., login credentials or SSH keys) to clone private repositories. We discuss authentication procedures in Chapter 8.1.2.

7.4.3 Tracking Files

Now our project is under version control. Suppose we have the following project structure,

```
my-project/
    |-- README
    |-- data/
    |     |-- raw-data.csv
```

```
    |-- documents/
    |-- code/
    |     |-- my-functions-1
    |     |-- my-functions-2
```

We can run the command `git status` the check the project state.

```
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    code/
    data/
    README

nothing added to commit but untracked files present (use "git add" to track)
```

We are on the principal branch `master` and at the moment all files are untracked. Note that `documents/` is not listed as Git does not track empty directories.

> Tip-Box: .git-track File
>
> As just pointed out, Git do not track empty directories. However, if we need to track an empty directory, we can add an hidden file. For example, we can create and hidden empty file named `.git-track` using the command
>
> ```
> $ touch <path-to-empty-dir>/.git-track
> ```
>
> Note that the name has no special meaning per se. We can use any name.

7.4.3.1 Adding Files to Staging Area

As suggested by Git, we can add untracked files using the command `git add` listing the desire files.

```
$ git add README <other-files>
```

We can also use the flag `--all` to add automatically all untracked files.

```
$ git add --all
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   code/my-functions-1
    new file:   code/my-functions-2
    new file:   data/raw-data.csv
    new file:   README
```

Note that using `git add --all` is not recommended as we may end up adding a lot of not useful files (e.g., system files or build artefacts). If we added some unwanted files, we can use the command `git rm --cached <file>` to remove a specific file from the staging area (or `git reset` to remove all files). We should always aim to keep our repository as clean as possible tracking only relevant files. To help us with this, we can specify which files to ignore listing them in the `.gitignore` file.

Details-Box: .gitignore File

The `.gitignore` file is used to list all files we want to ignore. These files will be not tracked by Git. The `.gitignore` file is usually saved in the project root.

We can specifay the exact file name or we can use matching patterns (i.e., globbing patterns, see https://linux.die.net/man/7/glob) to exclude specific subset of files. For example:

- **`note.txt`**: ignore all files named `"note.txt"` in any directory (e.g., `"my-folder/note.txt"`)
- **`folder-A/note.txt`**: ignore files named `"note.txt"` in `"folder-A"`. In this case the file `"folder-B/note.txt"` is not ignored.
- **`folder-A/`**: ignore files in any directory named `"folder-A"`. For example also files in `"folder-B/folder-A/"` are ignored.
- **`*.log`**: ignore files that ends with `".log"` in any directory.
- **`folder-A/*.log`**: ignore files that ends with `".log"` in `"folder-A"`.
- **`!important.log`**: the exclamation mark is used to create exceptions to previously define patterns. In this case the file `"important.log"` will not be ignored.

When creating or editing the `.gitignore` file, it is important to remember that:

- Each file or pattern has to be defined on a new line
- The character `#` is used to create comments
- Defined patterns are evaluated according to the `.gitignore` file position.

Moreover, the `.gitignore` file is usually saved in the project root but we can also define multiple `.gitignore` files and save them in different locations. Remember patterns are evaluated according to the `.gitignore` file position.

Finally, note that if we want to ignore a file that has been already tracked, first we need to remove it form the list of tracked files by Git. To do that, we can use the command `git rm --cached <file>`. Note that it is important to specify the `--cached` flag otherwise the file will be removed also from the working directory. Next, we can include it into the `.gitignore` file and commit the changes. Now the file will be ignored by Git.

To learn more about the `.gitignore` file and all the other exclusion rules, see https://www.atlassian.com/git/tutorials/saving-changes/gitignore

7.4.3.2 Commiting Changes

Now all files are in the staging area and we are ready for our first commit. To create a commit, run the command

```
$ git commit
```

This will open our preferred text editor asking us for a commit message. Each commit requires a message describing the content of the commit. In this case, we can add the following message

```
Init Repo

- add data and scripts
- add README
```

After we save the file and close the editor, the commit is created. Now we can continue creating new files or modifying existing ones. Once we are ready, we can stage (i.e., `add`) the desired files and create a new commit.

For example, let's modify the `README` file adding some information and save the changes. Git will detect the file has been modified.

```
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
```

```
  (use "git restore <file>..." to discard changes in working directory)
    modified:   README

no changes added to commit (use "git add" and/or "git commit -a")
```

We can stage and commit the new changes

```
$ git commit -am "Update README"
```

Note we staged the changes and passed the commit message directly in the command, using two common commit options:

- **`git commit -m "<commit message>".`** Specify the commit message in the inline command. x
- **`git commit -a.`** Commit all changes to already tracked files (no new files) without adding first.

These options are useful in the case of small commits, but we recommend limited use as it is always preferable to selectively stage files and add descriptive commit messages.

Tip-Box: Good Commit Messages

There are no strict rules when creating a commit message. A good practice, however, is to summarize the content of the commit on the first line with a brief title (less than 50 characters). Next, after a blank line, we can list a detailed description of all changes.

A good message should not only list what we changed but also why we did it. Descriptive messages are very useful for collaborators when reviewing the code and remember, this is likely to be the future us.

7.4.3.3 Checking Commit History

To get the history of all commits, we can use the command `git log`. Commits are listed from latest to oldest and for each commit we get the following information: commit ID, author, date, and commit message. For a more compact version we can specify the flag `--oneline`.

```
$ git log
commit ec15cb085c2e5aa463778bbef3239b06b9ac4010 (HEAD -> master)
Author: username <user@email.com>
Date:   Tue Feb 15 10:06:27 2022 +0100
```

```
    Update README

commit 93398d18e686d4698af3e52a0677585afe467a19
Author: username <user@email.com>
Date:   Tue Feb 15 09:26:13 2022 +0100

    Init Repo

    - add data and scripts
    - add README
```

We can check changes between commits using the function `git diff`. Depending on the specified arguments we can compare different commits. In particular,

- **`git diff`.** It displays changes since the last commit.
- **`git diff <commit-ID>`.** It displays changes since a specified commit.
- **`git diff <commit-ID-A> <commit-ID-B>`.** It displays changes from commit-A to commit-B.

Note that it is not necessary to provide the whole commit ID, but we can provide only the initial characters as long as they uniquely identify the commit. Usually 6 to 10 digits are enough. For example in our case we can compare the two commit running,

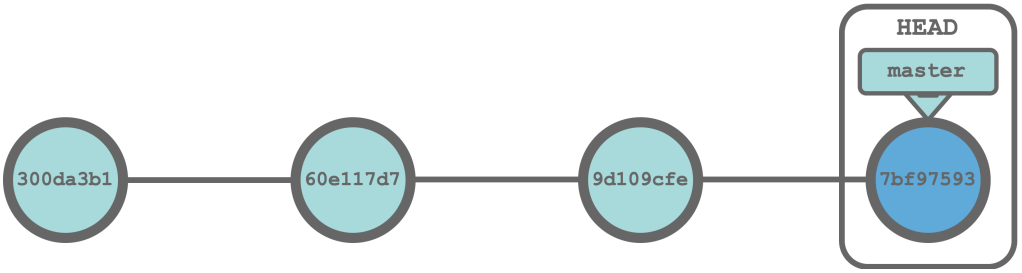
```
$ git diff 93398d1 ec15cb0
diff --git a/README b/README
index 64772c2..e994379 100644
--- a/README
+++ b/README
@@ -1 +1,2 @@
 This is my first project using Git!
+New information
```

Checking commit history and changes between commits directly form command line may be complicated. Popular IDEs (e.g., RStudio or Visual Studio Code) provide useful graphical interfaces to facilitate these tasks (see Section 7.6).

7.4.4 Undoing Commits

Git allow us to keep track of the whole history of our project. In this way we are sure we will never lose anything and, if something goes wrong during the development, we can always easily move back to previous commits.

For example, consider a project with the following Git history.



Imagine we ended up in some big troubles and we need to go back two commits (commit `60e117d7`) and restart the development over again. Using Git, there are three different ways of *undoing* commits:

- **`git checkout <commit-ID>.`**
- **`git revert <commit-ID>.`**
- **`git reset <commit-ID>.`**

Let's describe the difference between these three commands.

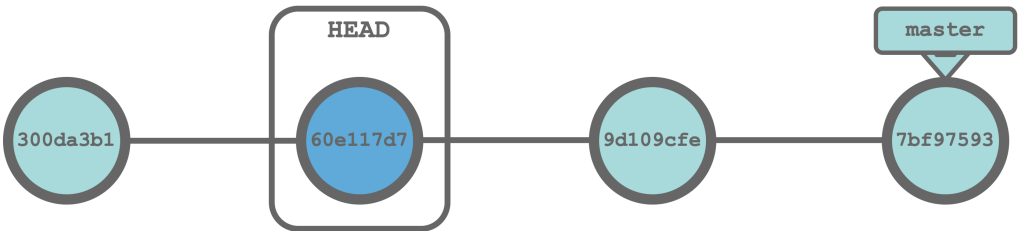
7.4.4.1 Checkout

As already introduced in Section 7.2.1, the `git checkout` command allows us to point our `HEAD` (i.e., the current commit we are viewing) to a target commit. In this way, all files in our project will be restored to their state at that exact commit.

```
$ git checkout <commit-ID>`
```

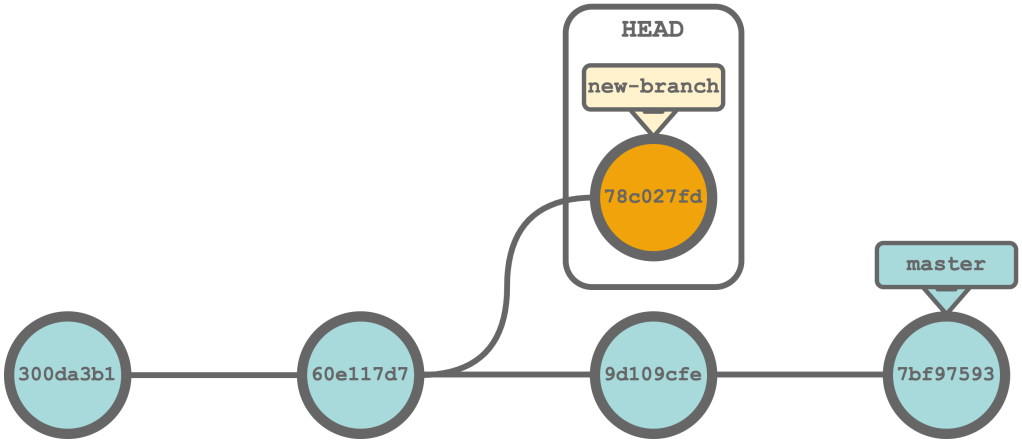
In our example, we can restore the desired commit (`60e117d7`) running the command,

```
$ git checkout 60e117d7
```



In this state, we can make any experimental changes and commit them. These new commits, however, will not affect our principal line of development (`master`) or other branches. In Git terms we are in `detached HEAD` state. This means that once we move back to our last commit (using the command `git checkout master`) all new commits will be lost.

If we want to retain the new commits, we need to create a new branch, using the command `git checkout -b <branch-name>` (see Section 7.4.6). In this way, however, our development would diverge from the `master` branch and this may not be desired.



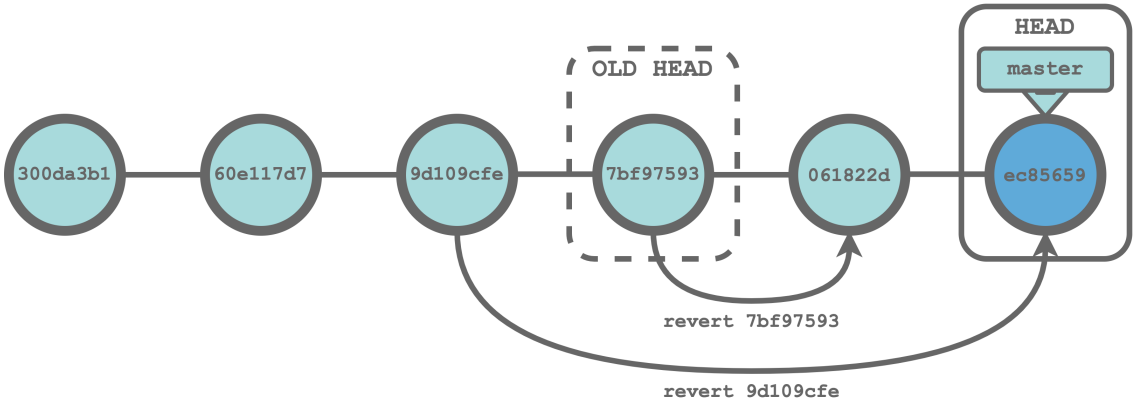
7.4.4.2 Revert

The `git revert` command allow us to *undo* commits. However, rather than removing the target commit from the Git history, `git revert` creates a new commit in which the changes of the target commit are reverted. Following this approach, the whole Git history is maintained without losing any commit. This allow us to safely collaborate with other colleagues on shared repositories.

```
$ git revert <commit-ID>`
```

In our example, we can restore the desired commit (`60e117d7`) reverting the two subsequent commits (`7bf97593` and `9d109cfe`). To do that, we run the commands,

```
$ git revert 7bf97593
$ git revert 9d109cfe
```



Note that by default each `git revert` command creates a new commit. Moreover, by using `git revert` we can undo any commit at any arbitrary point in the Git history.

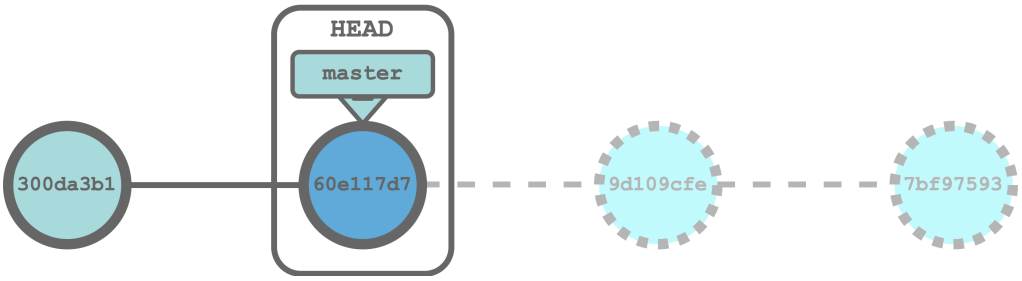
7.4.4.3 Reset

The `git reset` command allow us to restore a target commit. All the subsequent commits are removed from the Git history irremediably modifying the Git history. These changes may lead to severe issues if we are collaborating on a shared repository. Therefore, `git reset` should never be used for undoing commits on public repositories but only for undoing local changes.

```
$ git reset <commit-ID>
```

In our example, we can reset our Git history to the desired commit (`60e117d7`) running the command,

```
$ git reset 60e117d7
```



Note that by default, the `git reset` command reset th commit history and file staging state to the target commit. However, all files in our working directory still maintain all the changes from the last commit (previous to the reset command). This is our last occasion to revise all changes before losing everything. We can run the command `git restore <file>` to definitely discard changes. Alternatively, we can specify the flag `--hard` in the `git reset` command, in this case we will not be able to revise changes and all data would be irremediably lost.

Tip-Box: Undoing Commits

Summarizing, when we need to undo some commits:

- **`git checkout`** command is useful to restore the project to a previous state from where we can our test without worrying abut impacting the current Git history and we can create a new line of development . However, the `git checkout` command does not allow us to undo changes in the original line of development.
- **`git revert`** command is the recommended safe method for undoing changes as it never deletes old commits but create new commits for undoing changes.

Using the `git revert` command we always maintain the full Git history intact.

- `git reset` command is the dangerous way of undoing changes as old commits are irremediably deleted from the Git history. The `git reset` command should be only used for undoing local changes and should never used for undoing public commits.

7.4.5 Managing Collaboration

When collaborating with other colleagues on a project, we share a **remote repository** used as main reference. Each colleague can work independently on its own **local repository** and upload/download commits when required.

Let's describe the operations required to collaborate.

7.4.5.1 Adding a Remote Repository

If we created our repository using the `git clone` command, it automatically creates a connection to the remote repository. Otherwise, we need to add the name and the URL of a remote repository. To do that, we use the command,

```
$ git remote add <name> <url>
```

For example,

```
$ git remote add origin https://github.com/username/my-project.git
```

Note that the name `origin` is the standard convention to indicate the principal remote repository but it has no special meaning per se. Moreover, there are multiple ways of specifying the remote repository URL according to the authentication protocol used (e.g., HTTP or SSH protocol). This will affect our specific permissions (e.g., read and/or write access). Authentication and permissions are discussed in Chapter 8.1.2.

We can add more than one remote repository. This may be useful, for example, to manage large projects were different team collaborate on different project features. In this case, we would add both the principal remote repository (shared by all teams) and our team specific remote repository. To check all the remote repositories currently available, we can us the command,

```
$ git remote -v
origin  https://github.com/username/my-project.git (fetch)
origin  https://github.com/username/my-project.git (push)
```

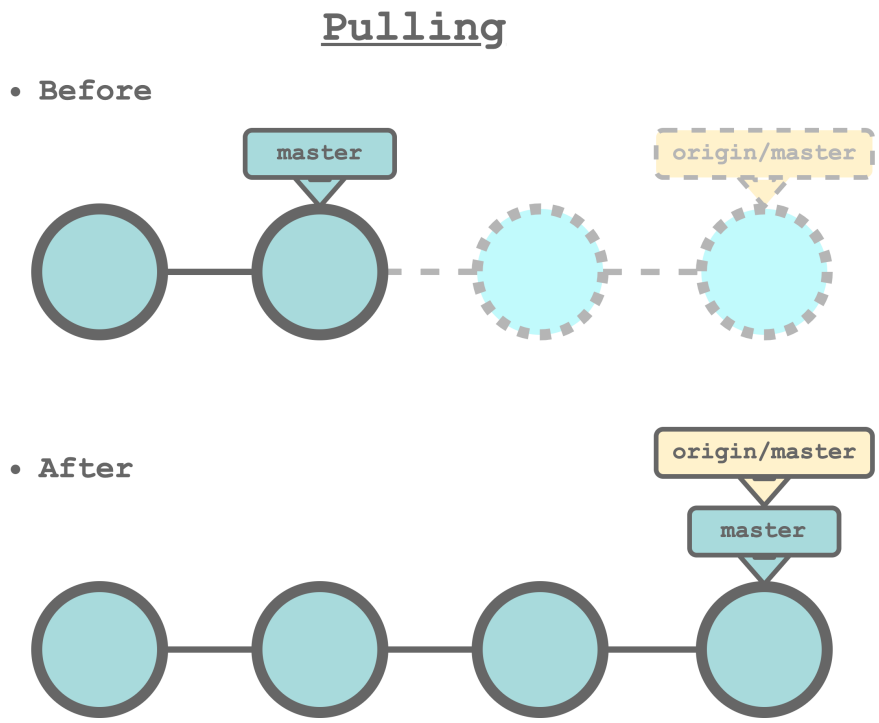
7.4.5.2 Pulling Commits

To download new commits from a remote repository to the current branch, we use the command,

```
$ git pull <remote-name> <branch-name>
```

specifying the remote repository name and the branch we want to pull from. For example, to update our current branch according to `origin/master`, we run the command,

```
$ git pull origin master
```



The `git pull` command downloads the new commits and automatically updates our working directory to match the last changes. Alternatively, we can use the safer command `git fetch` that downloads the new commits but does not update our working directory allowing us to review commits before integrating them in our local repository (see https://www.atlassian.com/git/tutorials/syncing/git-fetch).

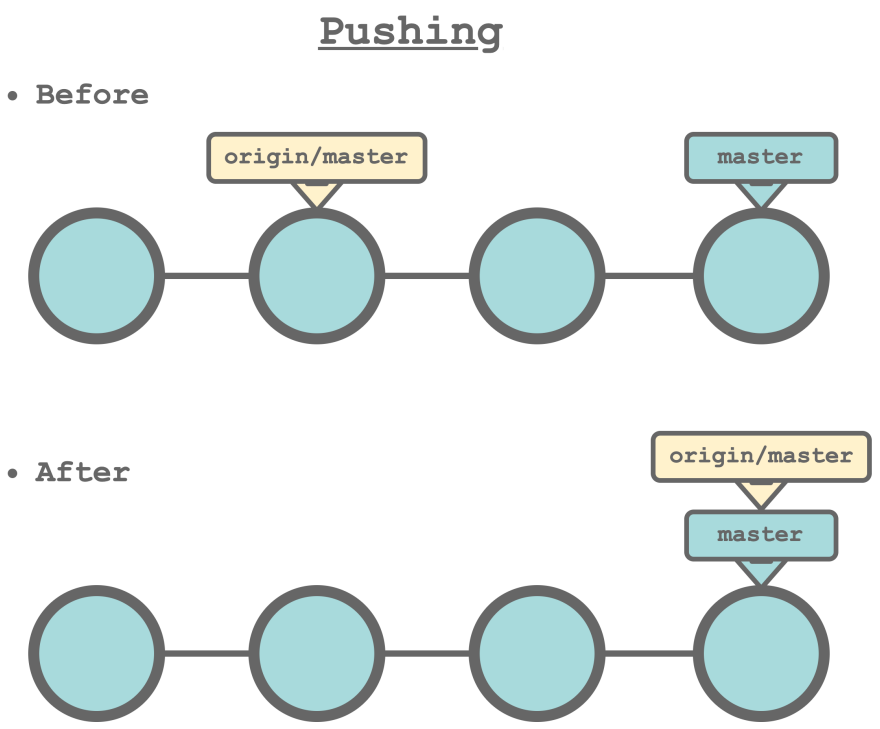
7.4.5.3 Pushing Commits

To upload new commits from a local branch to a remote repository, we use the command,

```
# Local branch and remote branch same name
$ git push <remote-name> <branch-name>

# Local branch and remote branch different names
$ git push <remote-name> <local-branch>:<remote-branch>
```

specifying the remote repository name and the branch name. For example, to upload the new commits from the local `master` branch to `origin/master`, we run the command,

```
$ git push origin master
```



Warning-Box: Always Pull First!

Git does not allows us to push our commits if our local repository is not updated with the last changes in the remote repository. If we try to push, a similar message is displayed,

```
$ git push
To https://github.com/username/my-project.git
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'https://github.com/username/my-project.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

In this case, we need to pull first and then push again. However, it may occur that our new commits have some conflicts with the last changes in the remote repository. Git would ask us to explicitly solve any conflict before pushing to the remote repository. To minimize this issue, we should always work starting from the latest updates by remembering to pull at the beginning of each session. We discuss how to deal with conflicts in Section 7.4.6.

7.4.5.4 Setting Upstream

Instead of specifying the remote repository name and the branch name each time, we can set the default remote branch for each of our local branch. In Git terms this is called the *upstream* branch. To do that, we can use the command,

```
$ git branch --set-upstream-to="<remote/branch>"
```

specifying the remote repository name and the branch name we want to use as upstream. For example, to set `origin/master` as the upstream branch for our local `master` branch, first we checkout to the desired branch, next we run the command,

```
$ git checkout master
$ git branch --set-upstream-to="origin/master"
Branch 'master' set up to track remote branch 'master' from 'origin'.
```

Now we can pull and push commits from and to the upstream branch simply running `git pull` and `git push` respectively. Alternatively, we can set the upstream branch when pushing commits specifying the flag `-u` (or `--set-upstream`). Considering the example above,

```
$ git push -u origin master
...
Branch 'master' set up to track remote branch 'master' from 'origin'.
```

7.4.6 Branching and Merging

As introduced in Section 7.2.3, with Git we can create branches to allow independent lines of development. The principal line of development is itself a branch and, following standard conventions, it is named `master` (or `main`; these are just names without any special meaning per se).

Let's describe the operations required to create and merge branches.

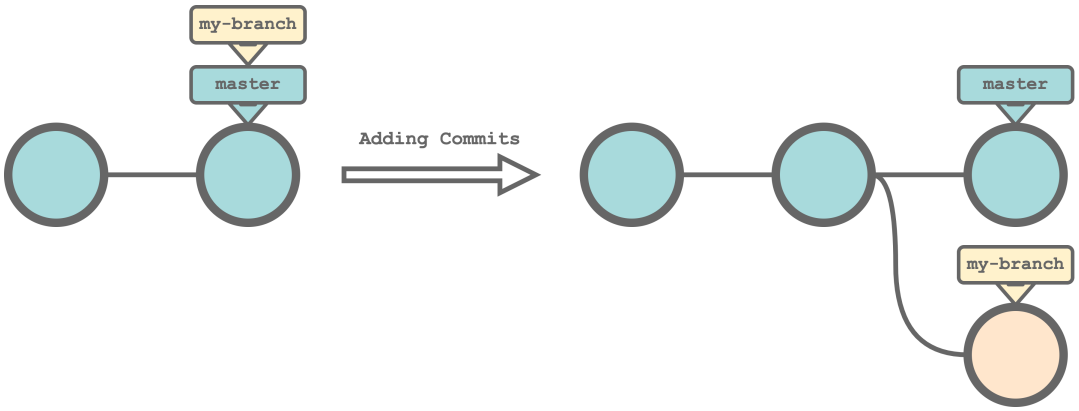
7.4.6.1 Creating Branches

It is important to understand that branches are simply pointers to a specific commit. When we create a branch we are creating a new pointer to the current commit. From here, each branch will follow its own line of development. To create a local branch, we use the command

```
$ git branch <branch-name>
```

For example, we can create a new branch named `my-branch` running the command,

```
$ git branch my-branch
```



A remote branch with the same name is automatically created when we push the branch to the remote repository. To do that, we run the command,

```
$ git push -u <remote-name> <branch-name>
```

Remember that we use the option `-u` to automatically set the upstream branch.

7.4.6.2 Selecting Branches

We can switch from a branch to another using the command,

```
$ git checkout <branch-name>
```

As already described in Section 7.2.1, the `git checkout` command allows us to point our `HEAD` (i.e., the current commit we are viewing) to a target commit. Specifying a branch, we move our `HEAD` to the commit the branch is pointing at. Again, all files in our project will be restored to their state in that exact branch.

Moreover, we can use the `-b` flag to create and move to a new branch.

```
$ git checkout -b <new-branch-name>
```

We can list all currently available branches running the command `git branch`. In particular, we can use the flag `-vv` to obtain a verbose output with more information (i.e., branch name, commit ID, upstream branch, commit message) or the `-r` flag to list all the remote branches.

```
$ git branch
* master
  my-branch

$ git branch -vv
* master    00ab7a8 [origin/master] Update master
  my-branch 702b800 [origin/my-branch] Create my-branch

$ git branch -vvr
origin/master    00ab7a8 Update master
origin/my-branch 702b800 Create my-branch
```

Note that the `*` symbol indicates the branch currently active.

7.4.6.3 Merging Branches

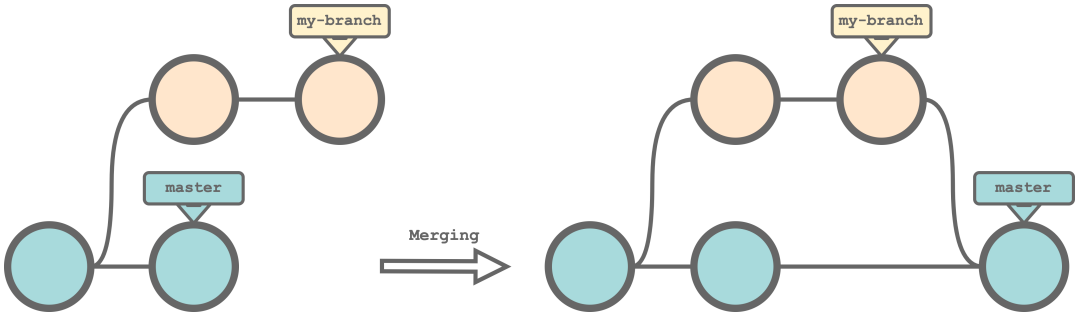
To integrate two line of development together, we merge one branch into another. The directionality is important. Branches are not mixed together creating a single branch, but commits in the timeline of one branch are integrated in the timeline of the other branch.

Before merging we need to ensure that the two branches are up-to-date with the remote repository. Next, we switch to the receiving branch and we merge the other branch using the command,

```
$ git merge <branch-name>
```

For example, we can merge `my-branch` into `master` running the command,

```
$ git checkout master
$ git merge my-branch
```



Note that when merging, we always obtain a *"merge commit"* that joins the two lines of development.

Once merged, we can delete the branch if no longer useful. To delete a branch (local and remote), we use the commands,

```
# Remove local branch
$ git branch -d <branch-name>

# Remove remote branch
$ git push <remote-name> --delete <branch-name>
```

7.4.6.4 Solving Conflicts

When merging two branches or pulling changes from a remote repository, conflicts can arise. Conflicts occurs when two different commits modify the same line in a file (or one commit deletes the file and the other edit the same file). In these cases, Git is no longer able to automatically integrate the changes. Therefore, Git requires us to explicitly solve the conflicts by choosing which of the two versions to keep.

Suppose we create a file `my-file.txt` as follow,

```
#----    my-file.txt    ----#

First text line
Second text line
Third text line
```

Next we create two branches (`branch-A` and `branch-B`) and we modify the second text line differently in the two branches (`"Branch-A text line"` and `"Branch-B text line"` respectively). If we try merging `branch-B` into `branch-A` we would get a conflict.

```
$ git checkout branch-A
$ git merge branch-B
Auto-merging my-file.txt
CONFLICT (content): Merge conflict in my-file.txt
Automatic merge failed; fix conflicts and then commit the result.
```

Examining `my-file.txt`, we see that the conflict is marked as follow

```
#----    my-file.txt    ----#

First text line
<<<<<<< HEAD
Branch-A text line
=======
Branch-B text line
>>>>>>> branch-B
Third text line
```

The `<<<<<<< HEAD` and `>>>>>>> branch-B` lines delimit the conflict and the `=======` line separates the two conflicting versions. In particular, we get the changes of the currently active branch `HEAD` (in our case `branch-A`) followed by the changes of the merging branch (in our case `branch-B`).

To solve a conflict, we need to edit the file modifying the text to keep the desired version and removing the other together with the conflict delimiters. Finally, we commit the changes and Git will create a new *"merge commit"* finalizing the merge.

We can abort a merge that created conflicts using the command,

```
$ git merge --abort
```

Trick-Box: Conflict Style diff3

We can choose between different styles with which Git marks conflicts. We recommend using the `diff3` style. To change the default merge conflict style, run

```
$ git config --global merge.conflictstyle diff3
```

In addition to the changes of the current branch and the changes of the merging branch, the `diff3` style also display the original version of the common ancestor commit. This give us more context, helping us understanding the reasons of the different changes and properly choosing which one to keep.

```
#----    diff3 conflict style    ----#

<<<<<<< HEAD

||||||| common ancestor commit

=======

>>>>>>> merging-branch
```

7.4.7 Summary

Git is very powerful but requires time (and tears) to learn how to use. We described its basic features and commands. Again, for more details and advanced features, we suggest the the following resources:

- *Bitbucket Git Tutorial:* https://www.atlassian.com/git/tutorials
- *Pro Git Book:* https://git-scm.com/book/en/v2
- *Git Official Manual:* https://git-scm.com/doc

For everything else there's *stackoverflow*.

Command Cheatsheet: Git

Here a brief summary of all the Git commands introduced so far.

```
#----    Configure Settings    ----#

git config --list          # Check configuration settings

git config --global user.name "<My Name>"          # Configure username
git config --global user.email "<user@email.com>"  # Configure email
git config --global merge.conflictstyle diff3      # Configure conflict style
```

```
#----    Initialize Git Repo    ----#

git init <path>             # Initialize local repository
git clone <repository-URL>  # Clone remote repository
```

```
#----    Tracking Files    ----#

git status                # Check repository status

# Staging
git add <files>           # Add selected files to staging area
git add --all             # Add all files to staging area

git rm --cached <file>    # Remove selected files from staging area
git reset                 # Remove all files from staging area

# Commit
git commit                # Create commit
    # Options
    -a                # Commit all changes of tracked files
    -m "<message>"    # Inline commit message

git log                   # Get commit history
git log --oneline         # Get commit history (compact)

git diff                                # Changes since the last commit
git diff <commit-ID>                    # Changes since a specified commit
git diff <commit-ID-A> <commit-ID-B>  # Changes from commit-A to commit-B.

git checkout <commit-ID>  # Point HEAD to specific commit/branch
git revert <commit-ID>    # Undo commit (forward-way)
git reset <commit-ID>     # Undo commit (disruptive-way)
```

```
#----    Collaboration    ----#

git remote -v                # List remote (verbose)
git remote add <name> <url>  # Add remote

git pull <remote-name> <branch-name>  # Pull commits from remote
git push <remote-name> <branch-name>  # Push commit to remote
         -u       # Set upstream remote branch


git branch --set-upstream-to="<remote/branch>"  # Set  upstream remote branch
```

```
#----    Branching    ----#

git branch          # List branches
git branch -vv      # List branches (verbose)
git branch -vvr       # List remote branches (verbose)

git branch <branch-name>           # Create new branch
git checkout -b <new-branch-name>  # Create and move to new branch

git merge <branch-name>            # Merge target branch
git merge --abort                  # Abort merge

git branch -d <branch-name>                    # Delete local Branch
git push <remote-name> --delete <branch-name>  # Delete remote Branch
```

7.5 Git Workflow

As discussed in the previous sections, Git is extremely useful (if not fundamental) to manage project development and collaboration. Here we discuss how we can organize the project development using branches following efficient and organized workflows.

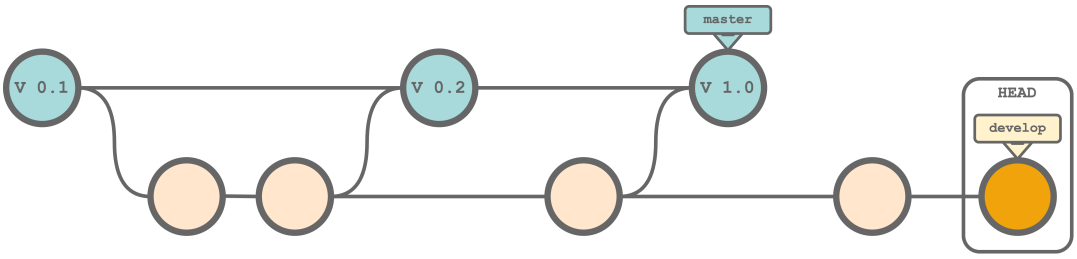
These workflows are most effective for projects creating applications or services used by other people and that requires continuous maintenance and development. However, they me be very useful even for complex research projects.

7.5.1 Two Branches Approach

A first workflow is based on **two branches approach**:

- **`master`.** This branch is used for official releases of our project. This branch contains only working stable versions of our project that other people can install and use.
- **`develop`.** This branch is used for the development of new features or any other operation required for project development (e.g., debugging or documentation). This branch contains the latest updates but the version could not be stable or not even working.

Following this approach, a stable version of our project will be always available in the `master` branch and we can proceed in the development without warring about breaking our project in the `develop` branch. Once we reach a new stable point, we can release a new version of our project by merging the `develop` branch in the `master` branch. Note that we do not delete the `develop` branch after the merge, but we keep using it for the project development.



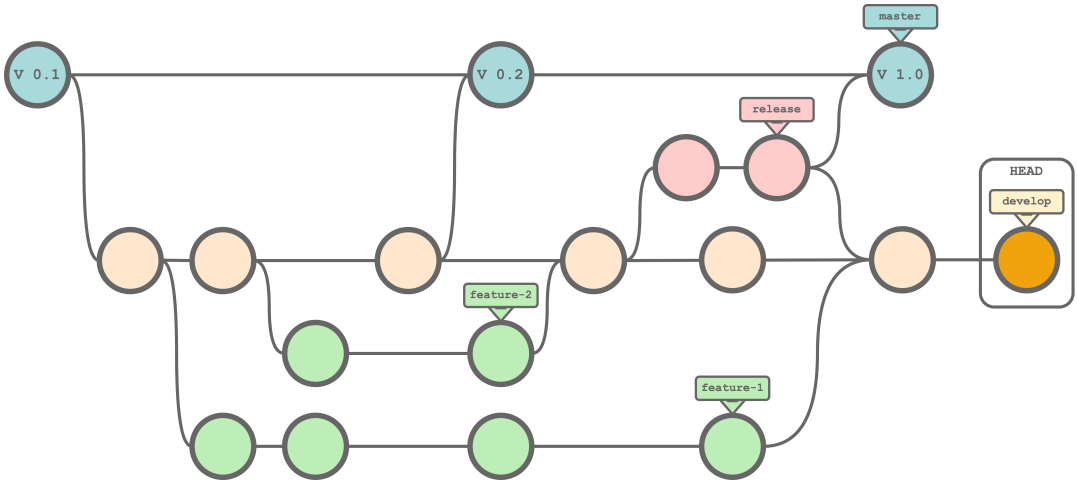
7.5.2 Multiple Branches Approach

If the project is more complex and composed of multiple parts, we can follow another workflow based on **multiple branches approach**:

- **`master`.** As before, this branch is used for official releases of our project. This branch contains only working stable versions of our project that other people can install and use.
- **`develop`.** Again, this branch is used for the project development. This time, however, any new feature is developed on a separate branch and in this branch we maintain a *development* version of the project with the latest working stable updates.
- **`feature-*`.** Any new feature is developed independently on a separate branch and merged back into the `develop` branch only when ready.
- **`release`.** When ready for a new release, this branch is used to ultimate the last changes before merging into the `master` branch. This branch is used to fixing bugs, creating documentation or other minor changes required for the release.

Following this approach, we can manage independently the development of different features. Only when a new feature is ready, we merge it into the `develop` branch and then we delete the (now useless) feature branch. In this way, we always keep a working version of the project with the latest updates in the `develop` branch from where we can start working on new features or fix other problems.

When we are ready for a new release we can use the `release` branch for the last changes required before releasing while the development can freely continue on the `develop` branch. Note that after the release, before deleting the `release` branch, it is important to merge the `release` branch also into the `develop` branch to keep all updates during the development.



For more information and about the Git Workflow, see https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow.

7.6 RStudio Git GUI

RStudio provides a useful GUI to manage the Git workflow. In this section, we describe how to configure Git in RStudio and use the RStudio GUI to manage the principal Git operation using the RStudio GUI. More details are provided in:

- https://support.rstudio.com/hc/en-us/articles/200532077
- https://r-pkgs.org/git.html

Remember, GUIs are attractive, but it is important to familiarise also with the Git command line interface. No GUI provides all the commands and options available in Git. At some point there will be no button to click to solve our problems but a single line of code could save us.

7.6.1 Configure Git in RStudio

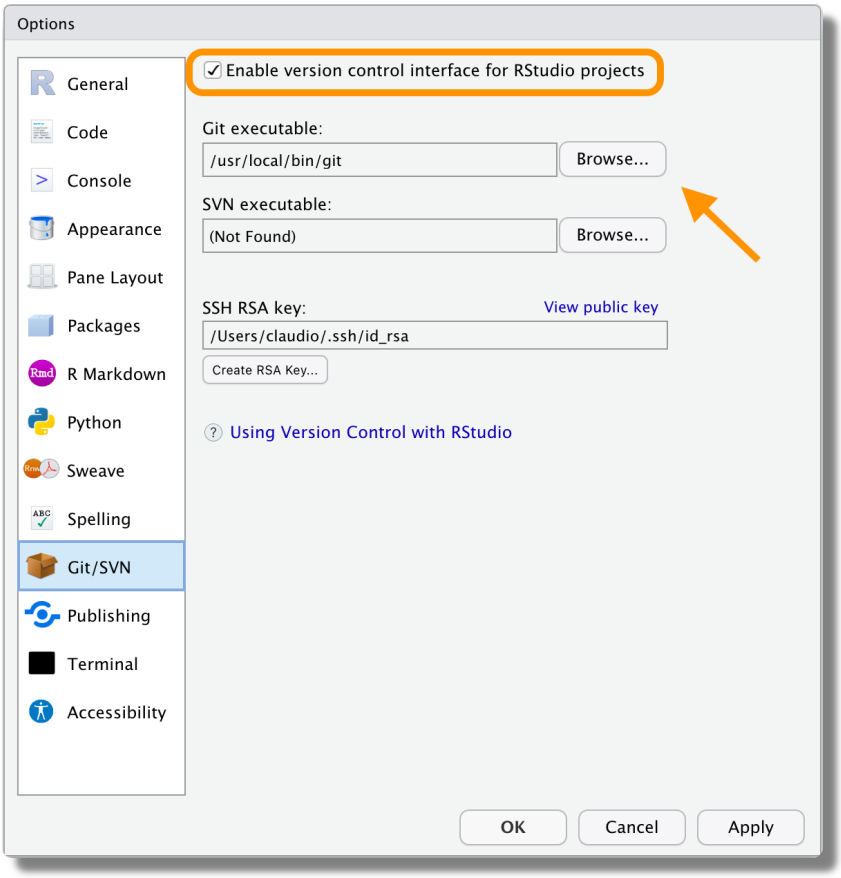
To use Git in RStudio, we need to activate the version control interface. To do that:

1. Open the *"Tools -> Global Options"*
2. Select the *"Git/SVN"* tab

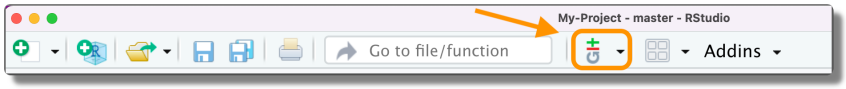
3. Select *"Enable version control interface for RStudio projects"*

4. Specify the path to the Git executable (in Windows, we can choose to use Git Bash as shell). To check where Git is installed, we can run from the terminal,

```
# Windows
where git

# Linux (macOS)
which git
```



5. Restart RStudio, the Git icon should appear in the toolbar



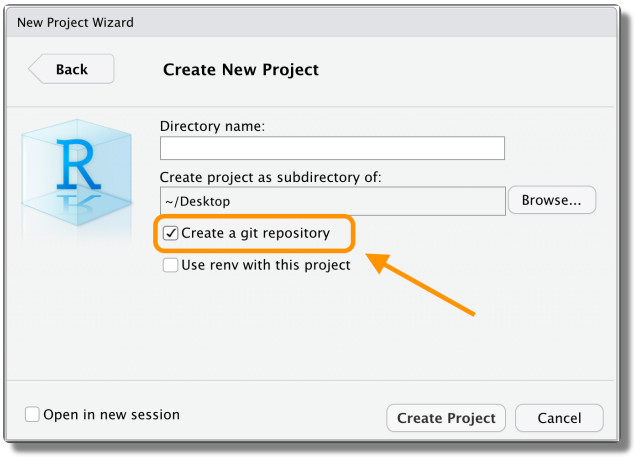
7.6.2 Using Git in Rstudio

We now describe how to execute the main Git operation using the RStudio GUI.

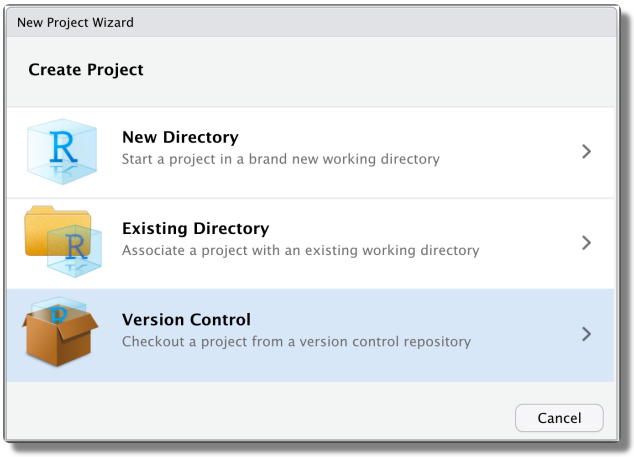
7.6.2.1 Create a Git Repository

We can create a Git repository in different ways:

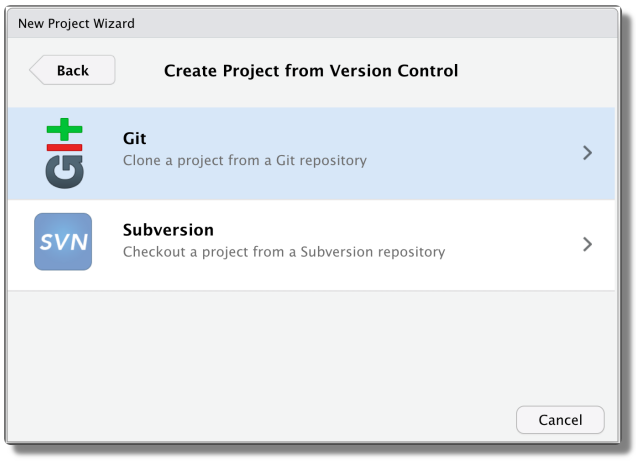
- **Create a New Project.** When creating a new project in a new directory (*"File > New Project… > New Directory"*), select the *"Create a git repository"* option.



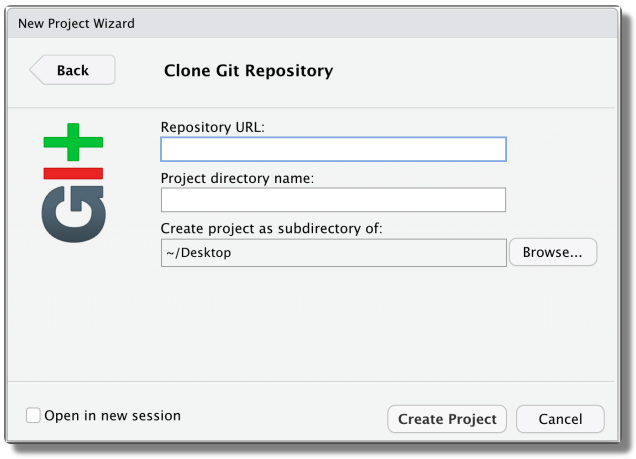
- **Clone a Remote Repository.** We can create a new project cloning a remote repository. From *"File > New Project…"*, select *"Version Control"*.



Next, select *"Git"*.



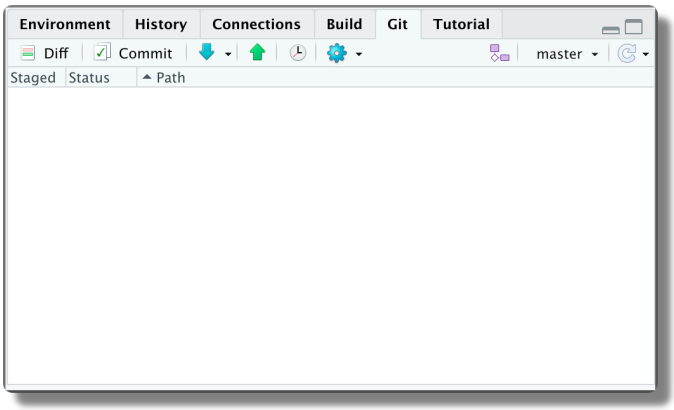
Specify the remote URL and click *"Create Projet"*.



- **Init Existing Project.** To enable Git in an already existing project, we can use the command `git init` from the terminal. Next, we need to restart RStudio (close and open it again).

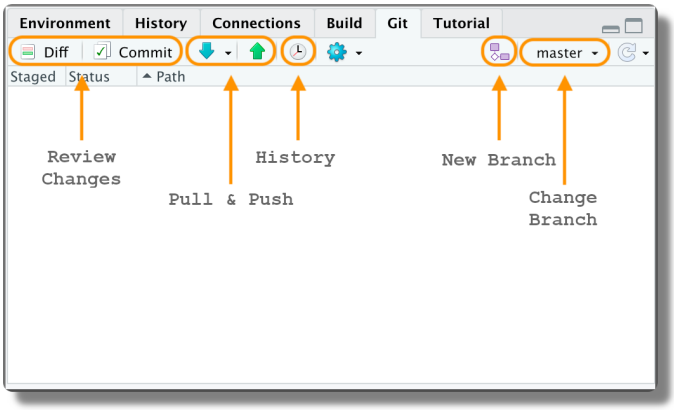
7.6.2.2 Git Panel

When working on a Git repository in RStudio, a new panel named *"Git"* is displayed (by default in the top-right corner).



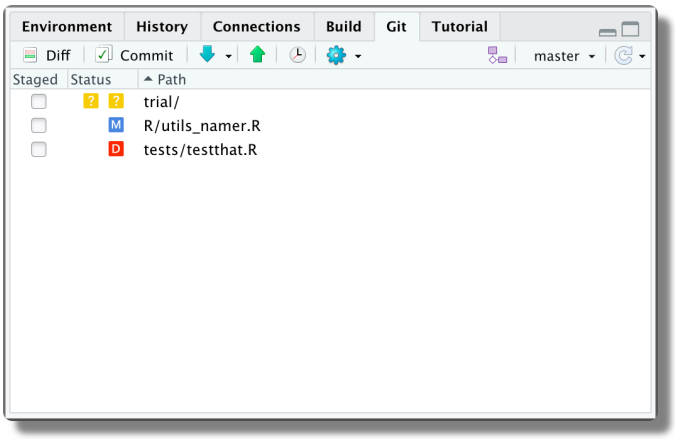
This panel allows us to execute the most common Git operations. In particular,

- **Review Changes.** Open the *"Review Changes"* window to evaluate the specific differences within each file and create new commits.
- **Pull & Push.** Pull or push commits between the current branch and its remote upstream.
- **History.** Navigate the commit repository history.
- **New Branch.** Create a new branch (local and remote).
- **Change Branch.** Change the active branch.



7.6.2.3 Tracking Changes

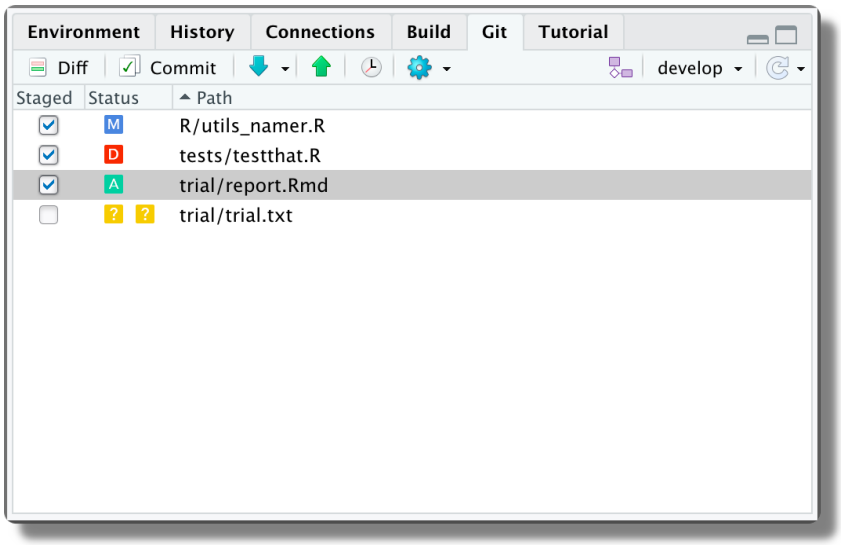
When we create, modify or delete files, these will be listed in the Git panel.



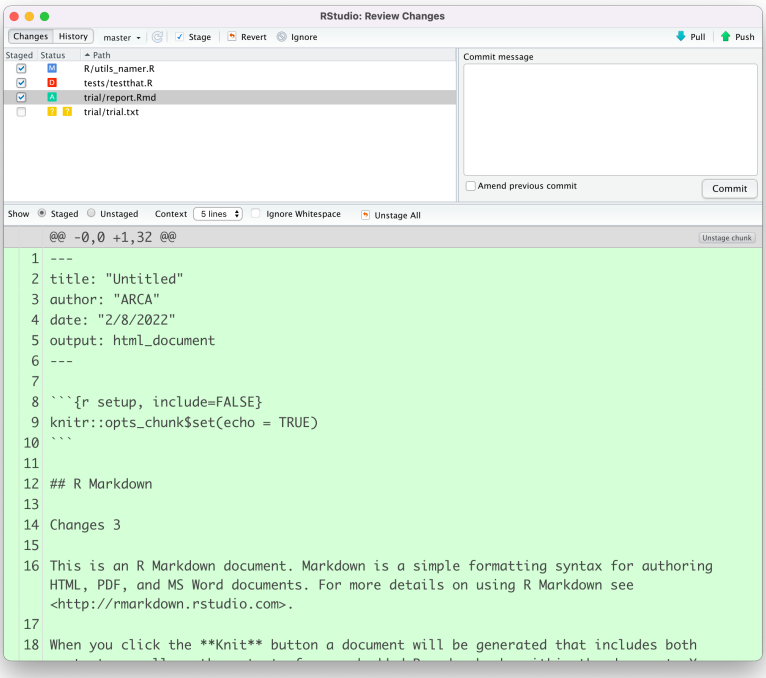
Each file is preceded by an icon indicating the file status:

-  **Untracked.** A new untracked file.
-  **Added.** An untracked file is added to the staging area.
-  **Modified.** The content of the file has been modified.
-  **Deleted.** The file has been deleted.
-  **Renamed.** The file has been renamed.
-  **Conflict.** The file contains a merge conflict.

- **Staging and Commiting.** We can stage files directly form the Git panel by selecting the desired files.

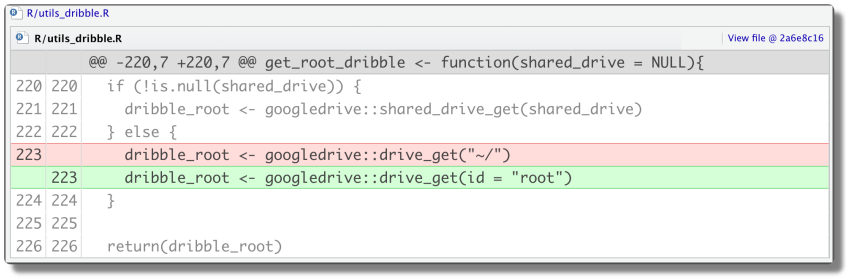


Alternatively, we can open the *"Review Changes"* window by clicking the *"Diff"* or the *"Commit"* button. This window allows us to review changes in each file before staging them.

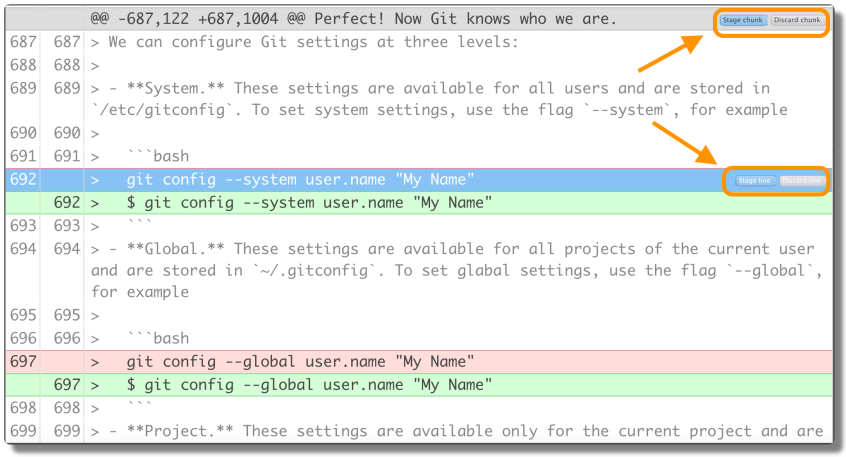


File specific changes are displayed highlighting in red the deleted lines and in green

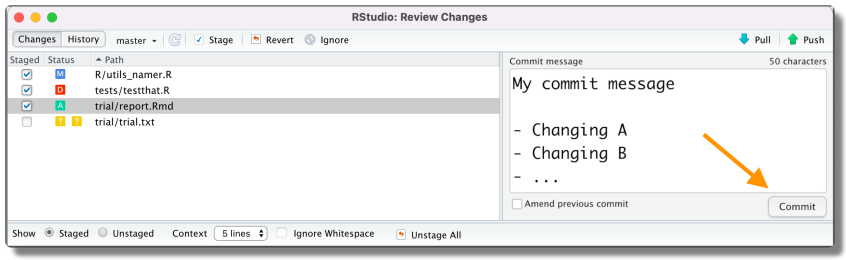
the new lines. Reviewing changes is very useful to check that we did not accidentally changed some part of the code.



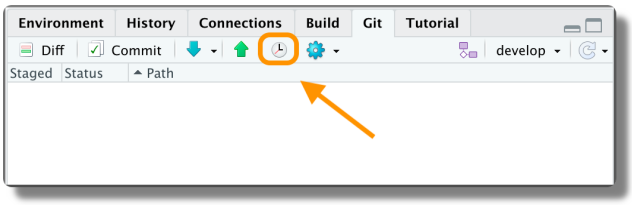
We can also stage or discard single chunks or a single line of text.



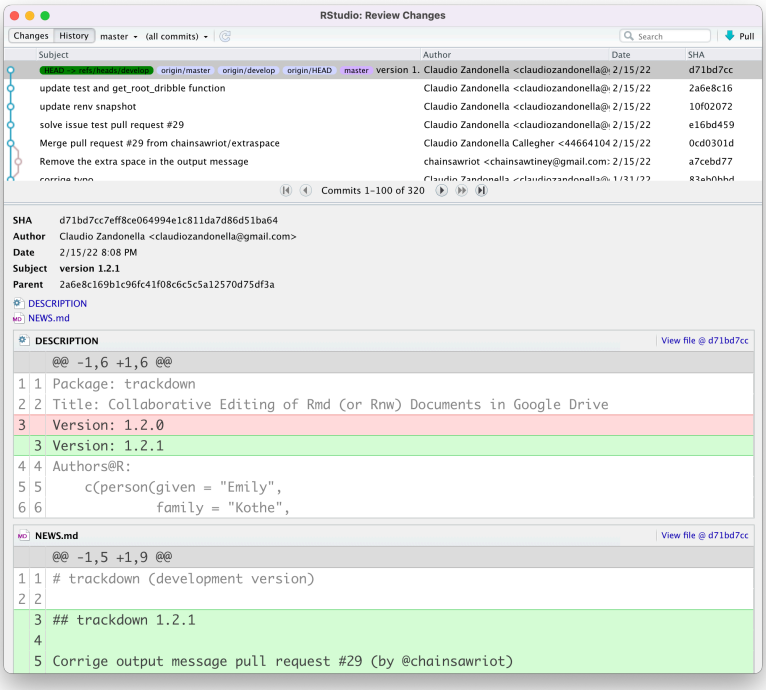
Once we have staged all the desired files, we write the commit message and click the *"Commit"* button.



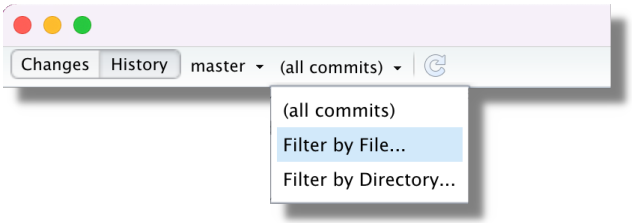
- **Commit History.** We can open the commit history window by clicking the history icon (or selecting *"History"* from the *"Review Changes"* window).



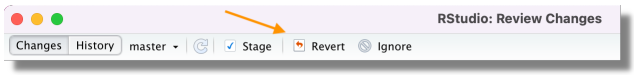
From here, wee can check all previous commits and see all the changes to the files.



Note, that we can filter commit history according to a specific branch or even a specific file.



- **Undoing Changes.** From the *"Review Changes"* window, we can undo current changes by selecting the desired file and clicking *"Revert"* from the top menu.



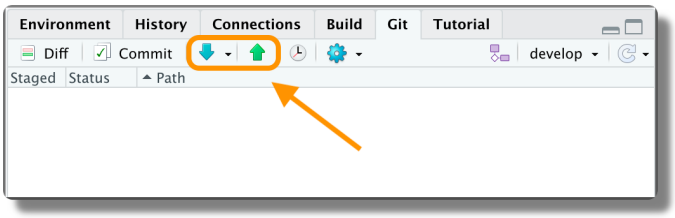
We can not undo commits or modify commits history from the RStudio GUI. All these actions can be done using the Git command line interface.

7.6.2.4 Managing Collaboration

To add the remote repository URL, we need to use the terminal. The command is,

```
$ git remote add <remote-name> <remote-URL>
```

Once we set the remote, repository we can proceed by Pushing and Pulling commits using the arrows in the Git panel (or in the *"Review Changes"* window).



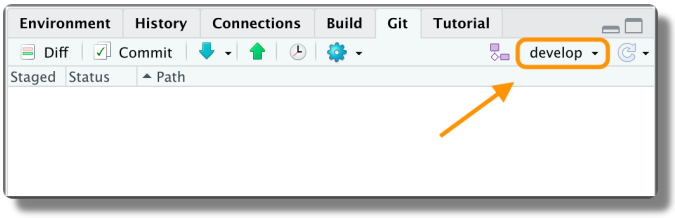
Warning-Box: SSH Required

To collaborate using remote repositories, we need specific authorization or authentication protocol (e.g., login credentials or SSH keys).

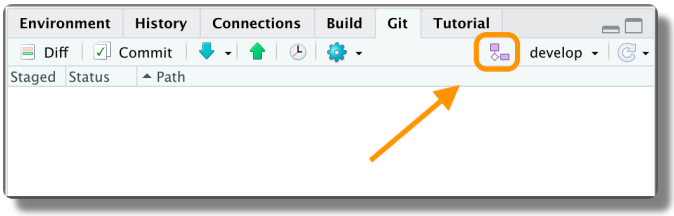
We discuss authentication procedures and SSH protocols in Chapter 8.1.2.

7.6.2.5 Branching and Merging

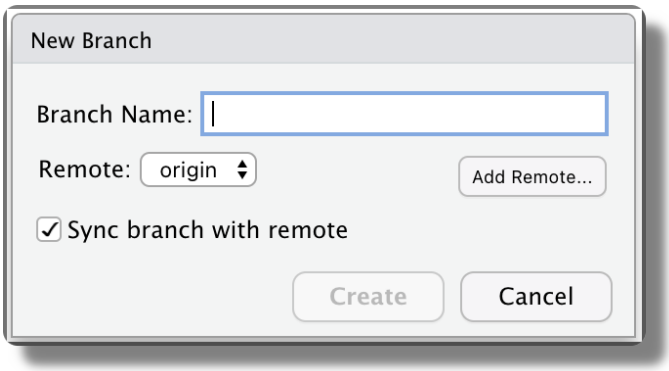
We can change the currently active branch from the top-down menu in the Git panel.



We can create a new branch clicking the branch icon in the Git panel.



A pop-up window will open in which wee can specify the branch name. By default, also an upstream branch in the remote repository is created.



We can not merge branches from the RStudio GUI. Merging operations can be done using the Git command line interface.

Documentation-Box

Install Git

- Install git
  https://git-scm.com/downloads.

Git Tutorials

- Bitbucket Git Tutorial
  https://www.atlassian.com/git/tutorials
- Pro Git Book
  https://git-scm.com/book/en/v2
- Git Official Manual
  https://git-scm.com/doc

Git Extra

- Configuration files
  https://stackoverflow.com/questions/8801729/is-it-possible-to-have-different-git-configuration-for-different-projects/5412596
- .gitignore file
  https://www.atlassian.com/git/tutorials/saving-changes/gitignore
- Fetching
  https://www.atlassian.com/git/tutorials/syncing/git-fetch
- Git workflows
  https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow.

RStudio Git GUI

- Version Control with Git and SVN
  https://support.rstudio.com/hc/en-us/articles/200532077
- Git and GitHub
  https://r-pkgs.org/git.html

8 GitHub

In Chapter 7, we introduced Git, a very powerful tool for tracking changes and managing the development of our project. To collaborate with other colleagues, however, we need to share a remote repository. We can not share a Git repository on common cloud storage services (e.g., Google Drive or DropBox) as they do not have Git running behind.

To collaborate with other colleagues using remote repositories, we need dedicated hosting services such as GitHub, GitLab, or Bitbucket. These services are similar to each other and they provide almost the same set of tools. In this chapter, we focus on the use of GitHub.

8.1 Using GitHub

GitHub (https://github.com/) is the most popular service and it is commonly used to share open-source projects or as an online portfolio for developers. Using Github, we can share our projects allowing other users to easily navigate our code and documentation, report issues or contribute to the development by suggesting new features or other improvements.

In this section, we provide a brief guide on how to work with Git in combination with GitHub. For more details and advanced features, see the official documentation at https://docs.github.com/.

Details-Box: OSF vs GitHub

OSF and GitHub may seem very similar tools, as both services allow us to share files and keep track of the different file versions.

However, the OSF and GitHub are designed for different purposes:

- **OSF** aims to facilitate scientific research by allowing easy sharing of research materials.
- **GitHub** is a Git repository-hosting service with a focus on software development.

Therefore, there are some important differences. For example, considering file tracking, the OSF simply stores all previous versions of a file as new ones are uploaded. This is nothing like a true version control system like Git. Only Git allows to properly track changes and manage collaboration (branching and merging) during the development of a project. The OSF is not a Git repository-hosting service.

However, although OSF lacks the power of Git, it serves a more general purpose allowing to share files of any type (e.g., PDFs and images), does not require specific technical competencies to be used, and provides specific features useful in scientific research (e.g., getting a DOI or anonymization for review process). On the contrary, Git (and therefore GitHub) is recommended only for plain-text files and requires specific technical competencies.

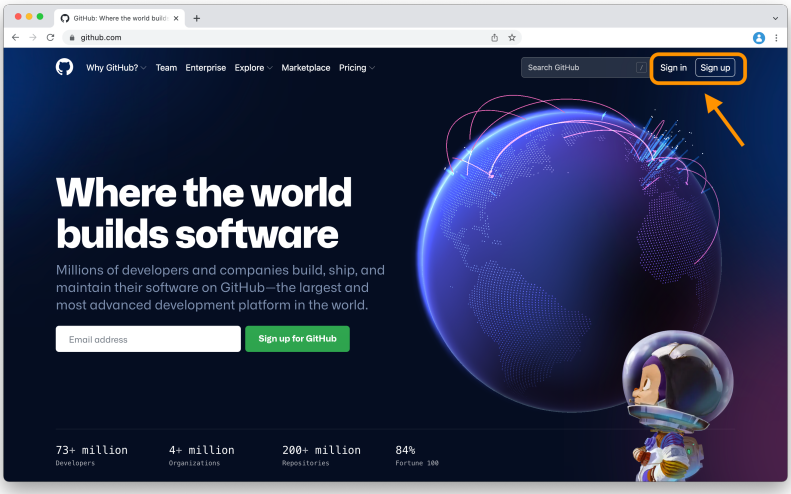
To know more about the differences between OSF and GitHub, see https://ropensci.org/blog/2020/08/04/osf/ or *"Sharing code"* by Kubilius (2014) available at https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4130510/.

Fortunately, we do not need to decide between the two, but we can connect our GitHub repository to an OSF project. To do that, follow the instructions at https://help.osf.io/article/211-connect-github-to-a-project.

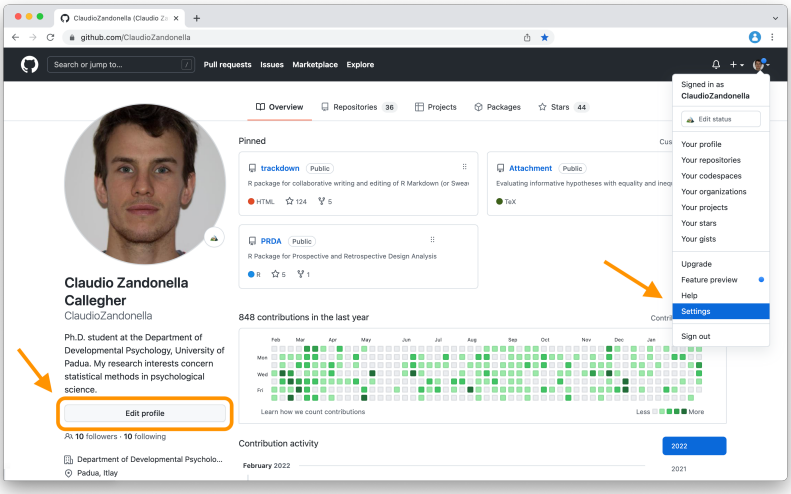
8.1.1 Initial Set Up

As a first step, we need to register on GitHub:

- **Create a GitHub Account.** Go to https://github.com and follow all the *"Sign up"* steps.



- **Edit your Profile.** GitHub is often used also as an online portfolio. Therefore, it is a good idea to fill our profile with useful information. To do that, click the *"Edit profile"* button on the profile page or go to *"Settings"*.



8.1.2 Authentication

When working on a remote repository, we need to authenticate using our password each time we pull from or push to the remote repository. This is extremely annoying. To authenticate without a password we need to use the **SSH protocol**.

The SSH protocol is based on two keys:

- A *private key* (`id_rsa`) that is saved on our local machine and should never be shared with others. We need to manage it with care.
- A *public key* (`id_rsa.pub`) that we share with online services to allow authentication.

SSH protocol allows our machine to communicate with online services without a password only if there is a matching pair of keys.

To enable SSH protocol:

1. **Generate SSH keys.** First we need to check if we already have an SSH key-pair available. These are usually saved in the `"~/.ssh/"` directory.

   If no SSH key is available, we can generate a new pair following the instructions at https://docs.github.com/en/authentication/connecting-to-github-with-ssh/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent. Note that we need to choose instructions according to our operating system. Moreover, if using WLS on Windows, check the dedicated section at https://www.howtogeek.com/762863/how-to-generate-ssh-keys-in-windows-10-and-windows-11/.

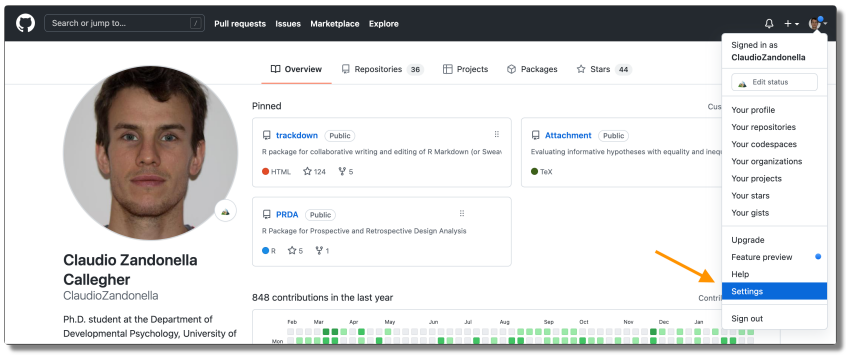
   Alternatively, we can generate an SSH key-pair directly from RStudio (see Box below).

2. **Copy the Public Key.** Next, we need to share our public key with GitHub. By default, our public key is saved at `"~/.ssh/id_rsa.pub"`. Open the file and copy the public key. We can do that directly from the command line by running,

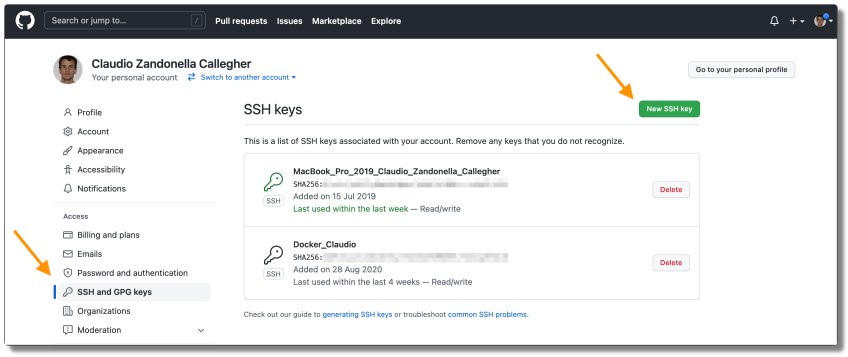
   ```
   # Windows
   clip < ~/.ssh/id_rsa.pub
   # Linux or macOS
   pbcopy < ~/.ssh/id_rsa.pub
   ```

   Our public key is automatically copied into our clipboard and we can now paste it with `Ctrl + V` (`command + V` in macOS).

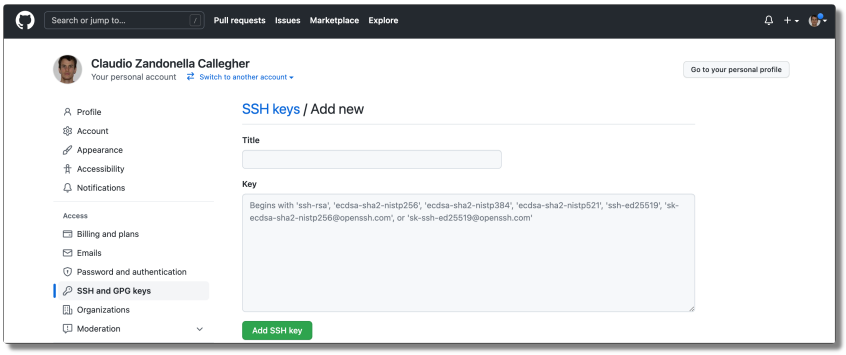
3. **Share the Public Key.** Now we need to add our public key to our profile on GitHub. To do that, we open our profile settings,



select the *"SSH and GPG keys"* panel and click on the *"New SSH key"* button.



Next, we paste our public key and assign a name. Finally, we press *"Add SSH key"*.



If we work using different machines (e.g., home and office), the best approach is to create a new pair of SSH keys for each machine and add them to our profile using descriptive names.

Warning-Box: Never Share the Private Key!

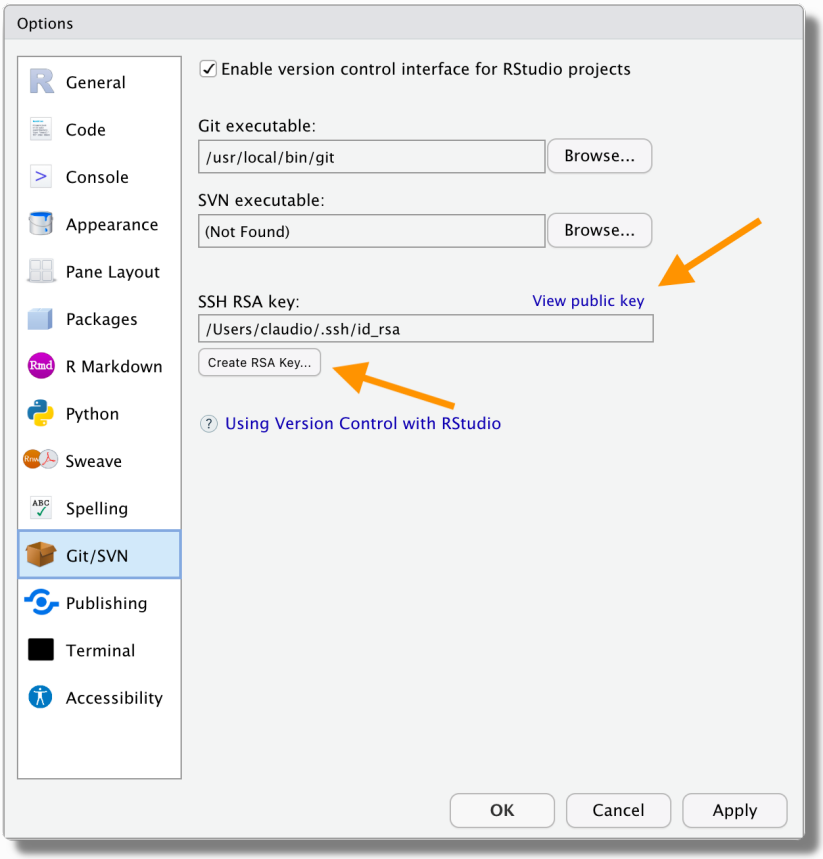
The *private key* (`id_rsa`) is extremely important, as it is used to identify our machine. Therefore we should keep it in a safe place and never share it with others.

In particular, we should pay attention to not sharing our private keys unintentionally by saving them inside directories shared with others or synchronized with cloud services.

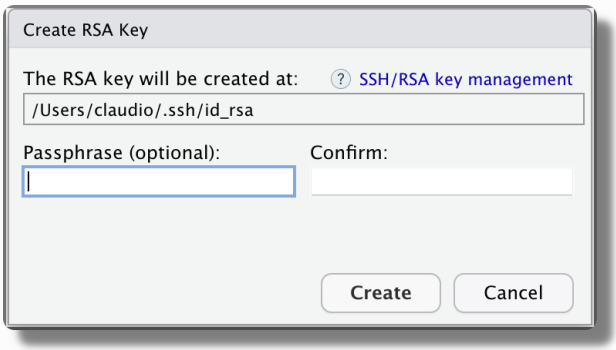
If someone manages to copy our private key, it will be able to access all services on which we registered the SSH key pairs.

Instructions-Box: SSH Keys in RStudio

In RStudio, we can easily view and copy our public SSH key. From *Tools > Global Options > Git/SVN*, we can view and copy our public SSH key by clicking *"View public key"*. If SSH keys are not available, we can create a new pair of keys by clicking the *"Create RSA Key..."* button.



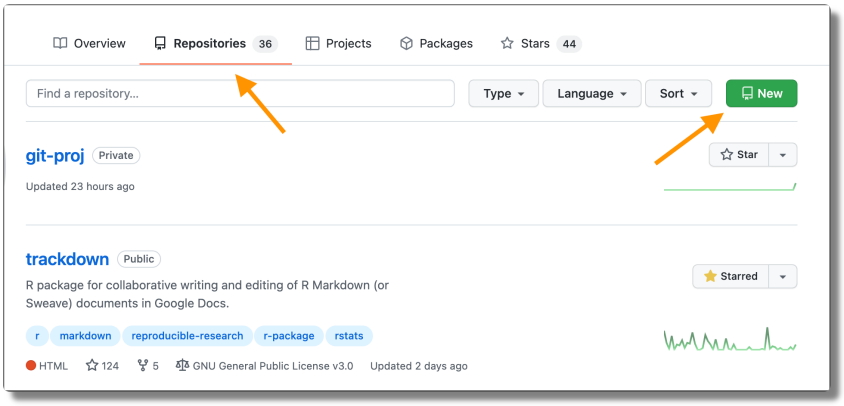
A new pop-up window will appear, and we click *"Create"*.



8.1.3 Create a Repository

Ready to create our first remote repository? Here is how to do it:

1. First, from our profile page we select the *"Repositories"* tab and then we click the *"New"* green button.

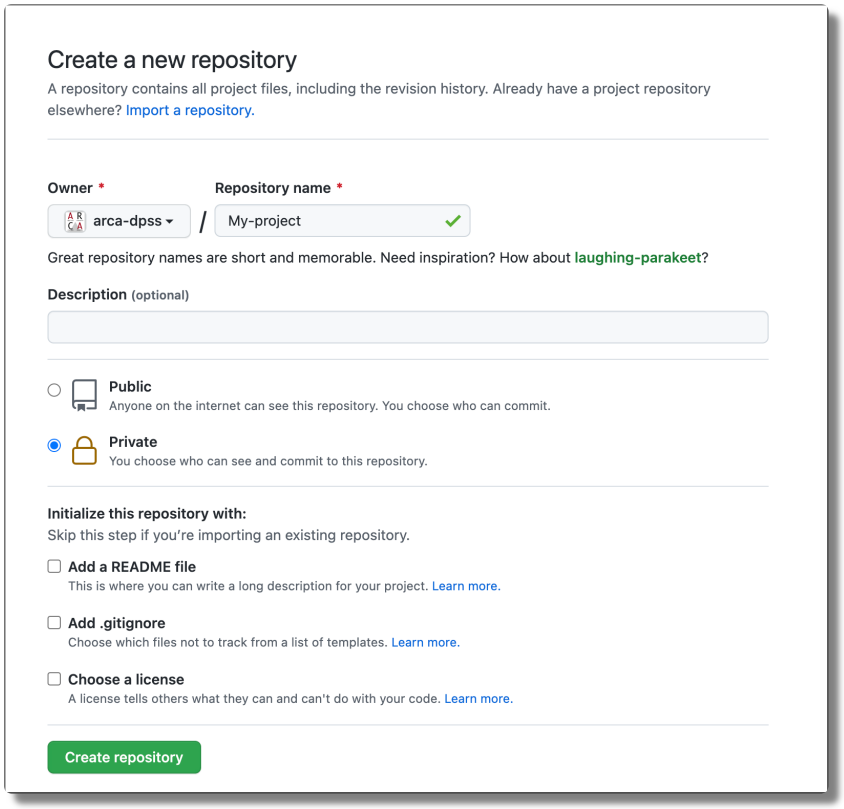


2. Next we complete a form with all the information about our repository. We need to specify a name for the repository (this has to be unique) and provide a description (optional but recommended if you want to help other users). Importantly, we have to choose the repository visibility:

   - **Public.** Anyone can see (and clone) the repository but only added collaborators can contribute (see Section 8.1.4).
   - **Private.** Only we and added collaborators can see and contribute to the repository.

   Moreover, if we are creating a repository from scratch, we can initialize it with some important files (e.g., `README`, `.gitignore`, `LICENSE`). We should not add any file if we intend to upload an already existing local repository.

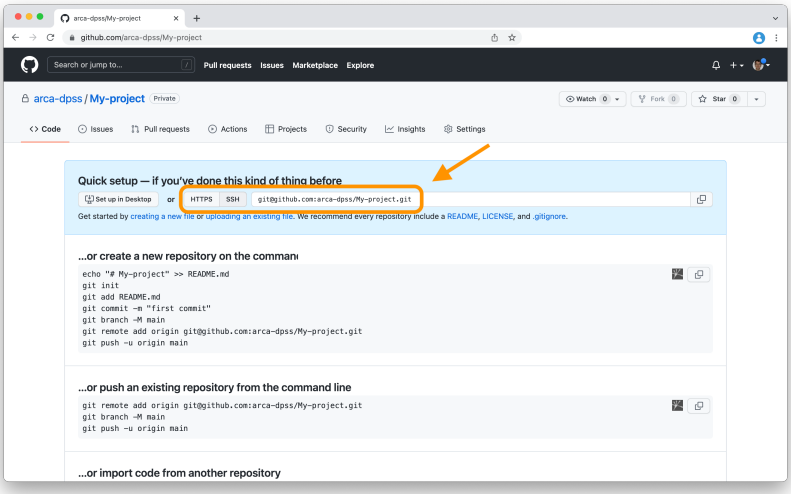
Once ready, we press the *"Create repository"* green button at the bottom. Note that we can also modify repository visibility or any other setting (e.g., name, owner, description, etc.) later from the repository settings.



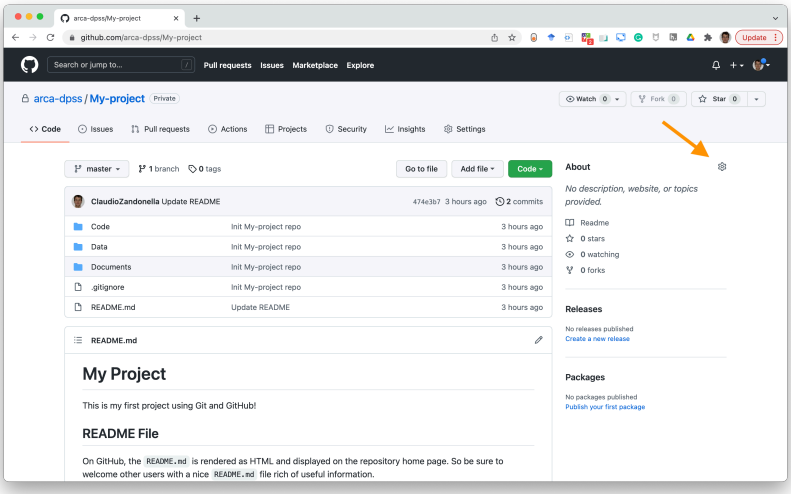
3. If we create an empty repository, it will display only some useful instructions. In particular, we obtain the remote repository URL that we can now add to our local repository and push all commits using the command,

```
git remote add origin <URL-repository>
git push -u origin master
```

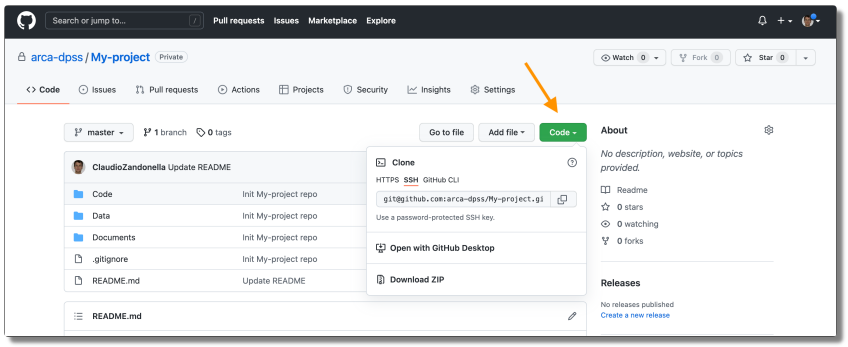
Note that two options are available: `HTTPS` and `SSH`. We should always select the `SSH` specific URL to allow authentication using the SSH protocol and without the password.



Once we added the remote URL and pushed all commits, all our files will be available online and we can now explore our project directly from GitHub. We can add further details about our repository (e.g., description, website link, or keywords) by clicking the gear icon on the top-right corner.



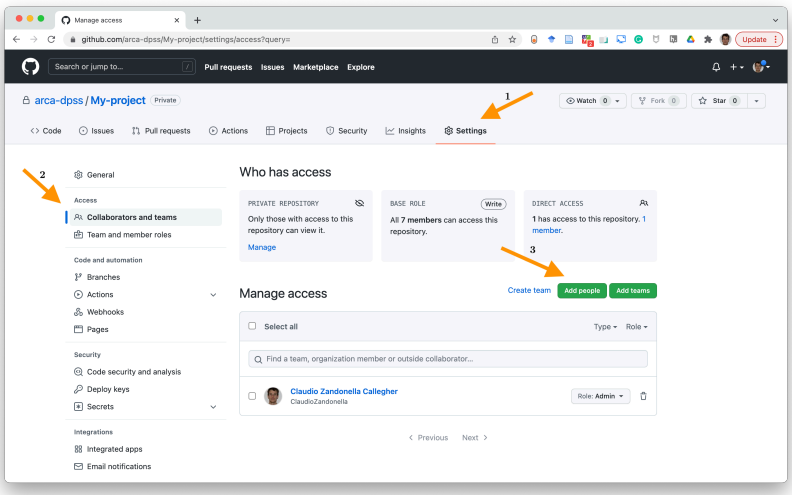
4. In the case we need the remote URL of any repository available on GitHub, we can obtain it by clicking the *"Code"* green button. Remember, we should always use the SSH one to allow automatic authentication.



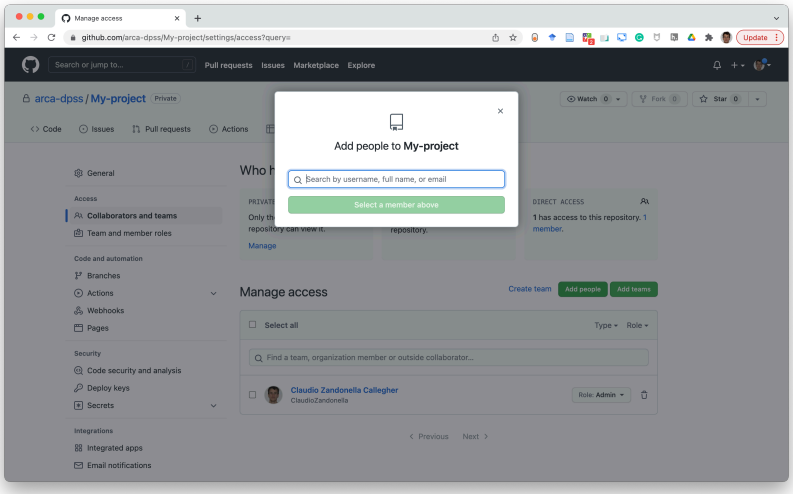
8.1.4 Add Collaborators

Independently of the repository visibility status (private or remote), only the owner and collaborators can contribute directly to a remote repository. Therefore, we need to add our collaborators to allow them to push their commits. To do that:

1. Open the repository settings and select the *"Collaborators and teams"* tab. Next, click the *"Add people"* green button.



2. A pop-up window will open allowing us to search and add our collaborators.



Even if a repository is public, only collaborators can contribute by pushing their commits directly to the repository. Other users from the GitHub community are not allowed to push commits directly to the repository. However, they can still contribute to any public repository using *Pull-Requests* as described in Section 8.3.2.

Finally, note that, even if we do not have any collaborator on a project, we can still use a remote repository on GitHub as a personal backup or to share the project results with the whole community.

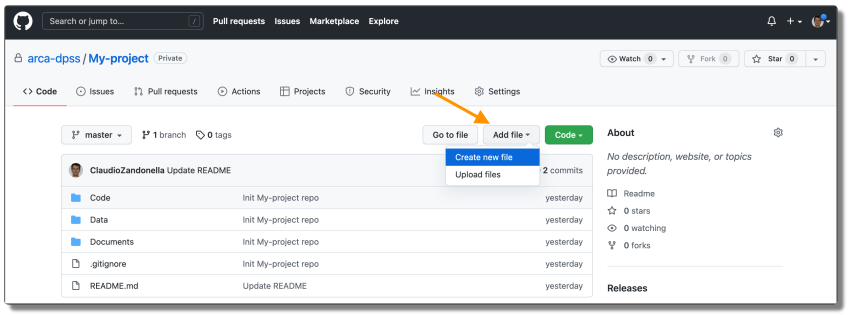
8.2 Main Features

Let's look at some of the main features of GitHub. Again, there are many things we can do with GitHub. It is worth spending some time reading the official documentation at https://docs.github.com/.

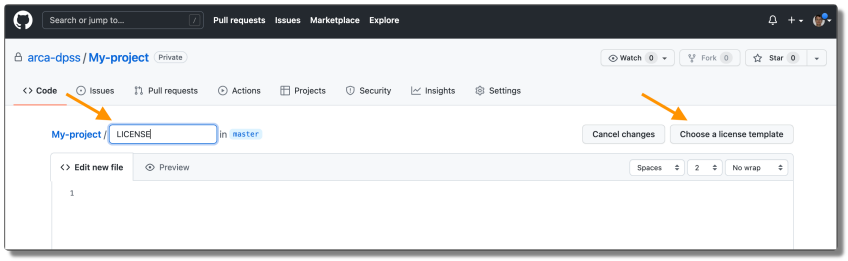
8.2.1 Adding License

GitHub provides many License templates we can use for our repository. To add a license to our repository using a template:

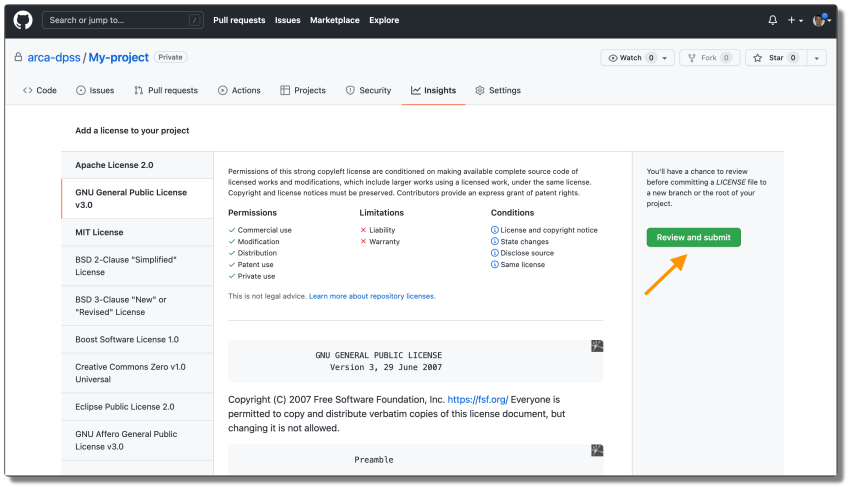
1. Click the *"Add file"* button and select *"Create new file"*



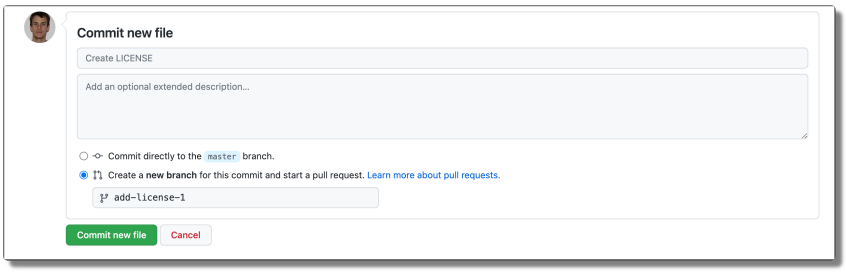
2. Type `"LICENSE"` in the file name field. The button *"Choose a license template"* will appear. Click it.



3. Now we can choose the preferred license and click the *"Review and submit"* button.



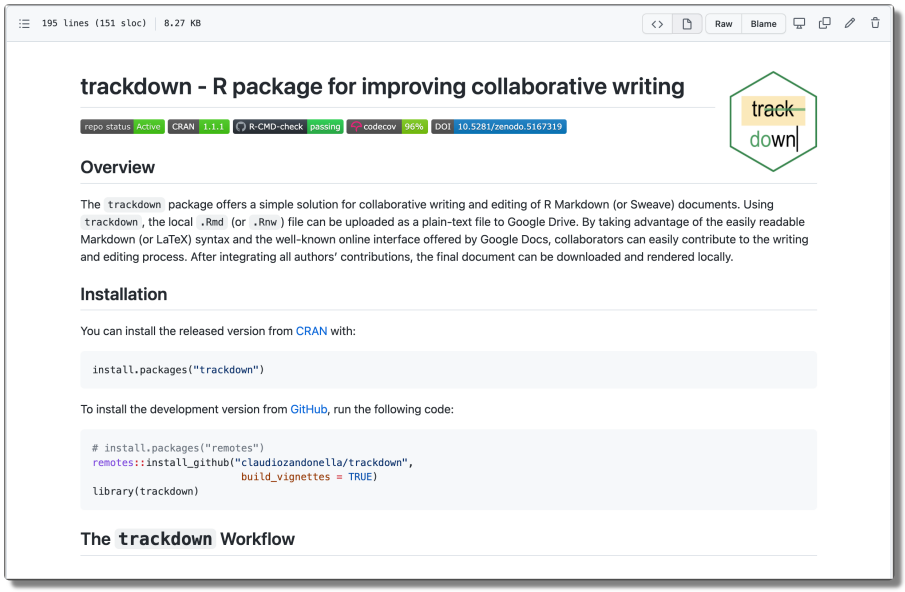
4. To add the license, we need to create a new commit. We write the commit message and click the *"Commit new file"* green button. Note that changes will be committed in the remote repository, thus, we need to pull the new commit from the local repository. Moreover, depending on the branch we used, we could be required to merge changes in the principal branch.



8.2.2 Documentation

GitHub automatically renders any `.md` file as an HTML allowing us to create nice-looking documentation for our project. In particular, if we include a `README.md` file in the root directory of our project, this will be displayed on the project homepage.

It is important to always create a `README.md` file providing an appropriate description of our project and all relevant information. Remember this is the first thing users will see when opening our project. We should use it to grab their attention and make them interested in learning more.

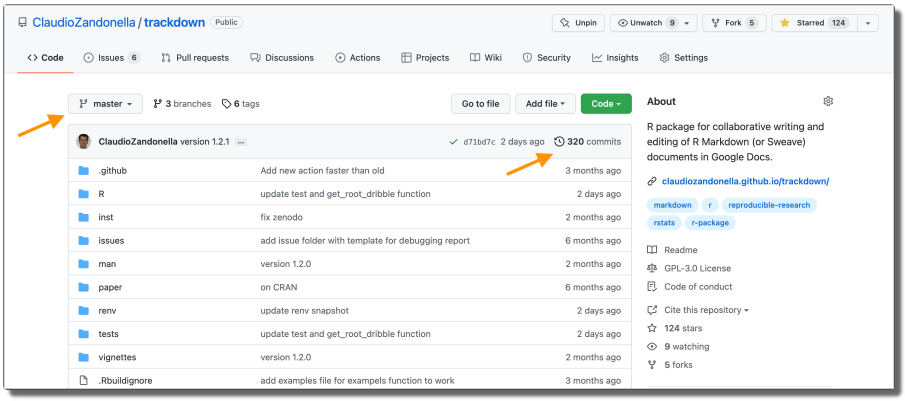


Note that, when navigating between the project directories, GitHub will automatically display any `README.md` file available in the current directory. We can take advantage of this by adding a `README.md` file within each directory providing specific information useful to guide other users.

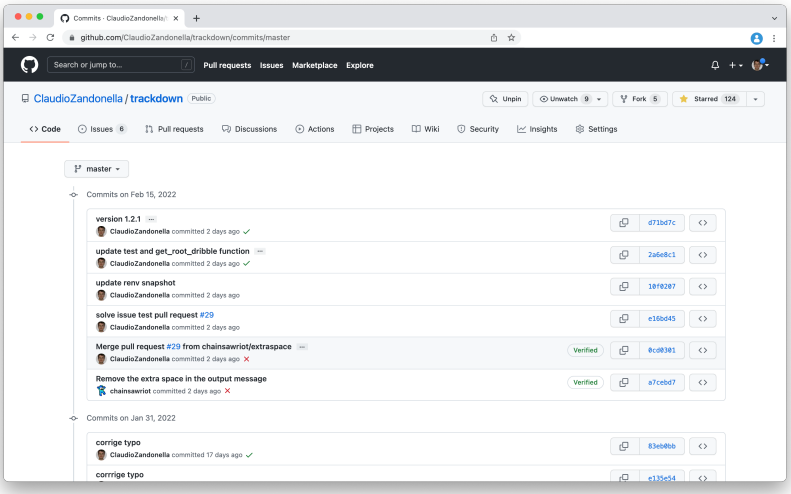
8.2.3 Commit History and Diff

From the repository home page, we can select the specific Git branch from the left drop-down menu and navigate between all our project files. Moreover, we can check the commit

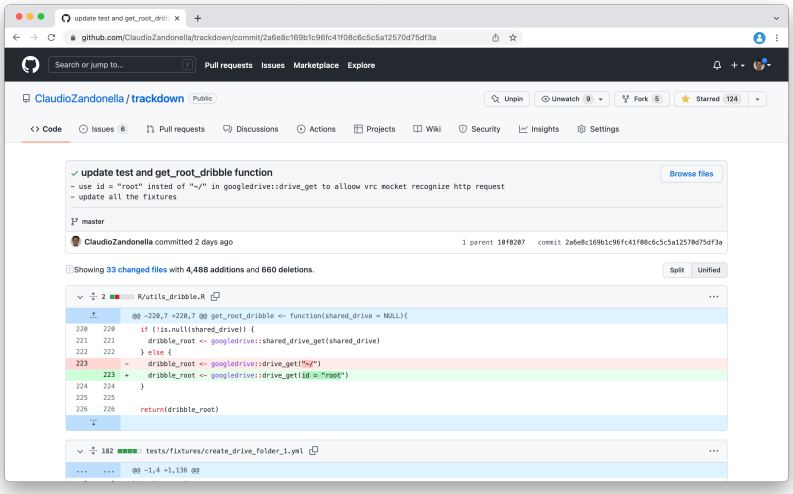
history by clicking the clock icon on the right with the number of commits.



- **Commit History.** We can visualize the commit history of the whole project by selecting a specific branch. Alternatively, if we open the commit history from a single file, we get only the commits relative to that file.



- **Commit Diff.** By clicking a single commit, we get more details together with the specific commit changes for each file.



8.2.4 Get DOI

To obtain a DOI for our repository, we need to use Zenodo. Zenodo is a general-purpose open repository developed under the European OpenAIRE program and operated by CERN that allows us to store research materials and creates a DOI for citing the content (from Wikipedia https://en.wikipedia.org/wiki/Zenodo).

To obtain a DOI for our GitHub repository, follow the instructions available at https://docs.github.com/en/repositories/archiving-a-github-repository/referencing-and-citing-content.

8.3 Contributing

One of the greatest things about Open Source is its community. Anyone can contribute to the improvement of Open Source projects and it is a very rewarding feeling. There are two main ways to contribute to someone else repository:

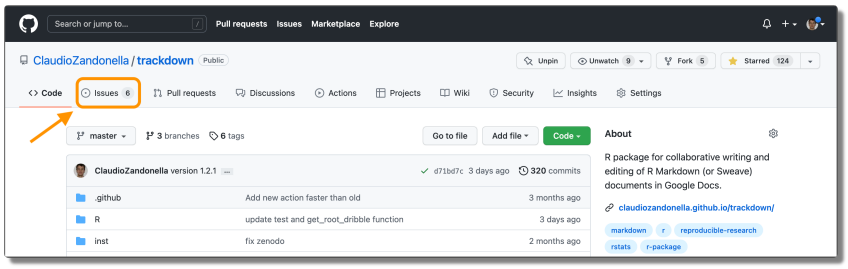
- **Issues.** Open an issue to report any bug, suggest possible improvements, or propose new features.
- **Pull Requests.** Contribute directly to project development by sending your code.

Let's describe these processes in more detail.

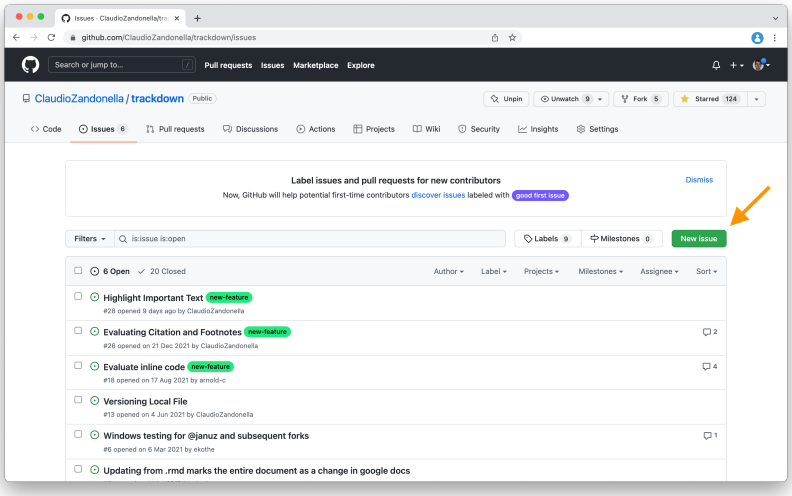
8.3.1 Issues

The first way to contribute to a project is to report any bug, suggest possible improvements, or propose new features. On GitHub, we can create an issue describing the bugs or new ideas. To open an issue:

1. Open the *"Issues"* tab from the repository homepage.



2. Click the *"New issue"* green button on the right.



3. We can now provide all the details about the bugs or the new ideas. If we are reporting some bugs or errors, it is extremely important to provide a minimal working example that allows reproducing the problems. By doing this, we help developers find a solution.

4. Once an issue is opened, we can write new comments to further discuss the issue with the project maintainers or other issues.

Note that by opening an issue, we do not need to work on the project code ourselves. By opening an issue, we contribute indirectly to the project development simply by reporting bugs or suggesting some improvements. Project maintainers would take care of solving the problems of implementing the new features (if considered appropriate).

Maintainers are usually very friendly and willing to help (they love to know that other people are using their products). However, we should be aware the majority of developers

working on open source projects do not get paid for them. Therefore, we must always be nice and patient if we do not get an answer immediately.

Moreover, we should avoid creating duplicate issues. Therefore, before opening a new issue, check if there are similar issues already opened or that have been already answered (remember to check the closed issues as well).

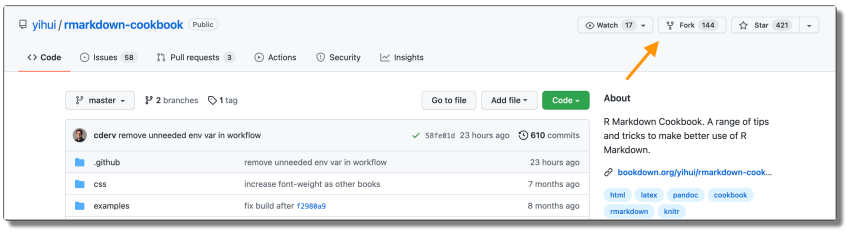
Finally, note that issues are used to report bugs and problems that require modifications of the project code itself. Anything related to the misuse or misunderstanding of the code is better addressed on other websites such as Stack Overflow.

8.3.2 Pull Request

Besides issues, we can also contribute directly to project development by sending our code. However, as discussed in Section 8.1.4, only users added to the repository as collaborators can push their commits directly to the remote repository. So, how can other users contribute to project development? The solution is to fork the repository, make the changes, and create a pull request.

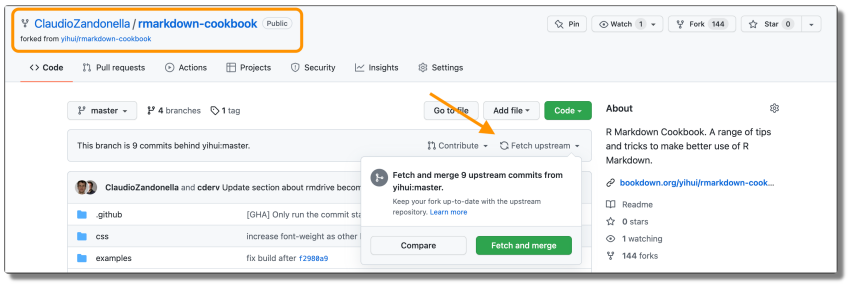
8.3.2.1 Fork and Fetch

- **Fork.** Forking allows us to create a personal copy of any public repository. In this way, we can make the desired changes to our personal copy without affecting the original repository. To create a fork, click the *"Fork"* button on the top-right corner of the repository homepage. The forked repository is automatically added to our GitHub profile.



- **Fetch.** Creating a fork is more than a simple copy of a repository. A forked repository always maintains a link to the original repository. This allows us to keep our forked copy up-to-date with the original repository.

  If new changes are available in the original repository, we can make them available in our fork by clicking *"Fetch upstream"* on our forked repository homepage and selecting *"Fetch and Merge"*.

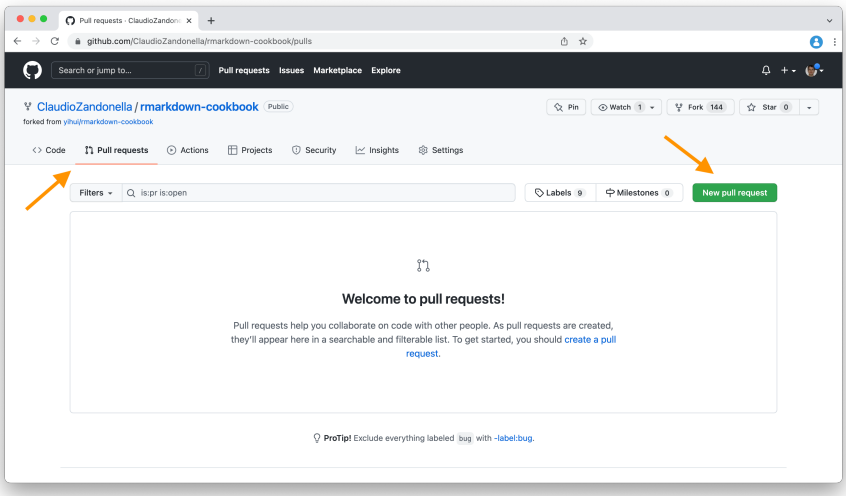


8.3.2.2 Create a Pull Request

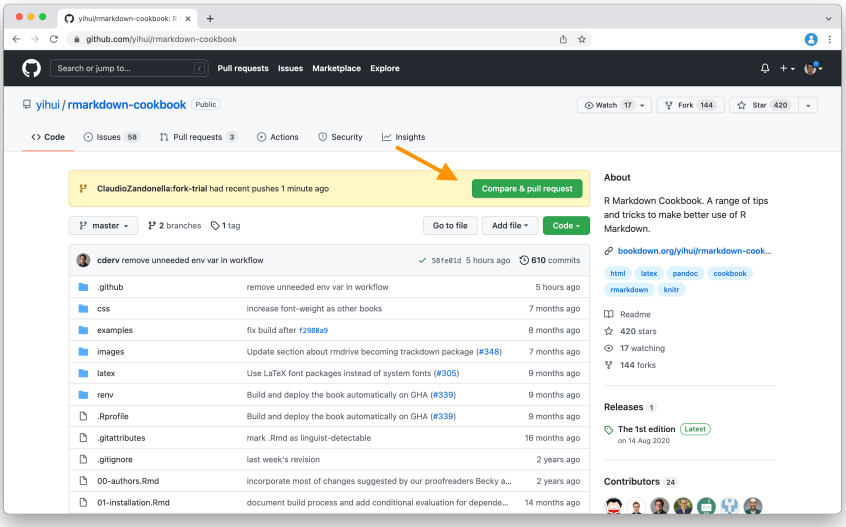
Once we made the desired changes and pushed the new commits, these will be available only in our forked repository but not in the original repository. If we want to share our changes with the original repository as well, we can create a *"pull request"*.

Creating a pull request, we inform the maintainers of the original repository about our changes. Once a pull request is opened, maintainers will evaluate and review our changes and we can discuss them. Maintainers could ask us for further improvements or other modifications and, when everything is ready and stable, they will merge our commits in the original repository.

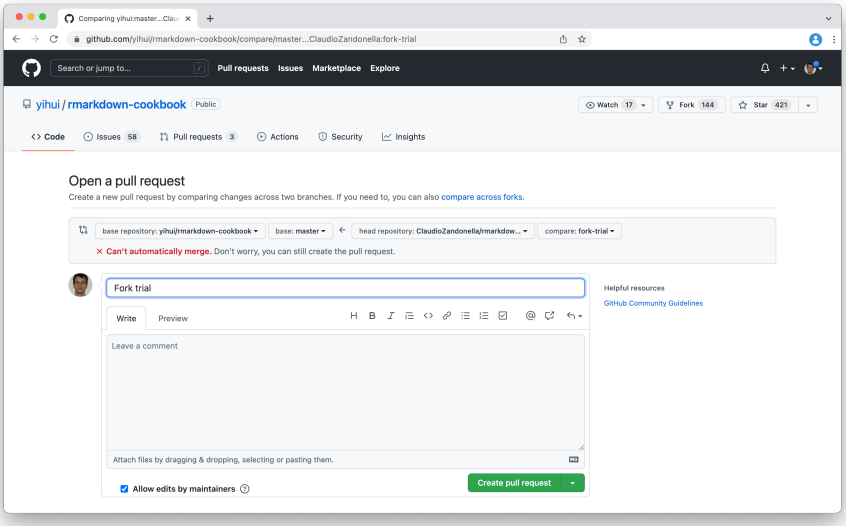
To create a pull request, open the *"Pull request"* tab in our forked repository homepage and click the *"New pull request"* green button.



Alternatively, if we just pushed the commits to our remote repository, we can visit the original repository homepage. A message is displayed about our recent push and we can click the *"Compare & pull request"* green button.



Next, we need to choose the branch in our fork repository from which to pull and the destination branch in the original repository. Finally, we fill the form with all the details about our changes and we click the *"Create pull request"* green button.



Tip-Box: Pulling or Pushing?

Note that it is called pull request and not push request as we are not pushing to the original repository but the original repository is pulling the

commits from our fork.

8.4 Advanced Features

Finally, we briefly present some other very useful but more advanced GitHub features. As always we recommend considering the official documentation at https://docs.github.com/ for all the options and features.

- **Github Pages.** We can create a website directly from our GitHub repository. This is an extremely useful feature for documenting our repository or sharing our results. For example, consider this R package documentation https://claudiozandonella.github.io/trackdown/. Even the online version of this book is hosted by GitHub Pages!

  In particular, we can collect all the files used to create the website in a specific project folder or a specific project branch. Github Pages will build the website using a static site generator. By default, GitHub uses Jekyll but we can also use other static site generators.

  For more details about GitHub Pages, see the official documentation at https://docs.github.com/en/pages.

- **Github Actions.** We can define a specific set of actions that are automatically executed each time we push new commits to the remote repository (or a specific branch). For example, we could ask Github to run unit tests, compile the documentation, or run the analysis each time we push new commits. These tools are commonly used in the continuous integration and continuous delivery (CI/CD) workflow to automate the process.

  For more details about GitHub Actions, see the official documentation at https://docs.github.com/en/actions.

- **GitHub Gists.** If we have some useful snippets of code, we can share them using GitHub Gists. A gist is like a minimal repository for easily sharing code. However, Gists are still Git repositories. Therefore, we can fork or clone any gist and view the commit history.

  For more details about GitHub Gists, see the official documentation at https://docs.github.com/en/get-started/writing-on-github/editing-and-sharing-content-with-gists.

- **GitHub Desktop.** GitHub provides a Desktop application to manage the Git and GitHub workflow. For more details about GitHub Desktop, see https://docs.github.com/en/desktop.

 Documentation-Box

GitHub

- GitHub link
  https://github.com/

GitHub Extra

- Generate SSH Keys
  https://docs.github.com/en/authentication/connecting-to-github-with-ssh/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent
- Add DOI
  https://docs.github.com/en/repositories/archiving-a-github-repository/referencing-and-citing-content
- GitHub Pages
  https://docs.github.com/en/pages
- GitHub Actions
  https://docs.github.com/en/actions
- GitHub Gists
  https://docs.github.com/en/get-started/writing-on-github/editing-and-sharing-content-with-gists
- GitHub Desktop
  https://docs.github.com/en/desktop

OSF and GitHub

- OSF vs GitHub
  https://ropensci.org/blog/2020/08/04/osf/
- "Sharing code" by Kubilius (2014)
  https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4130510/
- Connect GitHub and OSF
  https://help.osf.io/article/211-connect-github-to-a-project.

9 Workflow Analysis

In the previous chapters, we learned to organize all our files and data in a well structured and documented repository. Moreover, we learned how to write readable and maintainable code and to use Git and GitHub for tracking changes and managing collaboration during the development of our project.

At this point, we have everything we need to run our analysis. In this chapter, we discuss how to manage the analysis workflow to enhance results reproducibility and code maintainability.

9.1 Reproducible Workflow

To enhance results reproducibility we need to establish a workflow that will allow other colleagues to easily run the analysis. First, we describe how to organize the code used to run the analysis. Next, we discuss the importance of appropriate documentation and common issues related to results reproducibility. Finally, we discuss workflow management tools used to create specific pipelines for running our analysis. These tools allow us to improve the analysis maintainability during the project development.

9.1.1 Run the Analysis

In Chapter 5.1.2, we introduced the functional style approach that allows us to organize and develop the code required for the analysis very efficiently. In summary, instead of having a unique script, we define functions to execute each analysis step breaking down the code into small pieces. These functions are defined in separate scripts and subsequently used in another script to run the analysis.

Therefore, in our project we can organize our scripts into two different directories:

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

But how can we organize the scripts used to run the analysis? Well, of course, this will depend on the complexity of the analysis and its specific characteristics. However, let's see some general advice:

- **Single Script.** If the analysis is relatively short and straightforward, we can simply collect everything in a single script. Using the functions defined elsewhere, we specify all the analysis steps in the required order. To run the analysis, we execute the script line by line in the given order (from top to bottom).
- **Multiple Scripts.** When the analysis is very long or composed of different distinct steps, it is preferable to break down the analysis into different scripts. In this way, each script is used to run a specific part of the analysis. A good idea is to name each script with an auto-descriptive name preceded by a **progressive number** (e.g., `xx-<script-goal>`). Auto-descriptive names allow us to easily understand the aim of each script, whereas progressive numbers indicate the required order in which scripts should be executed. As later scripts may rely on results obtained in previous ones, it is necessary to run each script in the required order one at a time.
- **Main Script.** In the case of complex analysis with multiple scripts, it may be helpful to define an extra script to manage the whole analysis run. We can name this special script `main` and use it to manage the analysis by running the other scripts in the required order and dealing with other workflow aspects (e.g., settings and options). By doing this, we can run complex analyses following a simple and organized process.

Following these general recommendations, we obtain a well-structured project that allows us to easily move between the different analysis parts and reproduce the results. As a hypothetical example, we could end up having a project with the following structure.

```
- my-project/
    |
    |-- analysis/
    |     |-- 01-data-preparation
    |     |-- 02-experiment-A
    |     |-- 03-experiment-B
    |     |-- 04-comparison-experiments
    |     |-- 05-sensitivity-analysis
    |     |-- main
    |-- code/
    |     |-- data-munging
```

```
|     |-- models
|     |-- plots-tables
|     |-- utils
```

9.1.2 Documentation

Independently of the way we organize the scripts used to run the analysis, it is important to always provide appropriate documentation. This includes both comments within the scripts to describe all the analysis steps and step-by-step instructions on how to reproduce the analysis results.

- **Comments Analysis Steps.** In Chapter 5.1.1.3, we described general advice on how to write informative comments for the code. In summary, comments should explain "*why*" rather than "*what*." However, as we are commenting on the analysis steps rather than the code itself, in this case, it is also important to clearly describe "*what*" we are doing in the different analysis steps.

  Of course, we still need to provide information about the "*why*" of particular choices. However, specific choices during the analysis usually have theoretical reasons and implications that could be better addressed in a report (supplemental material or paper) used to present the results. Ideally, comments should describe the analysis steps to allow colleagues (not familiar with the project) to follow and understand the whole process while reading the analysis scripts.

- **Instructions Analysis Run.** Step-by-step instructions on how to run the analysis are usually provided in the `README` file. We need to provide enough details to allow colleagues (not familiar with the project) to reproduce the results.

Documenting the analysis workflow is time-consuming and therefore an often overlooked aspect. However, documentation is extremely important as it allows other colleagues to easily navigate around all the files and reproduce the analysis. Remember, this could be the future us!

9.1.3 Reproducibility Issues

A well structured and documented analysis workflow is a big step toward results reproducibility. However, it is not guaranteed that everyone will obtain the exact same results. Let's discuss some aspects that could hinder result reproducibility.

- **Random Number Generator.** During the analysis, some processes may require the generation of random numbers. As these numbers are (pseudo-) random, they will be different at each analysis run. For this reason, we could obtain slightly different values when reproducing the results. Fortunately, programming languages provide ad-hoc functions to allow reproducibility of random numbers generation. We should look at the specific functions documentation and adopt the suggested

solutions. Usually, we need to set the seed for initializing the state for random number generation.

- **Session Settings.** Other global settings related to the specific programming language may affect the final results. We need to ensure that the analysis is run using the same settings each time. To do that we can specify the required options directly in the analysis script as code lines to be executed.

- **Project Requirements.** Other elements such as operating system, specific software version, and installed libraries could affect the analysis results. In Chapter 11 and Chapter 12, we discuss how to manage project requirements to guarantee results reproducibility.

9.1.4 Workflow Manager

At this point, we would like something to help us manage the workflow. In particular, we need a tool that allows us to:

- **Automatically Run the Analysis.** Ideally, we should be able to run the whole analysis in a single click (or better a single command line), following a pre-defined analysis pipeline.
- **Discover Dependencies.** Tracking the dependencies between the different analysis parts allows for identifying the objects that are affected by changes or modifications in the code.
- **Update the Analysis.** Following changes in the analysis (or when new analysis parts are added), results should be updated computing only the outdated dependencies (or the new objects) that were affected by the changes made. In this way, we avoid re-running unnecessary analysis parts, optimizing the required time.
- **Caching System.** Saving copies of the intermediate and final analysis objects so they can be used later avoiding to re-run the analysis at each session.

A manager tool with these characteristics is particularly useful during the project development, allowing a very smooth workflow. In Section 9.1.4.1, we introduce Make, a Unix utility that allows us to automate tasks execution for general purposes. In Section 9.3, we present specific solutions for the R programming language.

9.1.4.1 Make

Make is a Unix utility that manages the execution of general tasks. It is commonly used to automatize packages and programs installation, however, it can be also used to manage any project workflow. In Windows, an analogous tool is NMake (see https://docs.microsoft.com/en-us/cpp/build/reference/nmake-reference)

Make has several powerful features. In particular, it allows us to define dependencies between the different project parts and it automatically figures out which files to update following changes or modifications. Moreover, Make is not limited to a particular language

but it can be used for any purpose. See official documentation for more details https://www.gnu.org/software/make/.

Make requires a `Makefile` (or `makefile`) where all the tasks to be executed are defined. `Makefile` has its own syntax that is beyond the aim of the present book. Interested readers can refer to this tutorial for a general introduction to Make https://opensource.com/article/18/8/what-how-makefile.

Ideally, we could create a `Makefile` with all the details and use Make to automatize the analysis workflow. This would be very useful but it requires some extra programming skills. In Section 9.3, we introduce alternative tools specific to the R programming language. However, Make may still be the choice to go if we need to integrate multiple programs into the workflow or for more general purposes.

9.2 R

Now we discuss how to manage the analysis workflow specifically when using the R programming language. First, we consider some general recommendations and how to solve possible reproducibility issues. Next, we describe the main R-packages available to manage the analysis workflow.

9.2.1 Analysis Workflow

In Chapter 5.2.2, we discussed how to create our custom functions to execute specific parts of the analysis. Following the R-packages convention, we store all the `.R` scripts with our custom functions in the `R/` directory at the root of our project.

Now, we can use our custom functions to run the analysis. We do that in separate `.R` scripts saved in a different directory named, for example, `analysis/`. Of course, during the actual analysis development, this is an iterative process. We continuously switch between defining functions and adding analysis steps. It is important, however, to always keep the scripts used to run the analysis in a separate directory from the scripts with our custom functions:

- `analysis/`: Scripts to run the analysis.
- `R/`: Scripts with function definitions.

Considering the previous example, we would have a project with the following structure.

```
- my-project/
    |
    |-- analysis/
    |     |-- 01-data-preparation.R
    |     |-- 02-experiment-A.R
    |     |-- 03-experiment-B.R
```

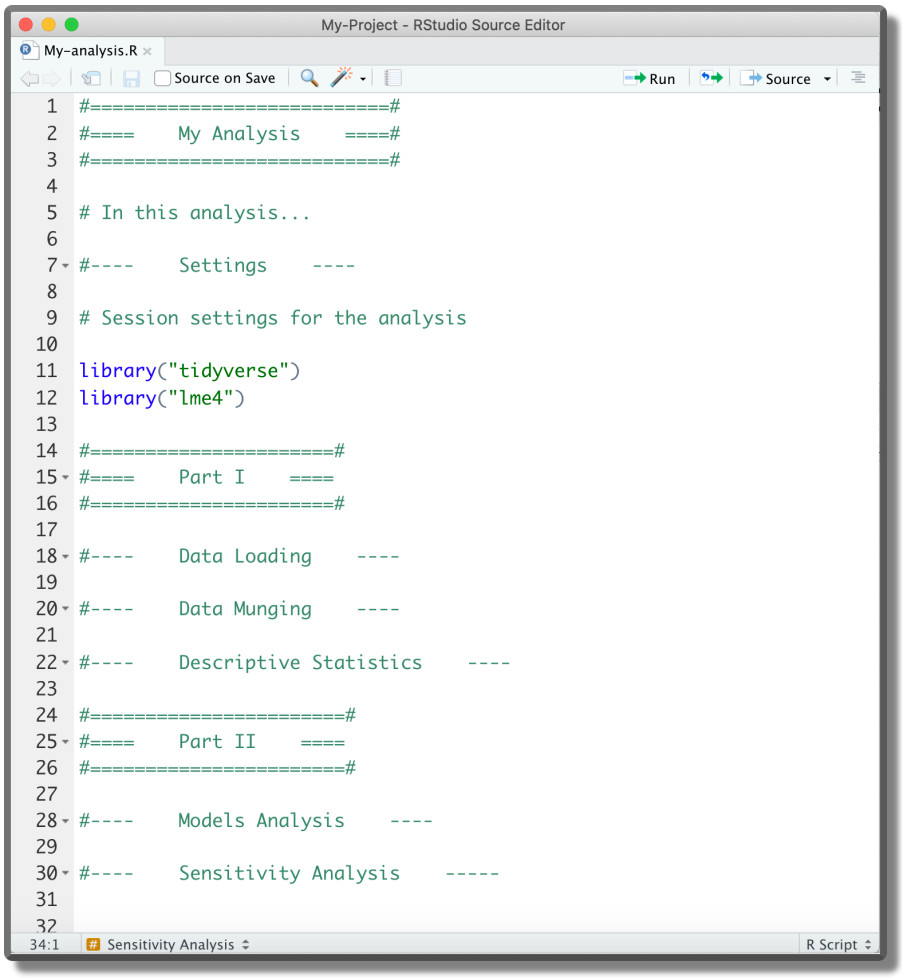
```
    |     |-- 04-comparison-experiments.R
    |     |-- 05-sensitivity-analysis.R
    |     |-- main.R
    |-- R/
    |     |-- data-munging.R
    |     |-- models.R
    |     |-- plots-tables.R
    |     |-- utils.R
```

9.2.1.1 Script Sections

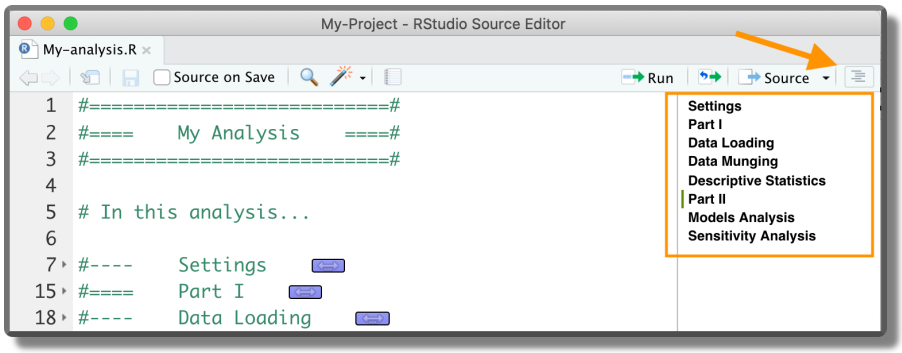
To enhance the readability of the analysis scripts, we can divide the code into sections. In RStudio, it is possible to create sections adding at the end of a comment line four (or more) consecutive symbols `####` (alternatively, `----` or `====`).

```
# Section 1 ####

# Section 2 ----

#----    Section 3    ----

####   Not Valid Section   --##
```

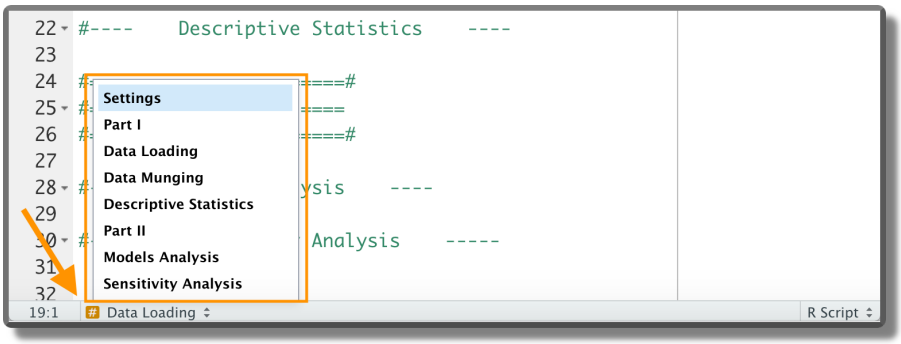
Using the available characters, it is possible to create different styles. The important thing is to finish the line with four (or more) identical symbols. As an example, we could organize our script as presented below



One of the advantages of organizing our script into sections is that at the top right corner we can find a navigation menu with the document outline. Section titles are given by the comment text.



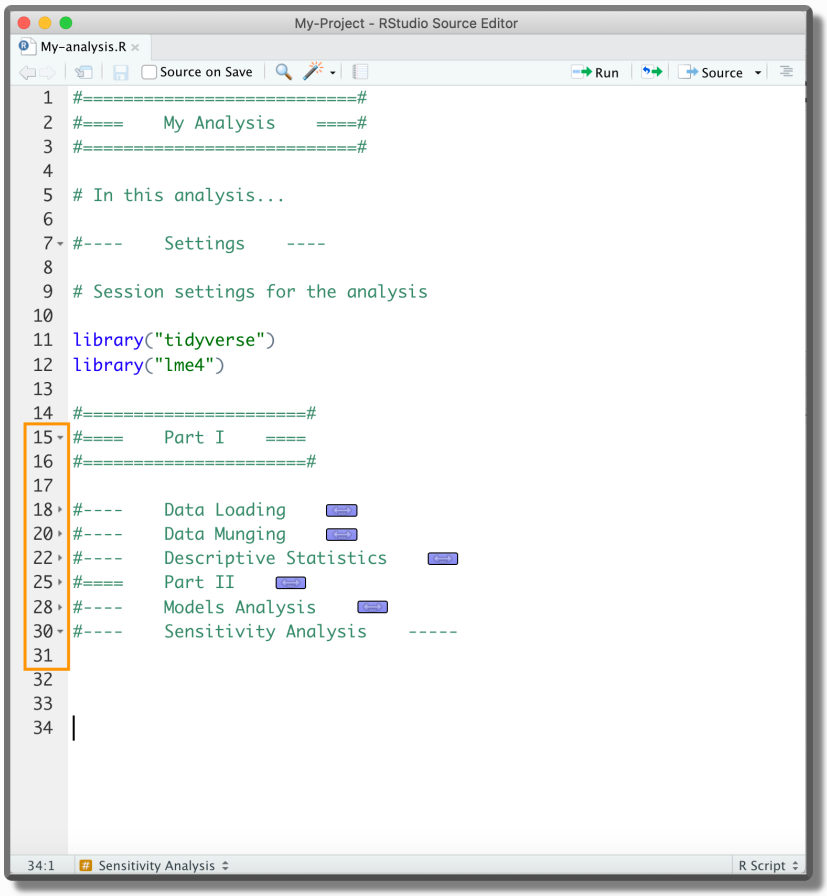
Another navigation bar is also available at the bottom left corner.



Dividing the code into sections enhances readability and helps us to navigate between the different analysis parts. However, we should avoid creating too long scripts as they are more difficult to maintain.

Trick-Box: Collapsing Sections

Note that next to the code line number, small arrows are now displayed. These arrows allow us to expand/collapse code sections.



9.2.1.2 Loading Functions

As we have defined our custom functions in separate scripts, before we can use them, we need to load them in our session environment. To do that we have two different solutions:

- **`source()`.** This function allows us to read code from R scripts, see `?source()` for options and details. Assuming all the required scripts are in the `R/` directory in the project root, we can use the following code lines to list all available scripts and source them:

```
# List all scripts in R/
script_list <- list.files("R", full.names = TRUE)

# Source scripts
invisible(sapply(script_list, source))
```

- **`devtools::load_all()`.** We briefly introduced the `devtools` R package in Chapter 5.2.2.2. This package provides many useful functions that facilitate our workflow when using the R-package project template (remember that the `DESCRIPTION` file is required). The function `devtools::load_all()` allows us to automatically source all scripts in the `R/` directory. See `?devtools::load_all()` for options and details. We can use `devtools::load_all()` in our analysis script specifying as argument the path to the project root where the `DESCRIPTION` file is present.

```
# Load functions
devtools::load_all(path = "<path-to-the-project-root>")
```

  The keyboard shortcut `Ctrl/Cmd + Shift + L` is also available. This is very handy during the analysis development as it facilitates the common workflow:

  1. *Write a new function*
  2. *Save the script*
  3. *Load all functions with* `Ctrl/Cmd + Shift + L`
  4. *keep working on the analysis*

We should include the code snippet used to load our custom functions at the beginning of the analysis scripts. Alternatively, we can include it in the `.Rprofile` to automatically load all functions at the beginning of each session. The latter approach, however, may lead to some problems. In particular, it limits the code readability as colleagues not familiar with the `.Rprofile` could not understand what is going on. Moreover, `.Rprofile` is not always automatically sourced when compiling dynamic documents. When compiling a document using the `Knit` button in Rstudio, a separate R session is launched using as working directory the document location. If this is not the project root (where the `.Rprofile` is located), the `.Rprofile` is not sourced.

Therefore, declaring the code snippet used to load our custom functions in the analysis scripts (or in the Rmarkdown file) following a more explicit approach is preferable (see *"Trick-Box: Using .Rprofile"* below for a good tip).

Trick-Box: Using .Rprofile

A good tip is to use the `.Rprofile` to run all commands and set options required to work on the analysis development. By doing this we can automate all those processes routinely done at the beginning of each session allowing us to jump straight into the development workflow without wasting time.

Common routine processes are loading our custom functions, loading required packages, and specifying preferred settings. For example, a possible `.Rprofile` may look like,

```
#----    .Rprofile    ----#

# Load custom functions
devtools::load_all()

# Load packages
library("tidyverse")
library("lme4")

# Settings ggplot
theme_set(theme_bw())
```

The actual analysis run, however, should rely only on the code declared explicitly in the analysis script. This would facilitate the analysis readability for colleagues not familiar with more advanced R features and avoid possible problems related to the creation of new R sessions (as in the case of dynamic documents compilation).

Note, however, that if we run the analysis script in our current session, the commands specified in the `.Rrofile` are still valid. To manage separately the session where we develop the analysis from the session where the analysis is run, we can evaluate whether the session is interactive or not. By specifying,

```
#----    .Rprofile    ----#

# Commands for interactive and non-interactive sessions
...

# Commands only for interactive sessions
if(interactive()){
  # Load custom functions
  devtools::load_all()

  # Load packages
  library("tidyverse")
  library("lme4")

  # Settings ggplot
  theme_set(theme_bw())
  }
```

commands defined inside the `if(interactive()){}` block are executed only in interactive sessions (usually when we develop the code). Note that commands defined outside the `if(interactive()){}` block will be always executed.

To run the analysis in a non-interactive session, we can run the script directly from the terminal (**not the R console!**) using the command,

```
$ Rscript <path-to/script.R>
```

For further details about running R in non-interactive mode, see https://github.com/gastonstat/tutorial-R-noninteractive. Note that using the `Knit` button in Rstudio (or using the `targets` workflow to run the analysis; see Section 9.3) automatically runs the code in a new, non-interactive session.

9.2.1.3 Loading R-packages

During our analysis, we will likely need to load several R packages. To do that, we can use different approaches:

- **Analysis Scripts.** We can declare the required packages directly at the beginning of the analysis scripts. The packages will be loaded when running the code.
- **`.Rprofile`.** Declaring the required packages in the `.Rprofile`, we can automatically load them at the beginning of each session.
- **`DESCRIPTION`.** When using the R-package project template, we can specify the required packages in the `DESCRIPTION` file. In particular, packages listed in the `Depends` field are automatically loaded at the beginning of each session. See https://r-pkgs.org/namespace.html?q=depends#imports for further details.

As in the case of loading custom functions (see Section 9.2.1.2), it is preferable to explicitly declare the required packages in the analysis scripts. This facilitates the analysis readability for colleagues not familiar with the `.Rprofile` and the `DESCRIPTION` file functioning. However, the `.Rprofile` (and the `DESCRIPTION` file) can still be used to facilitate the workflow during the analysis development (see *"Trick-Box: Using .Rprofile"* above).

Details-Box: Conflicts

Another aspect to take into account is the presence of **conflicts among packages**. Conflicts happen when two loaded packages have functions with the same name. In R, the default conflict resolution system is to give precedence to the

most recently loaded package. However, this makes it difficult to detect conflicts and can waste a lot of time debugging. To avoid package conflicts, we can:

- **`conflicted`.** The R package `conflicted` (Wickham, 2021) adopts a different approach making every conflict an error. This forces us to solve conflicts by explicitly defining the function to use. See https://conflicted.r-lib.org/ for more information.
- **`<package>::<function>`.** We can refer to a specific function by using the syntax `<package>::<function>`. In this case, we are no longer required to load the package with the `library("<package>")` command avoiding possible conflicts. This approach is particularly useful if only a few functions are required from a package. However, not that not loading the package prevents also package-specific classes and methods from being available. This aspect could lead to possible errors or unexpected results. See http://adv-r.had.co.nz/OO-essentials.html for more details on classes and methods.

Common conflicts to be aware of are:

- `dplyr::select()` vs `MASS::select()`
- `dplyr::filter()` vs `stats::filter()`

9.2.1.4 Reproducibility

Finally, let's discuss some details that may hinder result reproducibility:

- **Random Number Generator.** In R, using the function `set.seed()` we can specify the seed to allow reproducibility of random numbers generation. See `?set.seed()` for options and details. Ideally, we specify the seed at the beginning of the script used to run the analysis. Note that functions that call other software (e.g., `brms::brm()` or `rstan::stan()` which are based on Stan) may have their own `seed` argument that is independent of the seed in the R session. In this case, we need to specify both seeds to obtain reproducible results.
- **Global Options.** If our project relies on some specific global options (e.g, `stringsAsFactors` or `contrasts`), we should define them explicitly at the beginning of the script used to run the analysis. See `?options()` for further details. Note that we could also specify global options in the `.Rprofile` to facilitate our workflow during the analysis development (see *"Trick-Box: Using .Rprofile"* above).
- **Project Requirements.** To enhance results reproducibility, we should use the same R and R packages versions as in the original analysis. In Chapter 11.2, we discuss how to manage project requirements.

Tip-Box: Settings Section

We recommend creating a section *"Settings"* at the top of the main script where to collect the code used to define the setup of our analysis session. This includes:

- Loading required **packages**
- Setting required **session options**
- Defining the **random seed**
- Defining **global variables**
- Loading custom functions

9.2.2 Workflow Manager

In R, two main packages are used to create pipelines and manage the analysis workflow facilitating the project maintainability and enhancing result reproducibility. These packages are:

- `targets` (https://github.com/ropensci/targets). The `targets` package (Landau, 2022b) creates a Make-like pipeline. `targets` identifies dependencies between the analysis targets, skips targets that are already up to date, and runs only the necessary outdated targets. This package enables an optimal, maintainable and reproducible workflow.
- `workflowr` (https://github.com/workflowr/workflowr). The `workflowr` package (Blischak et al., 2021) organizes the project to enhance management, reproducibility, and sharing of analysis results. In particular, `workflowr` also allows us to create a website to document the results via GitHub Pages or GitLab Pages.

Between the two packages, `targets` serves more general purposes and has more advanced features. Therefore, it can be applied in many different scenarios. On the other hand, `workflowr` offers the interesting possibility of creating a website to document the results. However, we can create a website using other packages with lots more customizable options, such as `bookdown` (https://github.com/rstudio/bookdown), `blogdown` (https://github.com/rstudio/blogdown), or `pkgdown` (https://github.com/r-lib/pkgdown). Moreover, using `targets` does not exclude that we can also use `workflowr` to create the website. For more details see https://books.ropensci.org/targets/markdown.html.

In the next section, we discuss in more detail the `targets` workflow.

9.3 Targets

The `targets` package (successor of `drake`) creates a Make-like pipeline to enable an optimal, maintainable and reproducible workflow. Similar to Make, `targets` identifies dependencies between the analysis targets, skips targets that are already up to date, and runs only the necessary outdated targets. Moreover, `targets` support high-performance computing allowing us to run multiple tasks in parallel on our local machine or a computing cluster. Finally, `targets` also provide an efficient cache system to easily access all intermediate and final analysis results.

In the next sections, we introduce the `targets` workflow and its main features. This should be enough to get started, however, we highly encourage everyone to take a tour of `targets` official documentation available at https://books.ropensci.org/targets/. There are many more aspects to learn and solutions for possible issues.

9.3.1 Project Structure

To manage the analysis using `targets`, some specific files are required. As an example of a minimal workflow, consider the following project structure.

```
- my-project/
    |
    |-- _targets.R
    |-- _targets/
    |-- data/
    |     |-- raw-data.csv
    |-- R/
    |     |-- my-functions.R
    |     |-- ...
```

In line with the common project structure, we have a fictional data set (`Data/raw-data.csv`) to analyse,

ID	x	y
1	A	2.583
2	A	2.499
3	A	-0.625
...	...	...

and `R/My-functions.R` with our custom functions to run the analysis. In addition, we need a special R script `_targets.R` and a new directory `_targets/`.

9.3.1.1 The `_targets.R` Script

The `_targets.R` script is a special file used to define the workflow pipeline. By default, this file is in the root directory (however, we can indicate the path to the `_targets.R` script and also specify a different name; see Section 9.3.3.1 and `?tarconfig_set()` for special custom settings). In this case, the `_targets.R` script looks like,

```
#==========================#
#====    _targets.R    ====#
#==========================#

library("targets")

#----    Settings    ----

# Load packages
library("tidyverse")
library("lme4")

# Source custom functions
source("R/my-functions.R")

# Options
options(tidyverse.quiet = TRUE)

#----    Workflow    ----

list(
  # Get data
  tar_target(raw_data_file, "data/raw-data.csv", format = "file"),
  tar_target(my_data, get_my_data(raw_data_file)),

  # Descriptive statistics
  tar_target(plot_obs, get_plot_obs(my_data)),

  # Inferential statistics
  tar_target(lm_fit, get_lm_fit(my_data))
)
```

Let's discuss the different parts:

- **`library("targets")`.** It is **required** to load the `targets` R package itself at the beginning of the script (it is only required before the workflow definition, but it is common to specify it at the top).

- **Settings.** Next, we specify required R packages (also by `tar_option_set(packages = c("<packages>"))`; see https://books.ropensci.org/targets/packages.html), load custom functions, and set the required options.
- **Workflow.** Finally, we define inside a list each target of the workflow. In the example, we indicate the file with the raw data and we load the data in R. Next, we get a plot of the data and, finally, we fit a linear model.

9.3.1.2 Defining Targets

Each individual target is defined using the function `tar_target()` specifying the target name and the R expression used to compute the target. Note that all targets are collected within a list.

Specifying `format = "file"`, we indicate that the specified target is a dynamic file (i.e., an external file). In this way, `targets` tracks the file to evaluate whether it is changed. For more details see `?tar_target()`

Each target is an intermediate step of the workflow and their results are saved to be used later. `targets` automatically identifies the dependency relations between targets and updates single targets that are invalidated due to changes made. Ideally, each target should represent a meaningful step of the analysis. However, in case of changes to the code they depend on, large targets are required to be recomputed entirely even for small changes. Breaking down a large target into smaller ones allows skipping those parts that are not invalidated by changes.

9.3.1.3 The _targets/ Directory

`targets` stores the results and all files required to run the pipeline in the `_targets/` directory. In particular, inside we can find:

- **`meta/`.** It contains metadata regarding the targets, runtimes and processes.
- **`objects/`.** It contains all the targets results.
- **`users/`.** It is used to store custom files

This directory is automatically created the first time we run the pipeline. Therefore, we do not have to care about this directory as everything is managed by `targets`. Moreover, the entire `_targets/` directory should not be tracked by git. Only the file `_targets/meta/meta` is important. A `.gitignore` file is automatically added to track only relevant files.

Note that we can also specify a location other than `_targets/` where to store the data (see `?tarconfig_set()` for special custom settings).

9.3.2 The targets Workflow

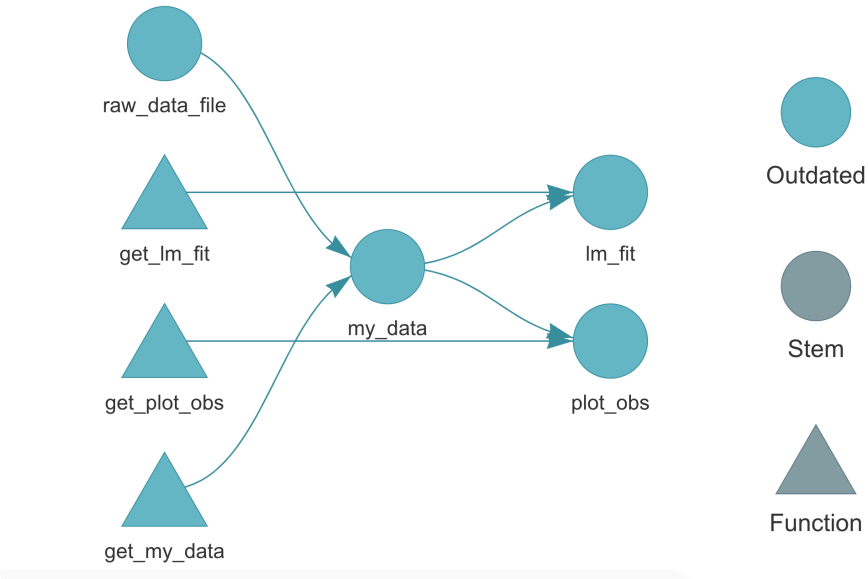
At this point, we have everything we need to run the analysis. Let's start the `targets` workflow.

9.3.2.1 Check the Pipeline

Before running the pipeline, we can inspect it to evaluate the possible presence of errors. Using the function `targets::tar_manifest()`, we obtain a data frame with all the targets and information about them. Note that targets are ordered according to their topological order (i.e., the expected order of execution without considering parallelization and priorities). See `?targets::tar_manifest()` for further details and options.

```
tar_manifest(fields = "command")
## # A tibble: 4 x 2
##   name          command
##   <chr>         <chr>
## 1 raw_data_file "\"data/raw-data.csv\""
## 2 my_data       "get_my_data(raw_data_file)"
## 3 plot_obs      "get_plot_obs(data = my_data)"
## 4 lm_fit        "get_lm_fit(data = my_data)"
```

We can also use the function `targets::tar_visnetwork()`, to visualize the pipeline and the dependency relationship between targets. The actual graph we obtain is made by the `visNetwork` package (we need to install it separately) and it is interactive (try it in RStudio). See `?targets::tar_visnetwork()` for further details and options. At the moment, all our targets are outdated.

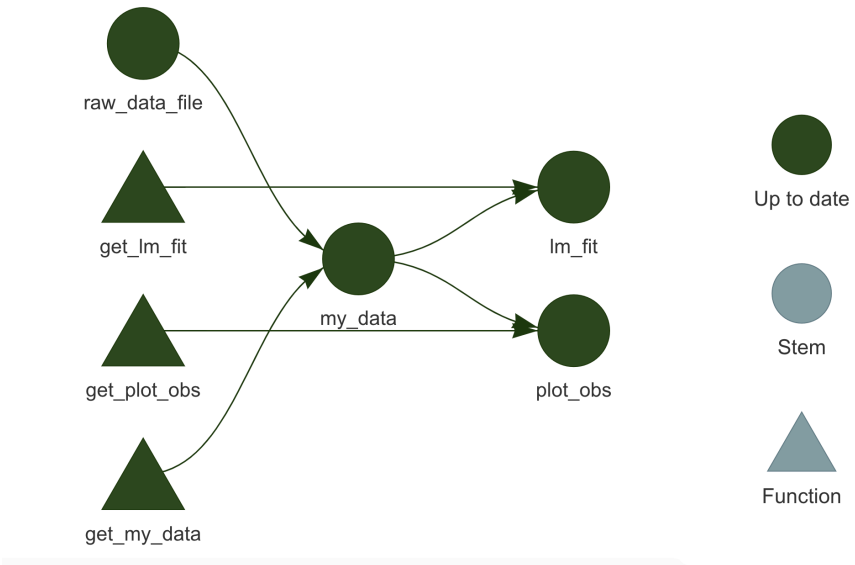


9.3.2.2 Run the Pipeline

Using the function `targets::tar_make()`, we can run the pipeline. All targets are evaluated in a new external session in the correct order and results are saved in the `_targets/` directory. See `?targets::tar_make()` for further details and options.

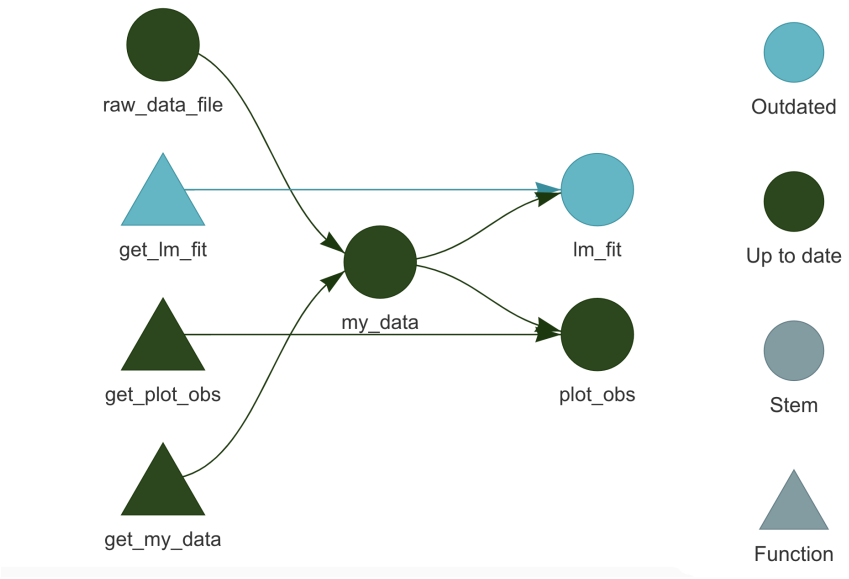
```
tar_make()
## * start target raw_data_file
## * built target raw_data_file
## * start target my_data
## * built target my_data
## * start target plot_obs
## * built target plot_obs
## * start target lm_fit
## * built target lm_fit
## * end pipeline
```

If we look again at `targets::tar_visnetwork()` graph, we can see that now all targets are up to date.



9.3.2.3 Make Changes

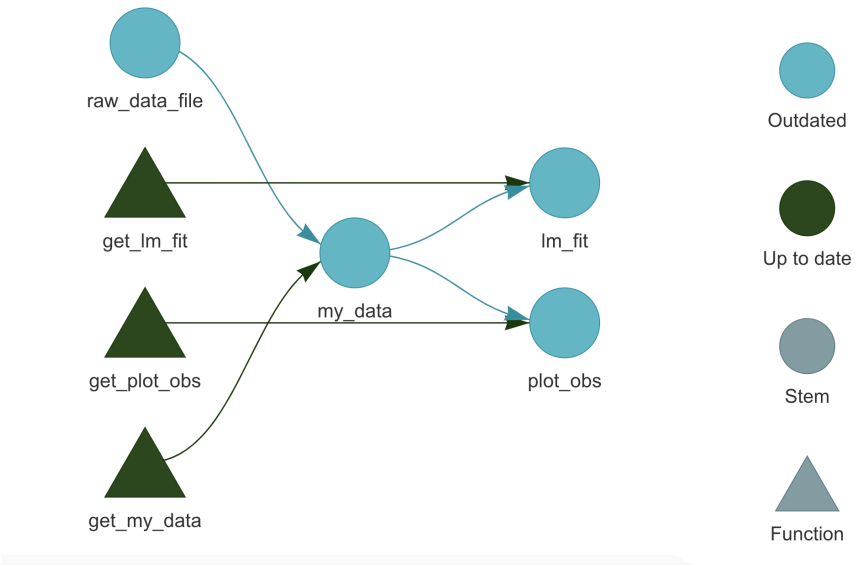
Let's say we make some changes to the function used to fit the linear model. `targets` will notice that and it will identify the invalidated targets that require to be updated. Looking at the `targets::tar_visnetwork()` graph, we can see which targets are affected by the changes made.



Running `targets::tar_make()` a second time, we see that up-to-date targets are skipped and only outdated targets are computed again, potentially saving us a lot of time.

```
tar_make()
## v skip target raw_data_file
## v skip target my_data
## v skip target plot_obs
## * start target lm_fit
## * built target lm_fit
## * end pipeline
```

Suppose, instead, that we made some changes to the raw data (e.g., adding new observations). `targets` will detect that as well and in this case, the whole pipeline will be invalidated.



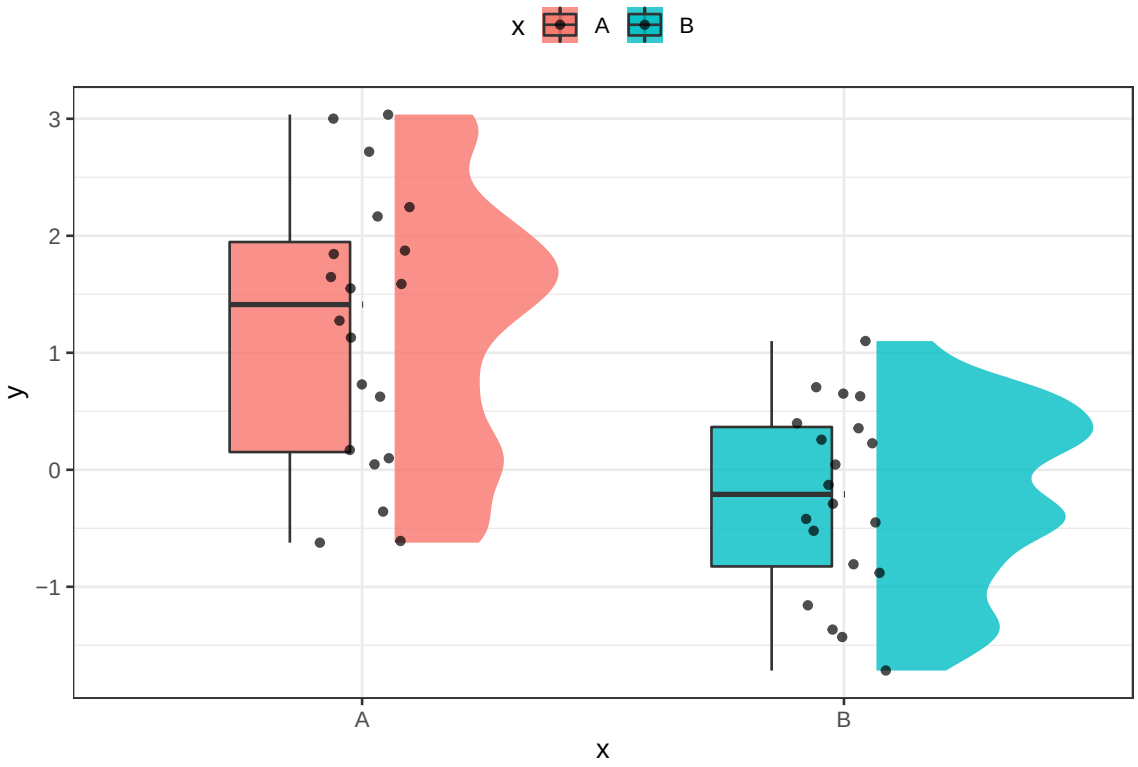
9.3.2.4 Get the Results

To access the targets results, we have two functions:

- **`targets::tar_read().`** Read the target from the `_targets/` directory and return its value.
- **`targets::tar_load().`** Load the target directly into the current environment (`NULL` is returned).

For example, we can use `targets::tar_read()` to obtain the created plot,

```
targets::tar_read(plot_obs)
```



or we can use `targets::tar_load()` to load a target in the current environment so we can use it subsequently with other functions.

```
targets::tar_load(lm_fit)
summary(lm_fit)
##
## Call:
## lm(formula = y ~ x, data = data)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1.83060 -0.70935  0.08828  0.64521  1.82840
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   1.2076     0.2225   5.428 3.47e-06 ***
## xB           -1.4477     0.3146  -4.601 4.58e-05 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.995 on 38 degrees of freedom
```

```
## Multiple R-squared:  0.3578, Adjusted R-squared:  0.3409
## F-statistic: 21.17 on 1 and 38 DF,  p-value: 4.575e-05
```

Again, we stress the difference between the two functions: `tar_read()` returns the target's value, whereas `tar_load()` loads the target in the current environment. Therefore, to subsequently use the target, we need to assign its value when using `tar_read()` or simply use the target after loading it with `tar_load()`. For example,

```
# Assign the target's value for later use
obs <- targets::tar_read(my_data)
head(obs)
##   ID x      y
## 1  1 A  1.274
## 2  2 A  0.169
## 3  3 A  2.718
## 4  4 A  1.588
## 5  5 A  3.001
## 6  6 A -0.609

# my_data is not available
my_data
## Error in eval(expr, envir, enclos): object 'my_data' not found

# Load target in the current environment
targets::tar_load(my_data)
head(my_data)
##   ID x      y
## 1  1 A  1.274
## 2  2 A  0.169
## 3  3 A  2.718
## 4  4 A  1.588
## 5  5 A  3.001
## 6  6 A -0.609
```

9.3.3 Advanced Features

Now we discuss some other more advanced features of `targets`. Again, `targets` is a complex package with many features and options to account for any needs. Therefore, we highly encourage everyone to take a tour of `targets` official documentation available at https://books.ropensci.org/targets/. There are many more aspects to learn and solutions for possible issues.

9.3.3.1 Project Structure

Let's see how we can optimize our project organization when using the `targets` workflow. A possible solution is to collect all directories and files related to `targets` in the `analysis/` directory.

```
- my-project/
    |
    |-- _targets.yaml
    |-- analysis/
    |     |-- targets-workflow.R
    |     |-- targets-analysis.R
    |     |-- _targets/
    |-- data/
    |     |-- raw-data.csv
    |-- R/
    |     |-- my-functions.R
    |     |-- ...
```

In particular, we have:

- **`analysis/targets-workflow.R`.** The R script with the definition of the workflow pipeline. This is the same as the `_targets.R` scripts described in Section 9.3.1.1.
- **`analysis/_targets/`.** The directory where `targets` stores all the results and pipeline information. See Section 9.3.1.3.
- **`analysis/targets-analysis.R`.** In this script we can collect all the functions required to manage and run the workflow.

  As we are no longer using the default `targets` project structure, it is required to modify `targets` global settings by specifying the path to the R script with the workflow pipeline (i.e., `analysis/targets-workflow.R`) and the path to the storing directory (i.e., `analysis/_targets/`). To do that, we can use the function `targets::tar_config_set()` (see the help page for more details). In our case, the `targets-analysis.R` script looks like this

```r
#================================#
#====    Targets Analysis    ====#
#================================#

# Targets settings
targets::tar_config_set(script = "analysis/targets-workflow.R",
                        store = "analysis/_targets/")
```

```
#----    Analysis    ----

# Check workflow
targets::tar_manifest(fields = "command")
targets::tar_visnetwork()

# Run analysis
targets::tar_make()

# End
targets::tar_visnetwork()


#----    Results    ----

# Aim of the study is ...
targets::tar_load(my_data)

# Descriptive statistics
summary(data)
targets::tar_read(plot_obs)

# Inferential statistics
targets::tar_load(lm_fit)
summary(lm_fit)
...
#----
```

After the code used to run the analysis, we can also include a section where results are loaded and briefly presented. This will allow colleagues to explore analysis results immediately. Note that appropriate documentation is required to facilitate results interpretation.

- **`_targets.yaml`.** A YAML file with the custom `targets` settings. This file is automatically created when `targets` global settings are modified using the `targets::tar_config_set()` function (see help page for more information about custom settings). In our case, the `_targets.yaml` file looks like this

```
#----    _targets.yaml    ----#
main:
  script: analysis/targets-workflow.R
  store: analysis/_targets/
```

9.3.3.2 `targets` and R Markdown

To integrate the `targets` workflow with dynamic documents created by R Markdown, there are two main approaches

- **R Markdown as Primary Script.** The R Markdown document is used as the primary script to manage the `targets` workflow. Following this approach, the whole pipeline is defined and managed within one or more R Markdown documents. To learn how to implement this approach, see https://books.ropensci.org/targets/markdown.html.
- **R Markdown as Target.** The R Markdown document is considered as a new target in the pipeline. Following this approach, the whole pipeline is defined and managed outside of the R Markdown document. R Markdown documents should be lightweight with minimal code used simply to present the targets' results retrieved with `targets::tar_read()` or `targets::tar_load()`. Targets should not be computed within the R Markdown documents. To learn how to implement this approach, see https://books.ropensci.org/targets/files.html#literate-programming.

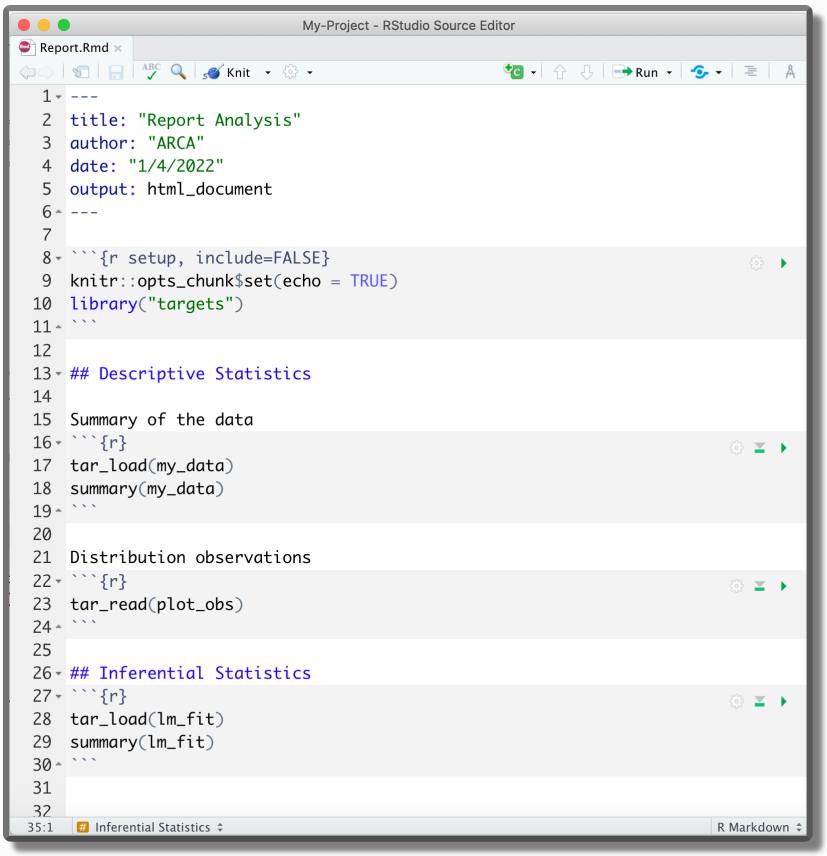
Among the two approaches, we recommend the second one. Using R Markdown documents as primary scripts to manage the analysis is fine in the case of simple projects. However, in the case of more complex projects, it is better to keep the actual analysis and the presentation of the results separate. In this way, the project can be maintained and developed more easily.

Details-Box: R Markdown as Target

Following this approach, the `target` workflow is defined and managed following the usual approach and R Markdown documents are considered as targets in the pipeline.

To add an R Markdown document to the pipeline, we have to define the target using `tarchetypes::tar_render()` instead of the common `targets::tar_target()` function. Note that we need to install the `tarchetypes` package (Landau, 2022a). See the help page for function arguments and details.

Suppose we created the following report saved as `documents/report.Rmd`.

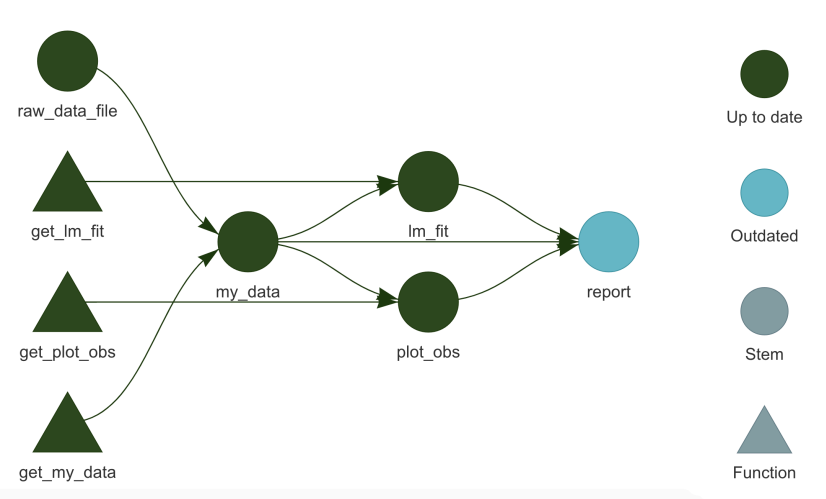


Next, we add it to the workflow pipeline,

```
#----    Targets-workflow.R    ----#
...

list(
  ...

  # Report
  tarchetypes::tar_render(report, "documents/report.Rmd"),

  ...
)
```

Now, `targets` will automatically recognize the dependencies our report is based on and will add the report to the workflow pipeline. Note that to

allow dependency identification, targets have to be explicitly retrieved with `targets::tar_read()` or `targets::tar_load()`.



Running `targets::tar_make()`, we can update the pipeline compiling the report as well.

Alternatively, we can also compile the report outside the pipeline as usual by clicking the `Knit` button in RStudio. However, unless the report is in the root directory, we need to specify the position of the directory `_targets/` (i.e., the directory with all the targets' results and information) relative to the report position. To do that do not use the `targets::tar_config_set()` function as this would overwrite global settings for the whole `targets` workflow. Instead, create manually a `_targets.yaml` file in the same directory as the report specifying the store location. Considering the report in the example above, we would define

```
#----    documents/_targets.yaml    ----#
main:
  store: ../analysis/_targets/
```

9.3.3.3 Reproducibility

`targets` enhance the reproducibility of the results by automatically running the pipeline in a reproducible background process. This procedure avoids that our current environment or other temporary settings affect the results.

Let's discuss some other details relevant for results reproducibility:

- **Random Number Generator.** When running the pipeline, each target is built

using a unique seed determined by the target's name (no two targets share the same seed). In this way, each target runs with a reproducible seed and we always obtain the same results. See `targets::tar_meta()` for a list of targets' metadata including each target specific seed. See function documentation for further details.

- **Global Settings.** Global settings are usually defined explicitly in the script used to specify the workflow pipeline (see Section 9.3.1.1). However, commands defined in the `.Rprofile` are also evaluated when running the pipeline. This is not a problem for reproducibility but it may limit the code understanding of colleagues not familiar with more advanced features of R. To overcome this issue, note that `targets` runs the analysis in a non-interactive session. Therefore, we can avoid that the `.Rprofile` code is evaluated following the suggestion described in *"Trick-Box: Using .Rprofile"*.
- **Project Requirements.** `targets` does not track changes in the R or R-packages versions. To enhance reproducibility, it is good practice to use the `renv` package for package management (see Chapter 11.3). `targets` and `renv` can be used together in the same project workflow without problems.

9.3.3.4 Branching

A very interesting feature of `targets` is branching. When defining the analysis pipeline, many targets may be obtained iteratively from very similar tasks. If we are already used to functional style, we will always aim to write concise code without repetitions. Here is where branching comes into play as it allows to define multiple targets concisely.

Conceptually branching is similar to the `purr::map()` function used to apply the same code over multiple elements. In `targets` there are two types of branching:

- **Dynamic Branching.** The new targets are defined dynamically while the pipeline is running. The number of new targets is not necessarily known in advance. Dynamic branching is better suited for creating a large number of very similar targets. For further details on dynamic branching, see https://books.ropensci.org/targets/dynamic.html.
- **Static Branching.** The new targets are defined in bulk before the pipeline starts. The exact number of new targets is known in advance and they can be visualized with `targets::tar_visnetwork()`. Static branching is better suited for creating a small number of heterogeneous targets. For further details on static branching, see https://books.ropensci.org/targets/static.html.

Branching increases a little bit the pipeline complexity as it has its own specific code syntax. However, branching allows obtaining a more concise and easier to maintain and read pipeline (once familiar with the syntax).

9.3.3.5 High-Performance Computing

`targets` supports high-performance computing allowing us to run multiple tasks in parallel on our local machine or a computing cluster. To do that, `targets` integrates in

its workflow the `Clustermq` (https://mschubert.github.io/clustermq) and the `future` (https://future.futureverse.org/) R packages.

In the case of large, computationally expensive projects, we can obtain valuable gains in performance by parallelizing our code execution. However, configuration details and instructions on how to integrate high-performance computing in the `targets` workflow are beyond the aim of this chapter. For further details on high-performance computing, see https://books.ropensci.org/targets/hpc.html.

9.3.3.6 Load All Targets

We have seen how targets' results can be retrieved with `targets::tar_read()` or `targets::tar_load()`. However, it may be useful to have a function that allows us to load all required targets at once. To do that, we can define the following functions in a script named `R/targets-utils.R`.

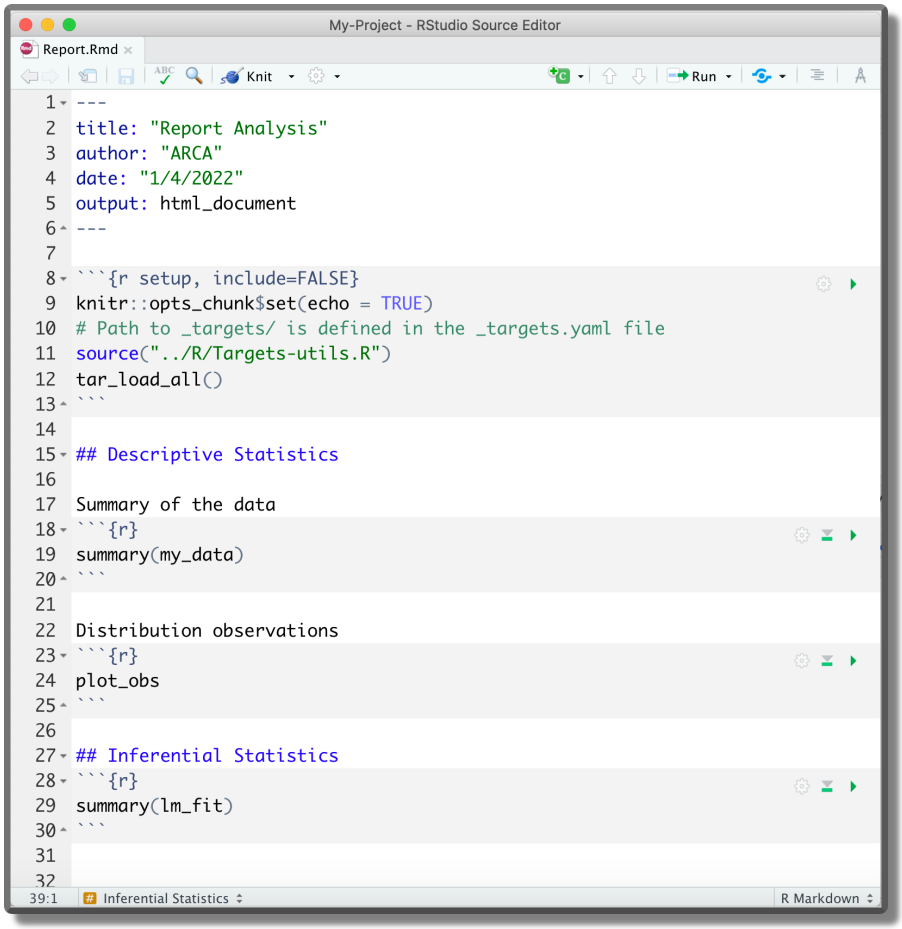
```
#----    R/Targets-utils.R    ----#

#----    load_glob_env    ----

# Load targets in the global environment
load_glob_env <- function(..., store = targets::tar_config_get("store")){
  targets::tar_load(..., envir = globalenv(), store = store)
}

#----    tar_load_all    ----

# Load listed targets
tar_load_all <- function(store = targets::tar_config_get("store")){
  targets <- c("my_data", "lm_fit", "plot_obs", "<other-targets>", "...")

  # load
  sapply(targets, load_glob_env, store = store)

  return(cat("Tartgets loaded!\n"))
}
```

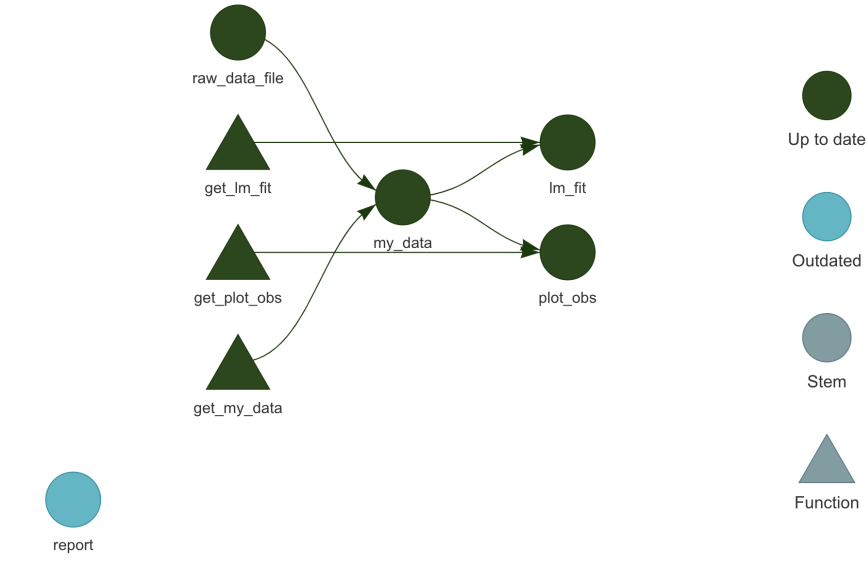
Where:

- **`load_glob_env()`** is used to load the targets directly in the global environment (otherwise targets would be loaded only in the function environment and we could not use them).

- **`tar_load_all()`** is used to create a list of the targets of interest and subsequently load them into the global environment.

Now we can use the function `tar_load_all()` to directly load all specified targets. Note, however, that loading the targets in this way in an RMarkdown document would not allow `targets` to detect dependencies correctly.





Trick-Box: Load Targets Using .Rprofile

We could automatically load targets into the environment by including the function `tar_load_all()` in the `.Rprofile`.

```
#----    .Rprofile    ----#

...

# Commands only for interactive sessions
if(interactive()){

    ...

    # Load custom function
    source("R/targets-utils.R")

    # alternatively devtools::load_all()

    # Load targets
    tar_load_all()

    ...
  }
```

In this way, each time we restart the R session all targets are loaded in the environment and we can go back straight into the analysis development.

 Documentation-Box

Make

- make official documentation
  https://www.gnu.org/software/make/.
- NMake
  https://docs.microsoft.com/en-us/cpp/build/reference/nmake-reference
- makefile
  https://opensource.com/article/18/8/what-how-makefile

R

- Run R non-interactive
  https://github.com/gastonstat/tutorial-R-noninteractive
- DESCRIPTION load packages
  https://r-pkgs.org/namespace.html?q=depends#imports
- `conflicted` R package
  https://conflicted.r-lib.org/
- Object Oriented
  http://adv-r.had.co.nz/OO-essentials.html
- workflowr R package
  https://github.com/workflowr/workflowr

Targets

- Official documentation
  https://books.ropensci.org/targets/
- Load packages
  https://books.ropensci.org/targets/packages.html
- Literate programming
  https://books.ropensci.org/targets/files.html#literate-programming
- Dynamic branching
  https://books.ropensci.org/targets/dynamic.html
- Static branching
  https://books.ropensci.org/targets/static.html
- High-performance computing
  https://books.ropensci.org/targets/hpc.html

10 Dynamic Documents

[Work in Progress]

10.1 Quarto

https://quarto.org/

10.2 `trackdown`

One of the authors of this book is also the maintainer of the `trackdown` R package. `trackdown` (Kothe et al., 2021) offers a simple solution for collaborative writing and editing of R Markdown (or Sweave) documents. Using `trackdown`, we can upload the local `.Rmd` (or `.Rnw`) file as a plain-text file to Google Drive.

By taking advantage of the easily readable Markdown (or LATEX) syntax and the well-known online interface offered by Google Docs, collaborators can easily contribute to the writing and editing process. After integrating all authors' contributions, we can download the final document and render it locally.

The package documentation is available at https://claudiozandonella.github.io/trackdown/.

10.2.1 The `trackdown` Workflow

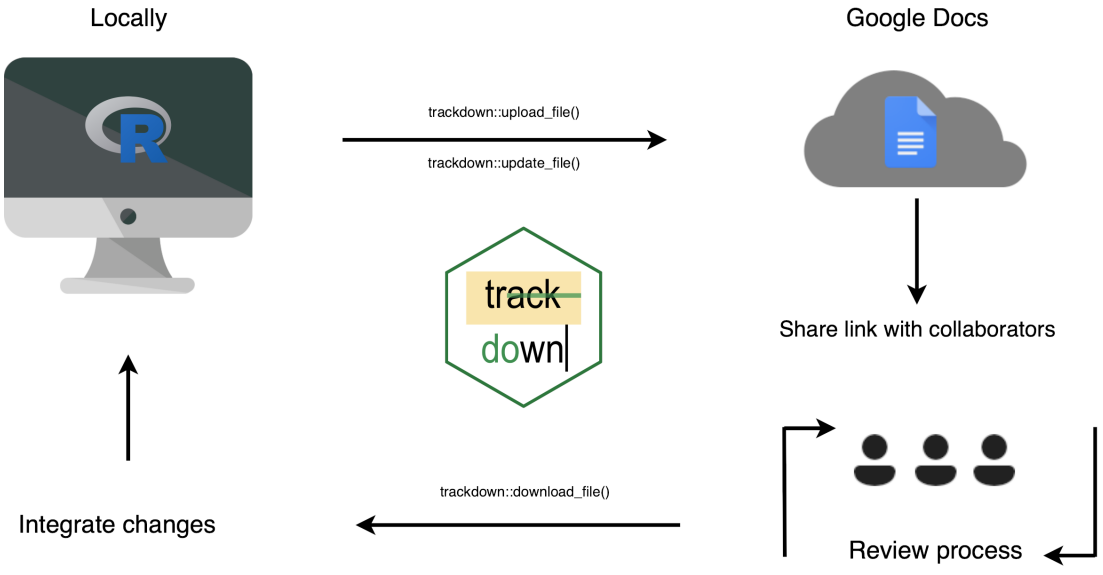
During the collaborative writing/editing of an `.Rmd` (or `.Rnw`) document, it is important to employ different workflows for computer code and narrative text:

- **Code** - Collaborative code writing is done most efficiently by following a traditional **Git**-based workflow using an online repository (e.g., GitHub or GitLab; see Chapter 8).
- **Narrative Text** - Collaborative writing of the narrative text is done most efficiently using **Google Docs** which provides a familiar and simple online interface that allows multiple users to simultaneously write/edit the same document.

Thus, the workflow's main idea is simple: Upload the `.Rmd` (or `.Rnw`) document to Google Drive to collaboratively write/edit the narrative text in Google Docs; download the document locally to continue working on the code while harnessing the power of Git for version control and collaboration. This iterative process of uploading to and downloading from Google Drive continues until the desired results are obtained. The workflow can be summarized as:

> Collaborative **code** writing using **Git** & collaborative writing of **narrative text** using **Google Docs**

For a detailed example of the workflow see https://ekothe.github.io/trackdown/articles/trackdown-workflow.html.



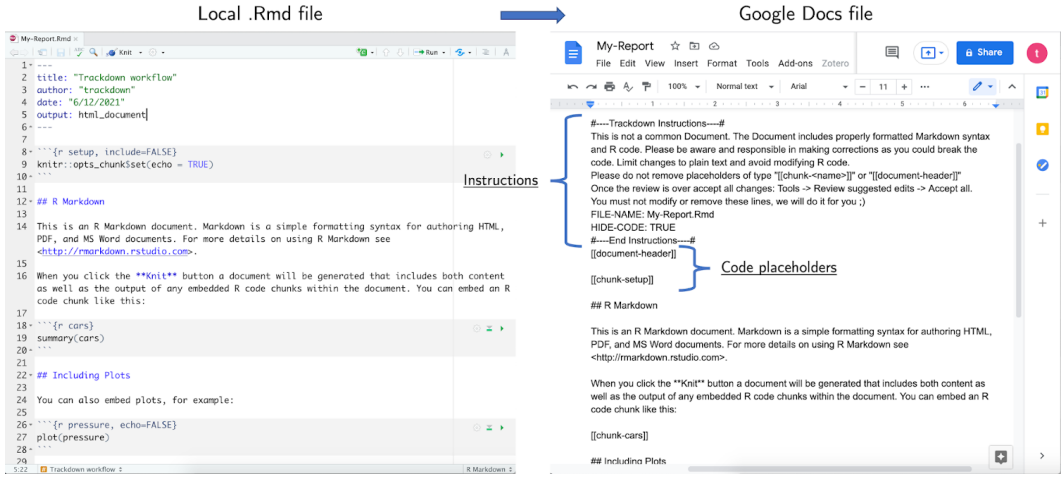
Functions and Special Features

`trackdown` offers different functions to manage the workflow:

- `upload_file()` uploads a file for the first time to Google Drive.
- `update_file()` updates the content of an existing file in Google Drive with the contents of a local file.
- `download_file()` downloads the edited version of a file from Google Drive and updates the local version.
- `render_file()` downloads a file from Google Drive and renders it locally.

Moreover, `trackdown` offers additional features to facilitate the collaborative writing and editing of documents in Google Docs. In particular, it is possible to:

- **Hide Code:** Code in the header of the document (YAML header or LaTeX preamble) and code chunks are removed from the document when uploading to Google Drive and are automatically restored during download. This prevents collaborators from inadvertently making changes to the code which might corrupt the file and allows them to focus on the narrative text.



- **Upload Output:** The actual output document (i.e., the rendered file) can be uploaded to Google Drive in conjunction with the `.Rmd` (or `.Rnw`) document. This helps collaborators to evaluate the overall layout including figures and tables and allows them to add comments to suggest and discuss changes.

- **Use Google Drive shared drives:** The documents can be uploaded to either a personal Google Drive or to a shared drive to facilitate collaboration.

10.2.1.1 Advantages of Google Docs

Google Docs offers users a familiar, intuitive, and free web-based interface that allows multiple users to simultaneously write/edit the same document. In Google Docs it is possible to:

- track changes (incl. accepting/rejecting suggestions)
- add comments to suggest and discuss changes
- check spelling and grammar errors (potentially integrating third-party services like Grammarly)

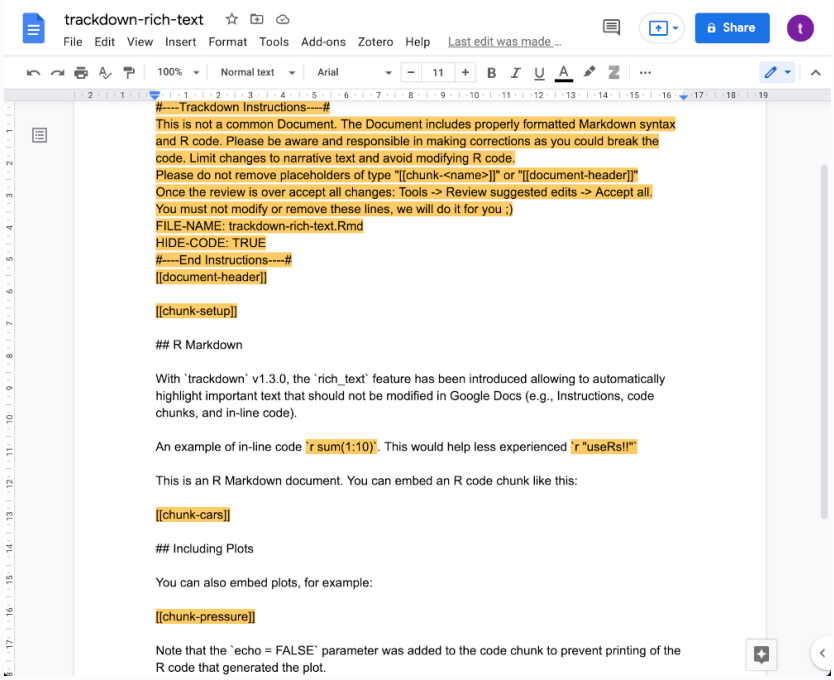
Moreover, Google Docs allows anyone to contribute to the writing/editing of the document. No programming experience is required, users can just focus on writing/editing the narrative text.

Note that not all collaborators have to have a Google account (although this is recommended to utilize all Google Docs features). Only the person who manages the `trackdown` workflow needs to have a Google account to upload files to Google Drive. Other collaborators can be invited to contribute to the document using a shared link.

Details-Box: trackdown Development Version

The developmental version of `trackdown` (v1.3.0, currently only available on GitHub), introduced a new important feature:

- **`rich_text`.** Upload rich documents to Google Docs where important text that should not be changed is automatically highlighted (e.g., placeholders hiding the code, header of the document, code chunks, and in-line code). This prevents collaborators from inadvertently making changes to the code which might corrupt the file. See rich-text feature details.



To install the developmental version of `trackdown` form GitHub (https://github.com/claudiozandonella/trackdown),

```
# install.packages(remotes)

# install the development version
remotes::install_github("claudiozandonella/trackdown",
                        build_vignettes = TRUE)
```

Very soon, we will also add support for Quarto documents as well, see https://github.com/ClaudioZandonella/trackdown/issues/33

11 Requirements

In the previous chapters, we learned to organize all our files and data in a well structured and documented repository. Moreover, we learned to use Git and GitHub for tracking changes and managing collaboration during the development of our project. Finally, we introduced dedicated tools for managing the analysis workflow pipeline and for creating dynamic documents.

To guarantee the reproducibility of the results, however, it is not enough to automatize the analysis execution, but we also need to satisfy all project requirements. Ideally, the analysis should be reproduced using the same settings and configuration as in the original analysis. This means that system prerequisites, specific software, and required libraries all should match the specific project requirements.

In this chapter, we discuss how to manage our project requirements. In Chapter 12, instead, we introduce Docker and the container technology that allows us to create and share an isolated, controlled, standardized environment for our project. The Holy Grail of reproducibility.

11.1 Project Requirements and Installation

We need to provide a clear description of the project requirements and how to install them. As described in Chapter 3.1.1.5, we can create a section in the project `README` named *"Requirements"*, with all the project requirements, and another section named *"Installation"*, with step-by-step instructions on how to get our project ready on someone else's machine.

11.1.1 Requirements

Let's discuss project requirements. Of course, these are just general indications that have to be adapted according to the specific project aims and needs.

11.1.1.1 System Prerequisites

A project may require some specific system prerequisites. For example, a minimum GPU memory may be required for computationally intensive processes or the project could be based on a specific Operating System. In this case, it is important to list all the system requirements.

System prerequisites are common in the case of applications or software. In most research projects, however, there are no specific system prerequisites. Nevertheless, it could be useful to list the system characteristics (e.g., Operating System and CPU details) of the machine used to obtain the original results. Of course, in this case, it should be highlighted that these are not requirements but only the machine characteristics used in the original analysis. This information may be useful in the case of inexplicable reproducibility issues.

11.1.1.2 Software

The most important thing, surely, is to list all the software required for the project. These can be our favourite programming language for statistical analysis (e.g., R, Python, or Julia), software to create documents (e.g., LaTeXor Pandoc), specific compilers (e.g., Clang or GCC) or other software for any particular needs.

Note that it is not enough to indicate only the name of the software, but is also necessary to specify the exact software version. Most software is actively developed and new versions are regularly released. Usually, new versions bring some new features or some bugs are fixed. In the case of major releases, however, changes are so important that backward compatibility is no longer guaranteed. This means that our old code may no longer work using the new software versions. Specifying the required software version allows for avoiding these issues.

Details-Box: Versioning

Software version numbers usually have three components `major.minor.patch` (e.g. R version 4.1.2). New version number is decided according to the actual changes made:

- **Patch**. When some bugs are fixed but no new feature is added, the `patch` value is incremented (e.g. R version 4.1.3).
- **Minor Release**. When a new feature is added (with or without bug fixes), the `minor` value is incremented (e.g. R version 4.2.0).

- **Major Release**. When changes affect backward compatibility, the `major` value is incremented (e.g. R version 5.0.0).

Note that, in the case of a minor or a major release, all previous values are set to zero. For example, if the current version is `2.1.11`, with a new minor release we obtain `2.2.0` and with a new major release `3.0.0`.

In the case of a development version the `.9000` arbitrary suffix is added (e.g., `2.1.11.9000`).

11.1.1.3 Libraries

Finally, most of the currently popular programming languages for statistical analysis rely on multiple libraries to implement several features. Also, these libraries are regularly updated and their backward compatibility may be compromised. Therefore, it is necessary to list all the required libraries with their exact versions.

The number of required libraries can grow quite large very easily. For this reason, listing all of them in the `README` file may not be the best solution. Alternatively, we could list all the libraries in another file in our project. For example, we can create a `requirements.txt` file with each library name and version. Ideally, this file should be structured smartly to allow other colleagues to automatically retrieve the required information to install the libraries using some function. For example, the `requirements.txt` file can be structured as follows.

```
library_1==x.xx.xx
library_2==x.xx.xx
...
```

In this way, using regular expressions (for a tutorial on regular expressions see https://ryanstutorials.net/regular-expressions-tutorial/) library name and version can be easily retrieved and used to install all the dependencies.

In the `README` file, we should still point to the `requirements.txt` file providing relevant indications on how to use it and how to install the required libraries. We could also provide a custom function to automate this process. However, major programming languages already provide ad-hoc solutions to manage project dependencies easily and automatically. In Section 11.2, we describe how to manage dependencies in R (for Python, see https://pip.pypa.io/en/stable/user_guide/#requirements-files and https://www.geeksforgeeks.org/pipenv-python-package-management-tool).

11.1.2 Installation

Most projects do not need special installation procedures. We simply need to make sure we have the right version of the software for the statistical analyses. Next, we download the project repository, install all the required libraries and we are ready to run the analysis.

This is a very common workflow that requires a minimal set of instructions. We can reasonably assume that other colleagues already know how to install popular statistical software. Therefore, we usually specify only how to download the project and how to install the required libraries.

In other cases, however, our project may need unusual software or require specific installation options and configuration settings (e.g., exporting environment variables). In these cases, a detailed step-by-step description of the installation procedure is needed. Remember, we should always provide enough information to allow other colleagues to install our project and get ready for the analysis.

Note that installation instructions should also take into account possible differences due to different operating systems and how to solve possible common issues. In this regard, links to external documentation (e.g., official software documentation or troubleshooting help pages) could be very helpful. Keep in mind that the issues we have found when developing our project are likely to be the same issues other colleagues will stumble on.

For an example of complex install documentation, see Jekyll documentation on GitHub (https://docs.github.com/en/pages/setting-up-a-github-pages-site-with-jekyll/testing-your-github-pages-site-locally-with-jekyll) or RStan documentation (https://github.com/stan-dev/rstan/wiki/RStan-Getting-Started).

Trick-Box: install.sh script

An alternative approach is to create a bash script (e.g., `install.sh`) to automate the whole installation procedure or some part of it. We introduced the use of the terminal and bash command language in Chapter 6.

Doing this will require a lot of hard work. We have to deal with lots of possible complications. However, it will allow other colleagues to install our project simply by running the `install.sh` script. A very elegant solution.

11.2 R Dependencies

In this section, we focus on the usual requirements to take into account when using R. We start considering the R version and related installed software. Next, we discuss the R package ecosystem.

11.2.1 R version

The first thing to take into account is the actual R version used in the project. This is particularly relevant as, on the 24th of April 2020 (curiously, the birthday of one of the book authors), R 4.0.0 was released introducing several new features that were not available before. Moreover, as for any major release, backward compatibility is not guaranteed. Thus, guess what?...Code working on the old 3.6.x versions may fail (or give

surprisingly different results) if run on the new 4.x.x. versions. For these reasons, it is important to use the exact required R version.

We can check our current R version (plus other system info) by running the command `version` (without parenthesis, it is an object, not a function; see `?version` for further details).

```
version
##                _
## platform       x86_64-apple-darwin17.0
## arch           x86_64
## os             darwin17.0
## system         x86_64, darwin17.0
## status
## major          4
## minor          1.2
## year           2021
## month          11
## day            01
## svn rev        81115
## language       R
## version.string R version 4.1.2 (2021-11-01)
## nickname       Bird Hippie
```

Details-Box: R Release Updates

Info-Box: R Release Updates To know the new feature introduced in R 4.0.0 see https://www.r-bloggers.com/2020/04/r-4-0-0-now-available-and-a-look-back-at-rs-history/

To know the new feature introduced in R 4.1.0 see https://www.r-bloggers.com/2021/05/new-features-in-r-4-1-0/

For all the other R release news updates see https://cran.r-project.org/doc/manuals/r-release/NEWS.html

11.2.1.1 Multiple R versions

It is possible to install multiple versions of R side by side on our machine. These are stored in separate directories with their own independent libraries. In this case, we need to select the desired version of R to run. To do that in RStudio, follow the official documentation https://support.rstudio.com/hc/en-us/articles/200486138-Changing-R-versions-for-the-RStudio-Desktop-IDE.

An alternative solution is to use Docker (see Chapter 12). Docker allows us to create isolated, controlled environments in which we can install the required version of R.

11.2.1.2 Other Related Software

The specific R version may not be the only software requirement. In fact, some R functions rely heavily on other software. The main example is `pandoc`, a software used by the `rmarkdown` package to create documents. `pandoc` is not bundled with the `rmarkdown` package, but is normally provided by RStudio. Also, a specific RStudio version may be required if the project is based on some of its features (e.g., addins). Given the strong bond between R and RStudio and `pandoc` it is reasonable to list their version as well.

Using the function `sessioninfo::session_info()$platform`, we get a list of useful information about the current R session. These include the R version, the pandoc version, the operating system, CPU, and other local settings. The RStudio version is also provided if used in the session (check on your machine). See `?sessioninfo::session_info()` for further details.

```
sessioninfo::session_info()$platform
##  setting  value
##  version  R version 4.1.2 (2021-11-01)
##  os       macOS Big Sur 10.16
##  system   x86_64, darwin17.0
##  ui       X11
##  language (EN)
##  collate  en_US.UTF-8
##  ctype    en_US.UTF-8
##  tz       Europe/Rome
##  date     2022-04-26
##  pandoc   2.14.0.3 @ /Applications/RStudio.app/Contents/MacOS/pandoc/ (via rmarkdown)
```

Providing all this information about the session used to obtain the original results may be useful in the case of inexplicable reproducibility issues. For example, we can save the output as an object or print it in the project `README` or another file.

11.2.2 R packages

The R packages ecosystem changes quite rapidly, new packages are released every month and already available packages are updated from time to time. New package versions can solve bugs, add new functions, or introduce new features. At the same time, old functions can be deprecated or removed from the package itself. This means that in a year or two, our code may fail due to some changes in the underlying dependencies. To avoid this issue it is important to ensure that we install the same package versions used in the original analysis.

We can obtain a list of the packages used in the project and their version number using the function `sessioninfo::session_info()`. In the example below, we limited the number of packages displayed (usually quite long). See `?sessioninfo::session_info()` for all the options and output details.

```
sessioninfo::session_info()
```

```
## - Session info ---------------------------------------------------------
##  setting  value
##  version  R version 4.1.2 (2021-11-01)
##  os       macOS Big Sur 10.16
##  system   x86_64, darwin17.0
##  ui       X11
##  language (EN)
##  collate  en_US.UTF-8
##  ctype    en_US.UTF-8
##  tz       Europe/Rome
##  date     2022-04-26
##  pandoc   2.14.0.3 @ /Applications/RStudio.app/Contents/MacOS/pandoc/ (via rmarkdown)
##
## - Packages -------------------------------------------------------------
##  ! package   * version date (UTC) lib source
##    ggplot2   * 3.3.5   2021-06-25 [2] CRAN (R 4.1.0)
##    renv        0.15.4  2022-03-03 [1] CRAN (R 4.1.2)
##    rmarkdown   2.13    2022-03-10 [2] CRAN (R 4.1.2)
##    tidyverse * 1.3.1   2021-04-15 [2] CRAN (R 4.1.0)
##  P trackdown   1.1.1   2021-12-19 [?] CRAN (R 4.1.0)
##
##  [1] /Users/claudio/Library/Caches/org.R-project.R/R/renv/library/manual-open-science-dc9
##  [2] /Library/Frameworks/R.framework/Versions/4.1/Resources/library
##
##  P -- Loaded and on-disk path mismatch.
##
## - Python configuration -------------------------------------------------
##  python:         /usr/local/bin/python3
##  libpython:      /usr/local/opt/python@3.9/Frameworks/Python.framework/Versions/3.9/lib/p
##  pythonhome:     /usr/local/Cellar/python@3.9/3.9.12/Frameworks/Python.framework/Versions
##  version:        3.9.12 (main, Mar 26 2022, 15:51:15)  [Clang 13.1.6 (clang-1316.0.21.2)]
##  numpy:          /usr/local/lib/python3.9/site-packages/numpy
##  numpy_version:  1.22.3
##
##  python versions found:
##   /usr/local/bin/python3
##   /usr/bin/python3
##   /usr/bin/python
##
## ------------------------------------------------------------------------
```

This output provides all the main information about the R session to reproduce the original results. We can save it as an object or print it in the project `README` or another file. Saving it as an object is particularly helpful as it can be later used to automatically install all project dependencies with some custom function.

However, as different package versions may be required in different projects, we will need to reinstall the packages each time we move from one project to another. This would be very annoying. Fortunately, in R there is always a package to solve our problems. Concerning package management, the answer is `renv`.

11.3 `renv` a Package to Rule Them All

The idea behind `renv` (successor of `packrat`) is to create an isolated, portable, and reproducible environment for our projects. Using `renv` (Ushey, 2022), we can easily manage the dependencies of all our projects in a very smooth workflow. In the next sections, we introduce the main features of `renv`, how to use the `renv` workflow in our project and some other more advanced aspects.

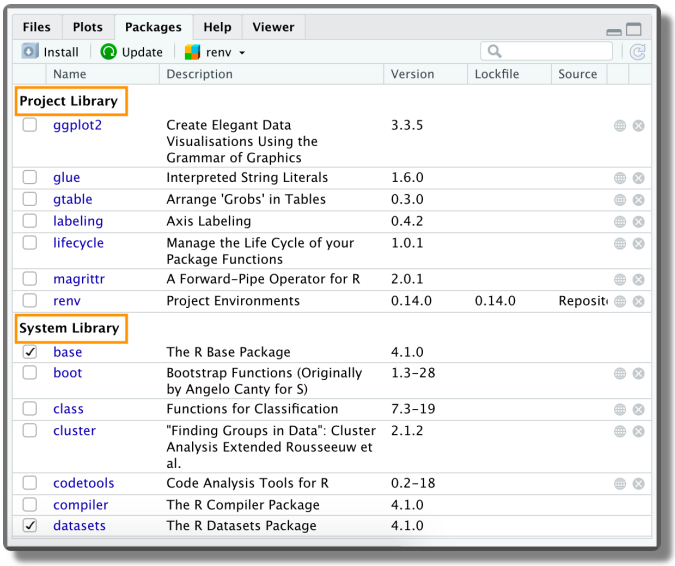
This should be enough to get started, however, we highly encourage everyone to take a tour of `renv` official documentation available at https://rstudio.github.io/renv/. There are many more aspects to learn and solutions for possible issues.

11.3.1 Introduction to `renv`

Using `renv`, each project will have its own dedicated library. In this way, we can install different versions of the same package in different projects avoiding any conflict.

Looking at the package panel we can note that there are two different libraries:

- **Project Library**. The specific packages installed within a given project. This library is unique for each project.
- **System Library**. The base packages that are automatically installed with R. This library is common to all projects. To be more precise, this library is common to all projects based on the same R version. Different R versions have their own system library.



This subdivision allows us to easily manage our project packages without affecting other projects' dependencies.

Moreover, using `renv` we can easily save the state of a project's R package dependencies. In `renv` terms, we get a *snapshot* of the currently installed packages and their versions. Other colleagues can later use this snapshot to install the specific package versions restoring the same project dependencies.

We may think that installing the same package over and over in multiple projects may be a waste of time and space. Actually, `renv` works through a very efficient global cache system that is shared across all projects. Therefore, when the same package is used in different projects, the package is installed only the first time. In future installations, `renv` will simply link to the already installed package in the cache system, resulting in a much faster process and saving disk space.

Again, `renv` is a very useful package with lots of features and options to account for any specific need. See the official documentation for all the details https://rstudio.github.io/renv.

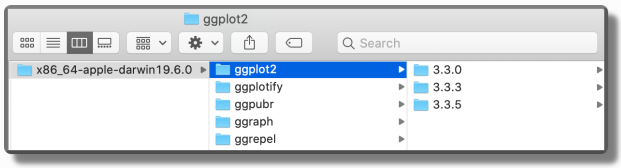
Details-Box: The renv Caching System

By default, `renv` generates its cache in the following directories:

Platform	Location
Linux	~/.local/share/renv
macOS	~/Library/Application Support/renv
Windows	%LOCALAPPDATA%/renv

Within the cache directory, `renv` organizes all packages according to the actual R version and actual platform used. This allows minimizing chances of incompatibility due to different machine or package compilation procedures.

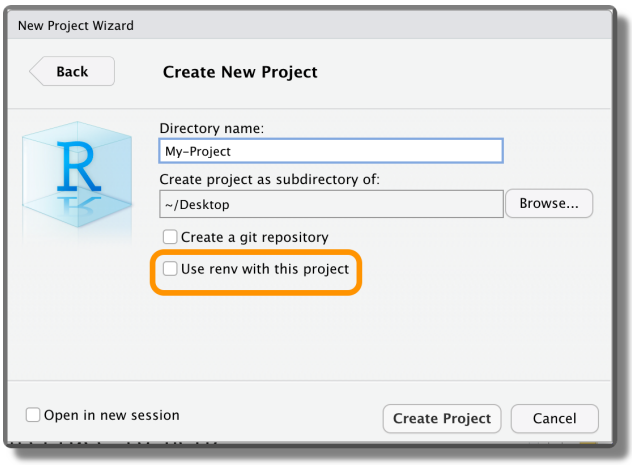
Finally, for each package, multiple versions are saved side by side. A very efficient and reliable organization. See the official documentation for further details about cache settings (https://rstudio.github.io/renv/articles/renv.html#cache).



11.3.2 The `renv` workflow

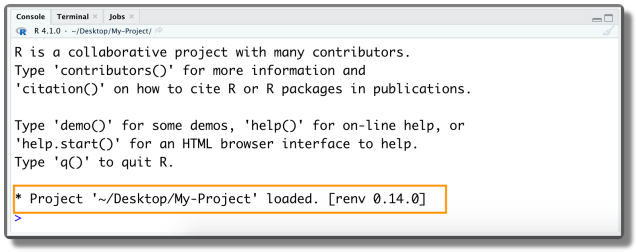
Let's see now how to get started with `renv`:

1. **Init.** We can activate `renv` in our project directly when creating a new project by selecting *"Use renv with this project"* (see Chapter 3.2.1).



   Alternatively, we can use `renv::init()` to enable `renv` in an already existing project. Once activated, `renv` automatically discovers all the packages currently

used and creates the project private library. Moreover, any time we open a project that uses `renv` a message is displayed in the console to confirm the project library is loaded.

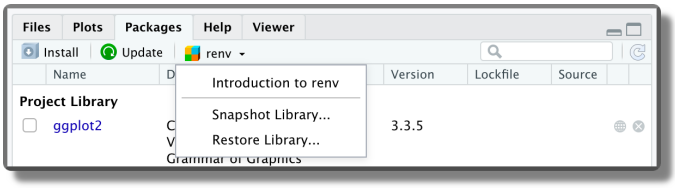


2. **Install Packages.** Now we can proceed as usual installing the required packages for our project or removing them if no longer used.

3. **Snapshot.** When required, use the function `renv::snapshot()` to save the state of a project's R package dependencies. All information regarding the version of R and R packages is automatically saved in a file named `renv.lock`.

4. **Restore.** Other colleagues can use the function `renv::restore()` to automatically install all the required packages according to the `renv.lock` file. We can also use `renv::restore()` to revert to a previously encoded state if newly installed packages introduce some unexpected problems.

Alternatively, it is also possible to manage the `renv` workflow using the menu in the package panel. However, command lines are usually preferred because they allow us to easily define specific options.



11.3.3 Advanced Features

Now we discuss some advanced features of `renv`.

11.3.3.1 Reproducibility Issues

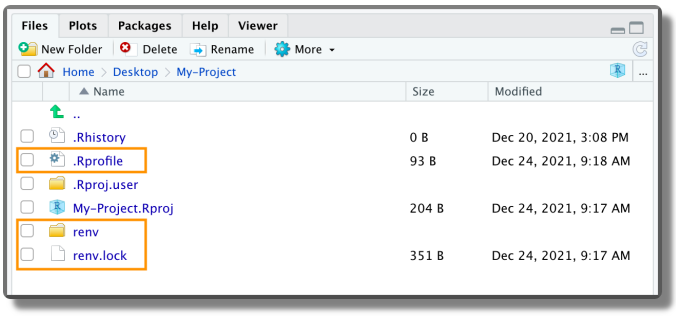
The `renv` workflow is very intuitive. We take a snapshot of the project's R package dependencies that can later be used by other colleagues to restore the same dependencies on their machines. Note, however, that `renv` does not solve all problems of reproducibility. In the first place, `renv` records the version of R but can not automatically install the required version, it only prompts a warning message:

```
## Warning: Project requested R version '3.6.1' but '4.1.0' is currently being
## used
```

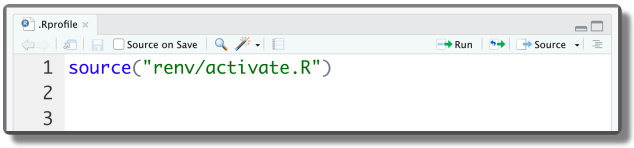
Moreover, other factors can affect the reproducibility of the results (e.g., operating system, system libraries, or the compiler). See https://rstudio.github.io/renv/articles/renv.html#caveats for detailed discussion. Most of these limits can be overcome using Docker (see Chapter 12).

11.3.3.2 `renv` Files

When `renv` is activated in a project, the following different files are created at the project root:

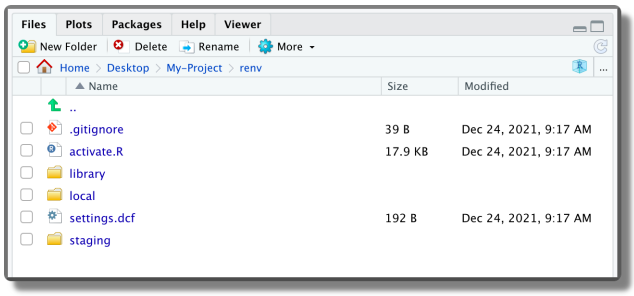


- **`.Rprofile`.** The following line is added to the `.Rprofile` to activate `renv` each time we work on the project. If for some reason we want to remove momentarily `renv`, we can simply comment out this code line.



- **`renv.lock`.** The file with the saved version of R and R packages. This file is formatted as JSON. It is a good practice to track this file with Git (see Chapter 7). Here is an example of the `renv.lock` file.

- **`renv/`.** Directory with all the files used by `renv`. In particular, `.gitignore` specifies the files that should not be tracked by Git; `activate.R` is the actual script used to activate `renv` each time we work on the project; `settings.dfc` contains `renv` project setting (see `?renv::settings` for further details); `library/` contains the link to the cached packages.



Again, for all the details, features and options, see the official documentation https://rstudio.github.io/renv.

11.3.3.3 Snapshot Types

We can choose different strategies to manage dependencies setting the `type` argument of `renv::snapshot()` function:

- **`"all"`:** Capture all packages within the active R libraries. This may also include undesired packages.
- **`"implicit"` (default):** Capture only packages which appear to be used in our project.
- **`"explicit"`:** Capture only packages listed in the project `DESCRIPTION` file (see Chapter 5.2.2.3).

- `"custom"`: Use a custom user-defined filter instead.

We can also define `renv` project-specific settings using the renv project settings mechanism (`?renv::settings`). Further details are presented in the official documentation, see https://rstudio.github.io/renv/reference/snapshot.html.

11.3.3.4 Detecting Packages

Using the default settings (i.e., `"implicit"`), `renv` captures in the `renv.lock` file only the packages which appear to be used in our project. However, `renv` may fail to discover some dependencies or packages loaded indirectly (see https://rstudio.github.io/renv/articles/faq.html#why-isn-t-my-package-being-snapshotted-into-the-lockfile-).

```
# Correctly detected by renv
library("tidyverse")

trackdown::upload_file(...)

# Not detected by renv
packages_list <- c("tidyverse", "trackdown")

lapply(packages_list, library, character.only = TRUE)

for (package in packages_list) {
  library(package, character.only = TRUE)
}
```

To overcome this issue, we can list these packages directly in the `DESCRIPTION` file.

11.3.3.5 `.renvignore` File

On the contrary, if we want to exclude specific files, we can list them in the `.renvignore` files. Listed files are not considered to discover dependencies. The same syntax as for `.gitignore` files is used.

11.3.3.6 Restore Issue

Finally, from personal experience, it may happen that `renv::restore()` fails at the first few attempts. Usually, this is due to some missing dependencies.

In some cases, these dependencies were installed during the restore process but, for unexpected reasons, they are not found by other packages. Re-running `renv::restore()` a second (or a third) time will usually solve the problem and the restore process is completed successfully.

In other cases, if these dependencies are missing from the `renv.lock` file we have to install them manually. Hopefully, these are not relevant dependencies and so not having the exact package version will not compromise reproducibility.

Documentation-Box

Extra

- Regular expressions tutorial
  https://ryanstutorials.net/regular-expressions-tutorial/

Python Dependencies

- Pip
  https://pip.pypa.io/en/stable/user_guide/#requirements-files
- PipEnv
  https://www.geeksforgeeks.org/pipenv-python-package-management-tool

R

- Release Updates
  https://cran.r-project.org/doc/manuals/r-release/NEWS.html
- RStudio select R version
  https://support.rstudio.com/hc/en-us/articles/200486138-Changing-R-versions-for-the-RStudio-Desktop-IDE

renv

- Official documentation
  https://rstudio.github.io/renv/
- Caching system
  https://rstudio.github.io/renv/articles/renv.html#cache
- Reproducibility limits
  https://rstudio.github.io/renv/articles/renv.html#caveats
- Snapshot
  https://rstudio.github.io/renv/reference/snapshot.html
- Snapshot packages
  https://rstudio.github.io/renv/articles/faq.html#why-isn-t-my-package-being-snapshotted-into-the-lockfile-

12 Docker

In the previous chapters, we learned to organize all our files and data in a well structured and documented repository. Next, we learned to use Git and GitHub for tracking changes and managing collaboration during the development of our project. Finally, we introduced dedicated tools for managing the analysis workflow pipeline and creating dynamic documents.

We are only one step from guaranteeing the reproducibility of our results. In Chapter 11, we discussed how to manage packages and other dependencies of our preferred statistical software. However, we still need to control for differences between specific software versions, operating systems, or other system aspects (.g., system libraries or available compilers). In this chapter, we introduce Docker and the container technology that allows us to create and share an isolated, controlled, standardized environment for our project. We have reached the *Holy Grail* of reproducibility.

12.1 Containers

> *"But it works on my computer!?!"* – Anonymus Programmer

Sometimes it happens that our code runs fine on our machine but fails on someone else's machine. Or even worse, that we obtain unexpected different results running the same code on different machines. Well, containers are the answer to these issues.

A container is an isolated, controlled, and portable environment that can be easily shared and used by other colleagues. Inside a container, we can collect all the elements

and required dependencies of our project. Next, we can share the container with other colleagues allowing everyone to run our project in the exact same environment as if everyone were using the same computer. Containers are a huge step towards reproducibility as we are no longer limited by differences between operating systems, software versions or other settings. Deploying our analysis using a container (plus all the other recommendations discussed in this book) guarantees the reproducibility of our results.

12.1.1 Containers and Virtual Machines

Containers and virtual machines both are forms of virtualization that allow us to emulate different computer systems on the same machine. However, there are some important differences:

- **Virtual Machines (VM).** VMs are an abstraction of the physical hardware. A special software layer, called *hypervisor*, manages the virtualization of multiple environments connecting the machine infrastructure (i.e. the physical hardware) to the different VMs. Each VM has its own *Guest* Operating System, libraries and applications. VMs are robust and secure systems, isolated from each other. However, they are usually large in size (tens of gigabytes) and they require minutes to boot (i.e., start).
- **Containers.** Containers are an abstraction at the Operating System (OS) level. A special container engine (e.g., Docker) manages the different containers. Each container has its own applications and dependencies but they share the same *Host* OS kernel (see *"Details-Box: A Linux Kernel for Everyone"* below). Containers are usually lightweight (tens of megabytes-few gigabytes) and they take just seconds to start. Thanks to their scalability and portability, containers have become industry standards for services and application deployment.

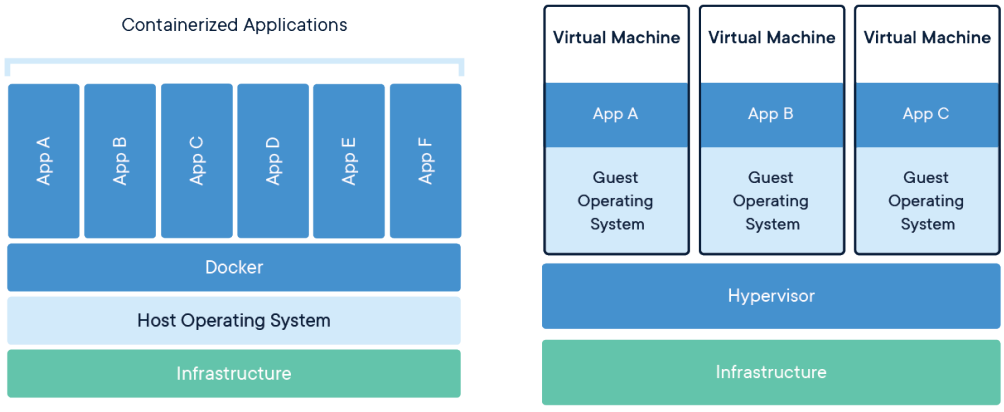
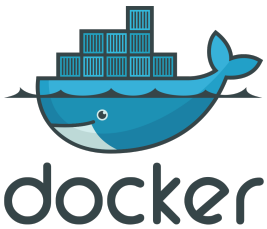


Figure 12.1: Image retrieved from Docker official documentation (https://www.docker.com/resources/what-container)

Both containers and VMs have their pros and cons and the choice between the two depends on the specific needs. Using a container allows reproducibility of the results and this is usually enough in most research projects. To learn more about containers and VMs, see https://www.docker.com/resources/what-container, https://www.backblaze.com/blog/vm-vs-containers/, or https://stackoverflow.com/questions/16047306/how-is-docker-different-from-a-virtual-machine.

12.2 Docker Getting Started



Docker (https://www.docker.com/) is one of the most popular container engines. Using Docker we can easily build, share, and run our preferred containers. Docker (and containerization in general) is a very huge topic. There are so many features and applications we can create using docker. Although things can become complicated very quickly, there are many solutions already available that we can implement right away. However, at least a minimal programming knowledge (e.g., basic use of the terminal; see Chapter 6) is required to work with Docker.

In this section, we provide a gentle introduction to Docker that will allow us to learn its basic features. In Section 12.3, we present some other slightly more advanced Docker features that are useful to manage our containers.

To learn more about Docker, see the official *Get Started* guide https://docs.docker.com/get-started/ or *Docker Tutorials and Labs* (https://github.com/docker/labs). However, These guides are more oriented toward programmers that want to deploy applications using Docker containers. Another useful Docker tutorial with a focus on R and RStudio is available at http://jsta.github.io/r-docker-tutorial/. Finally, as we dig deeper into the more advanced features of Docker, we will spend lots of time navigating the official documentation (https://docs.docker.com/).

Docker can appear very complex and intimidating at first but, as we will become more familiar with it, we will love it (hopefully).

12.2.1 Docker Elements

Before starting to play with containers, let's describe the main ingredients of the Docker workflow. Using Docker, we have three main elements:

- **`Dockerfile`.** A text file with the instructions defining how to build an image. We can think of it as the recipe used to create an image.
- **Image.** A special file that represents the prototype of a container. We can think of it as the original mould from which multiple new containers are created.
- **Container.** An instance of an image is a container. Here is when things come to life. We can create several running containers from the same image.

To summarize, we start by defining a `Dockerfile` with all the instructions to create the desired container. Next, we `build` an image (i.e., a prototype) of our container. Finally, we `run` the image to obtain a running container and get the work done. In the next sections, we describe each step in more detail.

Tip-Box: Docker Daemon

Often we find the term *"Docker daemon"* in the Docker documentation available online (or in error messages). No worries, our machine is not possessed by evil spirits.

A daemon is simply a computer program that runs as a background process (see https://en.wikipedia.org/wiki/Daemon_(computing)). The Docker daemon is the process that manages images, containers, and all other elements in the Docker workflow.

12.2.2 Install Docker

The installation procedure depends on the specific operating system. Install Docker following the instructions at https://docs.docker.com/get-docker/.

Note that, on macOS and Windows machines, we need to install Docker Desktop (simply follow the instructions at the previous link). Docker Desktop is an application that allows us to run Docker containers. We need to open Docker Desktop every time we work with Docker (this will start the Docker daemon).

Details-Box: A Linux Kernel for Everyone

As described in the previous section, containers are an abstraction at the Operating System (OS) level and all containers share the same *Host* OS. Therefore, we may wonder: how can Docker manage different OS (i.e, macOS, Windows, and Linux) on different machines allowing reproducibility? To answer this question, let's clarify some points.

Containers share the same *Host* OS **kernel**. A kernel is the lowest level of an operating system that is closer to the hardware. The kernel deals with basic services like memory management, process management, device driver, and system calls. On top of the kernel, we have all the other operating system applications and user interfaces that allow us to interact with the computer.

Docker is based on a **Linux kernel**. Wait! How can this be possible? If we have already installed a Linux OS on our machine, Docker can run containers directly using our kernel. All Linux distros (e.g., Ubuntu or Debian) are based upon the Linux kernel. But, what if we are on macOS or Windows machines? Well… the answer is Virtual Machines. When we install Docker on a macOS or Windows machine, we are also installing a Linux Virtual Machine. This allows Docker to run containers using a Linux kernel on different operating systems. Therefore, Docker allows reproducibility as all containers run on a Linux kernel, independently of our actual OS.

Docker also introduced Windows containers https://docs.microsoft.com/en-us/virtualization/windowscontainers/. Note that Windows containers can only run Windows applications and they require Windows OS on the host machine.

Docker Installation Under the Hood

On a Linux machine, installing Docker will install only Docker Engine, the core of Docker that allows us to build and run containers.

On a macOS or Windows machine, instead, installing Docker will install Docker Desktop. Docker Desktop is an easy-to-install application that includes Docker Engine plus all the elements required to run Docker on macOS and Windows.

Docker Desktop also includes some extra tools (e.g., Docker Compose; see Section 12.3.2) that facilitate the building, sharing, and running of Docker containers. On Linux, these tools have to be installed separately.

12.2.3 `Dockerfile`

A `Dockerfile` is a text file with all the instructions to build an image. An example of `Dockerfile` is,

```
#----    Dockerfile    ----
FROM ubuntu:18.04

# Install packages
RUN apt-get update \
    && apt-get install -y curl

# Get and run my app
```

```
COPY . /app
RUN make /app
CMD python /app/app.py
```

Comments are preceded by `#` (note that `#` could also assume different meanings in some conditions; see https://docs.docker.com/engine/reference/builder/#syntax). Whereas, all the other commands in the `Dockerfile` are the instructions to create the image and each command is a new layer of the image. Let's introduce the main instructions:

- **`From <image>:<version>.`** The `From` instruction allows us to define a *Base Image* from which we can start building our image. By doing this, we can take advantage of the images already available online on Docker Hub (https://hub.docker.com/). We can find plenty of images for all possible needs and some images are even officially supported by Docker. Usually, these images provide a good starting point for the majority of users. We only have to find the image that better suits our needs and point to it specifying `<DockerHub-Namespace>/<Repository>:<version>` (e.g., `nvidia/cuda:<version>`). Note that it is important to always indicate the exact starting image by specifying the version tag (see Section 12.2.3.2). In Chapter 13, we introduce available images to work in R.
- **`RUN <command>.`** The `RUN` instruction allows us to execute specific commands in a shell. Note that the Linux default shell is `/bin/sh`, but this depends also on the *Base Image* used. To specify the desired shell we can use the syntax `RUN <path-to-shell> -c '<command>'` (e.g., `RUN /bin/bash -c 'echo Hello World!'`). Moreover, we can use backslashes (`"\"`) to split long commands into multiple lines. This instruction can be used, for example, to install other packages not available in the *Base Image*.
- **`Copy <host-path> <image-path>.`** The `Copy` instruction copies files or directories from our host machine to the container image. This instruction can be used to copy all our application files into the container image.
- **`CMD <command>.`** The `CMD` instruction allows us to define a default command to run when our container starts. This instruction can be used, for example, to initiate our application. Note that while `RUN` instructions are executed during the image building time creating a new image layer, the `CMD` instruction is executed only when running the image. Moreover, we can specify only one `CMD` instruction in a `Dockerfile`.

Other common instructions are:

- **`ADD <src> <image-path>.`** The `ADD` instruction copies files or directories to the container image. The `ADD` and `COPY` are similar, but `ADD` has some additional features, such as tar extraction and retrieving files from remote URLs. `COPY` should be preferred if the specific `ADD` features are not required.
- **`LABEL <key>=<value>.`** The `LABEL` instruction adds metadata to an image by specifying a key-value pair. For example, we can add the maintainer email or the licensing information.

- **`ENV <key>=<value>.`** The `ENV` instruction allows us to define environment variables that can be used in the subsequent instructions. Note that these environment variables will persist when a container is run from the resulting image.
- **`WORKDIR <path-wd>.`** The `WORKDIR` instruction sets the working directory for the instructions that follow it in the `Dockerfile`.
- **`EXPOSE <port>.`** The `EXPOSE` instruction is used to specify the network ports on which the container listens at run time.
- **`SHELL ["executable", "parameters"].`** The `SHELL` instruction allows us to define the default shell used. On linux, the initial default shell is `["/bin/sh", "-c"]`.
- **`VOLUME ["/path-to-volume"].`** The `VOLUME` instruction allows us to create directories for persistent data storage. Volumes' functioning is described in detail in Section 12.3.1.

By combining these instructions, we can create a custom image according to our needs. Docker is a very versatile tool with many features that allows us to create different kinds of applications and services. This process may seem very complex at first as we need to be familiar with many programming arguments (e.g., shell commands, file system management, and network management). In most research projects, however, we will need only a couple of simple instructions to create our image. Usually, in most Dockerfiles we simply need to:

1. Select a *Base Image*
2. Install the required packages
3. Add all the relevant files
4. Execute the commands to get everything ready

Therefore, we don't have to be a Docker ninja master, but simply need to learn the basics to create our own containers and understand other people's Dockerfiles.

We can find a complete list of `Dockerfile` instructions and detailed information for each command in the official documentation at https://docs.docker.com/engine/reference/builder/. Moreover, `Dockerfile` best practices are described at https://docs.docker.com/develop/develop-images/dockerfile_best-practices/.

Before moving on, let's discuss a few other important aspects to take into account when creating a `Dockerfile`.

12.2.3.1 Layer Caching

All instructions in the `Dockerfile` need to be specified in the exact same order as they are required to be executed. When building the image, Docker will execute each command one at a time in procedural order (i.e., from top to bottom) and each command will become a new layer of the image.

The concept of *layers* of an image is important. Any changes or additions to the `Dockerfile` require rebuilding the image again to obtain the desired results. Unfortunately, building an image from zero can take several minutes. To limit this issue, Docker

implements a smart caching system which allows us to save time. Docker retrieves from cache all the image layers that precede the changes made and builds the image layers from there on.

To take advantage of this feature, first, we need to define large layers that are unlikely to change (e.g., installing dependencies) and only at the end do we define layers that are likely to be modified. This could save us a significant amount of time.

To learn more about layer caching see https://docs.docker.com/get-started/09_image_best/#layer-caching

12.2.3.2 Image Version Tag

When indicating a *Base Image* in the `Dockerfile`, we should always specify the exact version tag. For example,

```
FROM alpine:3.15.0
```

If we do not specify the version tag, `latest` will be automatically applied by default. This approach, however, is **not recommended**. The `latest` tag does not have any special meaning per se, but it is simply the default tag when no other tags have been specified. That is, the `latest` tag does not automatically indicate the latest published image version but it is a simple word as could have been `notag`.

However, many developers tag their latest image version with the `latest` tag. In these cases, without specifying the version tag we will obtain the latest published image version. However, this is still **not recommended** for two main reasons:

- **Hindered Reproducibility.** The main *Base Images* are constantly updated, thus yesterday's `latest` image may not be the same as today `latest` image. If we do not specify the version tag, other colleagues could end up using different images versions preventing reproducibility.
- **Unpredictable Behaviour.** The `latest` tag is static, it is not dynamic. That is, it does not check automatically for the latest updates and therefore colleagues may end up using different image versions. Let's clarify this point with an example. Suppose John and Sally are working on the same project using a Docker container. The container is based on the `alpine:latest` *Base Image*. John has never used an `alpine` image before so, the first time he builds the image, the latest image version is downloaded and used. On the contrary, Sally has already worked with different `alpine` containers. When she builds the image the first time, Docker will find a copy of the `alpine:latest` image already available on her machine and it will use it although the `latest` image on her machine refers to an older image version. Therefore, John and Sally will end up using two different containers.

To summarize, the take-home message is, ***"always specify the version tag"***. This will save us from a lot of trouble.

12.2.3.3 Files Permissions

If not already familiar with file-system permission, it is worth taking a look at https://ryanstutorials.net/linuxtutorial/permissions.php.

To summarize, permissions indicate what we can do with a file:

- **Read (r).** We can see the file content.
- **Write (r).** We can edit the file content.
- **Execute (x).** We can execute the file (script).

Moreover, file permissions are specified separately for:

- **Owner.** The user that owns the file.
- **Group.** Every file can be assigned to a group of users.
- **Others.** Other users that are not members of the group or the owner.

We can change file owner/group using the shell command `chown [OPTION]... [OWNER][:[GROUP]] FILE...` and we can specify file permission using `chmod [OPTION]... MODE[,MODE]... FILE...`.

By default `Dockerfile` instructions are executed as root (i.e., the conventional name of the user with all permissions). Consequently, ownership of all folders, files, and volumes specified in the `Dockerfile` is assigned to the root user. By default, containers also run as root ensuring us all permissions on all files in the container. However, this may not be the best thing from a security point of view.

Without going into details, the Docker container root is also the host system root. Therefore, if the Docker container has access to some files on our host system these could be modified or deleted with root permissions. A quite important security vulnerability issue (see https://stackoverflow.com/questions/41991905/docker-root-access-to-host-system).

Users and groups management in Linux and Docker is quite an advanced topic beyond the aim of this chapter. To learn more about usernames and groups in Docker, consider https://medium.com/@mccode/understanding-how-uid-and-gid-work-in-docker-containers-c37a01d01cf and https://github.com/docker/labs/tree/master/security/userns. Security is a relevant topic if we are developing a service that will be deployed on a server. In these cases, we should adopt best practices to limit security issues. For example, we could consider creating containers with isolated user namespaces (see https://docs.docker.com/engine/security/userns-remap/).

Fortunately, security issues are not so relevant in most research projects, as Docker containers are mainly used to replicate the results on a local machine. However, most *Base Images* do define a specific default user to use when running the image. In these cases, we must guarantee the required permission to the files added in the `Dockerfile`.

For example, suppose we start from a base image (`my-base-image`) in which the default user is `my-user`. Next, we copy a folder from our host system to the container. Remember `Dockerfile` instructions are executed as root so the added files will be owned

by the user. To allow `my-user` to modify these files, we need to change the ownership (`chown`) or permissions (`chmod`). The `Dockerfile` will look like something similar to,

```
#----    Dockerfile    ----
FROM my-base-image

# The default user is "my-user"

COPY host-folder docker-folder

RUN chown -R my-user docker-folder
```

To learn more about Docker file permission, see https://medium.com/@nielssj/docker-volumes-and-file-system-permissions-772c1aee23ca.

12.2.3.4 `.dockerignore` File

When adding files from our host system to the container using `ADD` or `COPY` instructions. We can exclude files not relevant to the build by listing them in a `.dockerignore` file.

We can use exclusion patterns similar to `.gitignore` files. For more information, see https://docs.docker.com/engine/reference/builder/#dockerignore-file.

12.2.4 Build an Image

> **Instructions-Box: Start Docker Daemon**
>
> If we are on macOS or Windows, first, remember to start the Docker daemon by opening the Docker Desktop application. This is required any time we are working with Docker.

Suppose, we are in our project directory (`my-project/`) and the project has the following structure,

```
my-project/
    |-- Dockerfile
    |-- README
    |-- data/
    |-- documents/
    |-- code/
```

In the `Dockerfile`, we simply copy all the project files in the image home directory.

```
#----    Dockerfile    ----
FROM alpine:3.15.0

COPY . /home/
```

Once the `Dockerfile` is ready, we can move on and build our first image by using the command,

```
$ docker build [OPTIONS] PATH
```

where `PATH` indicates the path to the directory with the `Dockerfile`. This directory will be also used by Docker as *context* to execute all the instructions. This means that all the required files should be collected in the same folder as the `Dockerfile`.

To assign a name to our image (and optionally a tag), we can specify the option `-t` (or `--tag`) indicating the image name in the format `image-name:tag`. For example,

```
$ docker build -t my-image:1.0.0 .
```

Note that the final `"."` indicates the current directory since the `Dockerfile` is placed in the project root directory (as it is commonly done). For more details on the `build` command and a complete list of available options, see the official documentation https://docs.docker.com/engine/reference/commandline/build/.

> Tip-Box: Linux sudo docker
>
> On Linux, we need to use `sudo` to run Docker commands. Without going into too technical details, the Docker daemon is owned by the root user. Therefore, if other users want to use Docker, we have to preface any `docker` command with the `sudo` command.
>
> This may be quite annoying. To run `docker` commands as a non-root user, we can set specific group permissions. Follow the instructions at https://docs.docker.com/engine/install/linux-postinstall/.

12.2.4.1 List and Remove Images

We can get a list of our currently available images using the command `docker images` (or `docker image ls`). For example,

```
$ docker images
REPOSITORY                  TAG                 IMAGE ID            CREATED             SIZE
my-image                    1.0.0               bdcd08818a09        2 minutes ago       5.59M
```

Together with the image name and tag, we obtain the `Image ID` that can be used to uniquely identify each image (we can have different versions of the same image, each one with its unique ID).

To remove an image, we can use the command

```
$ docker image rm <Image-ID>
```

For a list of all image commands, see https://docs.docker.com/engine/reference/commandline/image/.

12.2.5 Run an Image

We defined the `Dockerfile` and we built our image. Now it's time to run our first container by using the command,

```
$ docker run [OPTIONS] IMAGE
```

We need to specify the desired image used to create the container (`IMAGE`). Moreover, we can indicate different options according to our needs. Commonly used options are:

- **`--name <name-container>.`** Assign a name to the container. Remember that it is possible to initialize multiple containers from the same image, thus we can use different names to identify them.
- **`--rm.`** By default, containers remain available even after the container is stopped. This allows us to quickly start the container again or inspect its files. Using the flag `--rm`, instead, the container is automatically removed (together with all the files) when the container is stopped.
- **`-it.`** These are two separate flags `-i` and `-t` but they are usually combined to run the container in an interactive mode and allocate a virtual terminal session, respectively. This allows us to get access to the container's shell in an interactive session.
- **`-d.`** By default, Docker runs a container in the foreground. Using the flag `-d`, instead, the container is run in detached mode (i.e., in a background process). To learn more about the detached mode, see https://www.baeldung.com/ops/docker-attach-detach-container.

Moreover, we can also specify new instructions or override `Dockerfile` default settings when running the container. For example, we can,

- **`-v <host-path>:<container-path> (or --volume).`** Attach a volume or bind mount to the container. To learn more about volumes and bind mounts, see Section 12.3.1.
- **`-e <name>=<value>.`** Define (or overwrite) an environment variable. This is usually done to customize the container settings.

- **`-p <host-port>:<container-port>`.** Map a specific container port to a host port.
- **`-u` (or `--user`).** Define the username or UID used to run the container.

For more details on the `run` command and a complete list of available options, see the official documentation https://docs.docker.com/engine/reference/run/.

Using the image from the previous example, we can run our container named `my-container` in an interactive session by,

```
$ docker run --name my-container -it my-image:1.0.0
```

Once the container is initialized, a new interactive session will start in our terminal. Now, we are inside our container. Note the `#` prompt symbol, instead of `$`, indicating that we are root users (see Section 12.2.3.3). We can check whether our project files were copied in the `home/` directory and end the session by running `exit`.

```
/ # ls -l home/
total 20
drwxr-xr-x    2 root     root          4096 Jan 17 11:00 code
drwxr-xr-x    2 root     root          4096 Jan 17 11:00 data
-rw-r--r--    1 root     root            74 Jan 17 10:55 Dockerfile
drwxr-xr-x    2 root     root          4096 Jan 17 11:00 documents
-rw-r--r--    1 root     root            24 Jan 14 14:45 README

/ # exit
```

In this case, ending the session will also automatically stop the container. This could not be the case for other containers with long-running processes (e.g., web services; for more information see https://www.tutorialworks.com/why-containers-stop/).

12.2.5.1 Stop List and Remove Containers

To stop the container, we can use the command

```
$ docker stop <container-name>
```

If we did not specify the flag `--rm`, the container will still be available after it is stopped. We can use the command `docker container ls` (or `docker ps`) with the flag `-a` to list all the containers available (default shows only running containers).

```
$ docker container ls -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
fa28e5e1c2fc        my-image:1.0.0      "/bin/sh"           5 minutes ago       Exited (0) 2
```

This allows us to quickly start the container again using the command `docker start <container-name>`. Instead, if we want to definitively remove a container to free space, we use the command

```
$ docker rm <container-name>
```

Warning-Box: Removing Container = Losing Files

Note that removing a container will also delete all its files. Even if we copied our project folder from the host machine to the container using the `COPY` instruction, changes made in the container will not affect files on the host machine.

Remember that containers are isolated from the host machine so files in the containers are distinct from the files on the host machine. This is a problem as, removing a container, we would also lose all our results. In Section 12.3.1, we discuss solutions to allow persisting data storage in Docker.

12.2.5.2 Moving Files

We could copy files (e.g., analysis results) from the container to the host machine (or the reverse way) using the command

```
# Form container to host
$ docker cp <container-name>:<container-path> <host-path>

# Form host to container
$ docker cp <host-path> <container-name>:<container-path>
```

For more details, see official documentation https://docs.docker.com/engine/reference/commandline/cp/.

12.2.5.3 Running Commands

Finally, to run a command in a running container, we use `docker exec <container-name> <command>` (see official documentation at https://docs.docker.com/engine/reference/commandline/exec/). For example, to connect to the shell of a running container (in this case `bin/sh`), we can use the command

```
$ docker exec -it my-container bin/sh
```

12.3 Other Features

In the previous sections, we learned how to create and run a container. Now, let's see some other Docker features that may be useful in our workflow.

12.3.1 Data Storage

As we have already highlighted, the container file system is isolated from the host file system. This means that, when we use a container, all changes made to the files or the results we obtain are available only in the container. If we remove the container, we will lose everything.

This may be confusing at first, as we may think that adding our project folder to the container in the `Dockerfile` would connect our local project folder to the container. However, this is not true. When building the container image, the current content of our project folder is copied into the container image. These are independent and, thus, future changes to our project folder will not affect the image content (we would need to build the image again).

Moreover, the containers we create from the same image are independent. Each one is initialised according to the image. Imagine we have made some changes in a container and then we remove the container. If we run another container (from the same image), we will not find the changes we previously made but the container is initialized according to the image's initial state. We have lost our changes.

This leads to a clear question, how can we save our work if removing a container would delete everything? In Docker, there are two solutions for persisting data storage: *bind mounts* and *volumes*.

12.3.1.1 Bind Mounts

Bind mounts allow mounting (i.e., connecting) a directory (or a single file) from the host machine to a container. This allows the container to access the files on the host machine and save the changes made. When removing the container, all changes will be preserved in the directory on the host machine.

We can mount a local directory when running a container specifying the option `-v <host-absolute-path>:<container-absolute-path>`. For example,

```
$ docker run -v /Users/<user-name>/Desktop/Trial:/home/Trial/ -it my-image:1.0.0
```

Note that both paths have to be absolute. The directory (or file) does not need to exist already neither in the host machine nor in the container. If not existing yet, the directory (or file) is created on-demand.

Bind mounts are simple solutions to allow direct access from the container to a directory (or a file) in the host system. Changes made to the files in the container are saved directly in the host system and changes made from the host system will be immediately visible in the container as well.

However, bind mounts have two main drawbacks. First, bind mounts rely on the host machine specific file system (absolute paths are different on different machines). Second, allowing the container direct access to the host file system could have important security implications.

For more details and options about bind mounts, see https://docs.docker.com/storage/bind-mounts/

12.3.1.2 Volumes

Volumes are the preferred mechanism for persisting data storage. Think of volumes as storage units that are independent of the containers. We can mount (i.e., connect) volumes to one or multiple containers allowing accessing data and saving data. When removing the container, mounted volumes will be still available allowing us to save our work between sessions.

Volumes are created and managed directly by Docker and they are isolated from the host machine. To create a named volume use the command,

```
$ docker volume create <name-volume>
```

We can mount a named volume when running a container specifying the option `-v <name-volume>:<container-absolute-path>`. For example,

```
$ docker run -v my-volume:/home/my-volume/ -it my-image:1.0.0
```

Note that, if we initialize a container with a named volume that does not exist yet, Docker will automatically create it. In this case, any data that exists at the specified location within the Docker image is added to the created named volume.

Volumes can be also defined in the `Dockerfile` using the instruction `VOLUMES`(see Section 12.2.3). This would create a new mount point. In this case, if we do not mount a named volume at the running time (using the `-v` option), an **anonymous** volume will be automatically created with any data that exists at the specified location within the docker image. Anonymous volumes behave in the same way as named volumes. The only difference is that anonymous volumes do not have a name but they are referred to by a (not very handy) random alphanumeric sequence.

We can list all the currently available volumes by,

```
$ docker volume ls
```

To remove a specific volume, use the command,

```
$ docker volume rm <name-volume>
```

For more details and options about volumes, see https://docs.docker.com/storage/volumes/.

12.3.1.3 Bind Mounts VS Volumes

Volumes should be our preferred choice as they have several advantages compared to bind mounts. In particular, volumes do not depend on the host filesystem but they are managed internally by Docker. As a result, volumes are more portable and they guarantee a greater level of security compared to bind mounts.

These are very important considerations in the case of applications or online services. In the case of research projects, however, security issues are not so relevant, as Docker containers are mainly used to replicate the results on a local machine. Therefore, bind mounts are absolutely fine and they also allow us to easily access the mounted directory from our host machine as we would normally do (note that accessing volumes from the host machine is not so immediate; see https://forums.docker.com/t/how-to-access-docker-volume-data-from-host-machine/)

However, keep in mind that when working on a project using a Docker container it is important to consistently develop the project working from the container. If we continuously switch between working from the container to the local machine, we could end up having some unexpected and unpredictable behaviours. This is like running half of our project on a machine and the other half on another different machine.

For more detail about bind mounts and volumes comparison, see https://docs.docker.com/storage/.

12.3.2 Docker Compose

Multiple containers can be used to create complex applications or services. Docker Compose is a tool that helps us manage multi-container applications. Docker Compose needs to be installed separately on Linux (see https://docs.docker.com/compose/install/), whereas Docker Compose is already available within Docker Desktop on macOS and Windows.

With Compose, we use a YAML file named `docker-compose.yml` to configure all our application's services. In this file, we define and manage all the containers, volumes, and other elements of our application. A `docker-compose.yml` looks like this:

```
#----    docker-compose.yaml    ----#

version: "3.9"

services:
  db:        # First container
    image: postgres
    volumes:
      - db_data:/var/lib/postgresql/data
    environment:
      - POSTGRES_USER=postgres
      - POSTGRES_PASSWORD=postgres
```

```
  app:        # Second container
    build: .
    volumes:
      - app_volume:/code
    ports:
      - "8000:8000"
    environment:
      - DEBUG=1
    depends_on:
      - db

volumes:
  db_data:
  app_volume:
```

Next, we can start our application simply using the command,

```
$ docker-compose up
```

When we are done, we can stop the application simply by using the command,

```
$ docker-compose down
```

Docker Compose has many features and options that allow us to manage complex applications and services. Docker Compose, however, is an advanced topic that goes beyond the aim of the present chapter. Interested readers can find an introduction to Docker Compose at https://docs.docker.com/get-started/08_using_compose/ and detailed instructions at https://docs.docker.com/compose/.

In most research projects, we can easily get everything done using a single container. However, in more complex scenarios, Docker Compose may become a very useful tool.

12.3.3 Docker Hub

Docker Hub (https://hub.docker.com/) is an online repository with thousands of Docker images that can be used as *Base Images* for our containers. As we have already pointed out, many images are officially supported by Docker. Therefore, as the first step, we should usually spend some time online finding which image provides a good starting point for our needs.

We can start a container using any of the Docker images available online using the `docker run` command as we would do for locally available images.

```
$docker run [OPTIONS] <DockerHub-Namespace>/<Repository>:<version>
```

If the specified image is not already available on our machine, Docker will automatically download the specified image from Docker Hub. Alternatively, we can explicitly download the desired image using the command,

```
$docker pull [OPTIONS] <DockerHub-Namespace>/<Repository>:<version>
```

For more details, see official documentation https://docs.docker.com/engine/reference/commandline/pull/.

12.3.3.1 Sharing Images

Docker Hubs also allows us to publish our images online. In this way, we can easily share our images with other colleagues. To do that we need to,

1. Create an account on Docker Hub (https://hub.docker.com/) following the instructions at https://docs.docker.com/docker-id/.
2. Log into Docker Hub and press the *"Create Repository"* button. We need to specify a name and a description for our repository. Next, click *"Create"*.
3. Log into Docker Hub from our local terminal specifying using the command

   ```
   $ docker login --username=<username> --email=<name@email.com>
   ```

   Next, we will be prompted to enter the password.
4. Tag our image using the command

   ```
   $ docker tag <image-id> <username>/<name-repo>:<tag>
   ```

   Remember, we can get the image id from the `docker images` output.
5. Push the image to the repository using the command,

   ```
   $ docker push <username>/<name-repo>
   ```

6. Now the image is available online and other colleagues can download it simply using the command,

   ```
   $ docker pull <username>/<name-repo>:<tag>
   ```

For more details on Docker Hub, see https://docs.docker.com/docker-hub/.

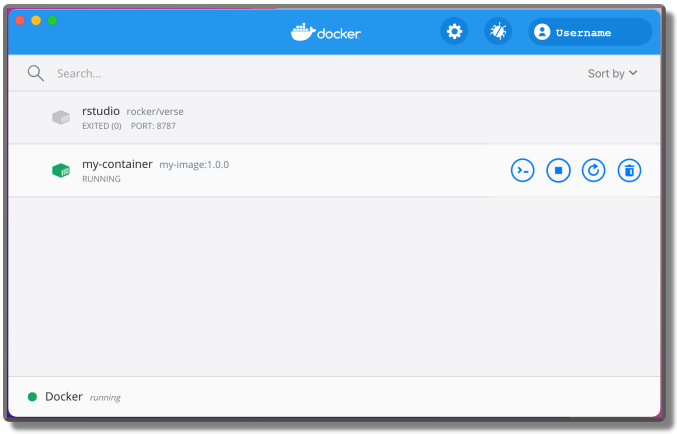
12.3.4 Docker GUI

We can manage the whole Docker workflow directly from the terminal. However, a Graphical User Interface (GUI) may be useful to manage some of the most common operations.

A GUI may also be very helpful for users who are not too familiar with the terminal. By using a point-and-click interface and providing visual feedback, a GUI helps less experienced users manage the Docker workflow. However, only limited options are usually available in a GUI. At some point, we will always need to open a terminal and write a line of code.

12.3.4.1 Docker Desktop

Docker Desktop provides its own dashboard (remember Docker Desktop is not available for Linux). We can see the currently available containers (running or stopped) and execute some basic operations such as starting/stopping a container, removing a container or getting access to the container terminal.

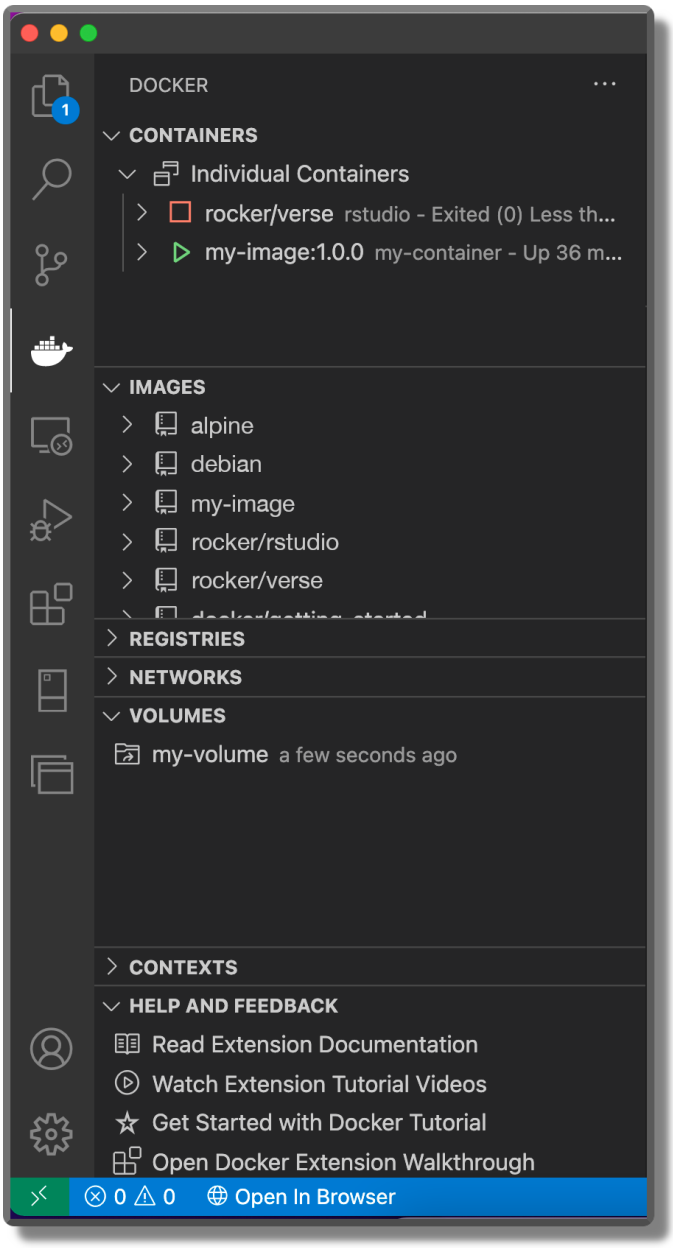


Using the Docker Desktop dashboard, we can also easily configure our Docker settings. To learn more about the Docker Desktop dashboard, see https://docs.docker.com/desktop/dashboard/.

12.3.4.2 Visual Studio Code

Visual Studio Code (VSC; https://code.visualstudio.com/) is a popular integrated development environment (IDE). VSC offers many extensions that provide tools and features to facilitate our work and customize the interface according to our needs.

VSC provides a Docker extension that allows us to easily build, manage and deploy containerized applications. Moreover, the extension enables code completion and parameter info that will assist us when editing our `Dockerfile` or `docker-compose.yml` files. A very useful feature.



To learn more about the VSC docker extension, see https://marketplace.visualstudio.com/items?itemName=ms-azuretools.vscode-docker.

Note that RStudio does not provide an extension to manage containers.

 Command Cheatsheet: Docker

We went through many commands and things could be confused. Here is a summary of all the main Docker commands.

```
docker build -t <name-image>:<tag> .    # Build an image from Dockerfile

docker images                           # List all images
docker image ls                         # List all images
docker image rm <id-image>              # Remove Image
```

Images

```
docker run --name <container-name> <image>:<tag> # Run a container from an Image
           # Other flags and options
           --rm      # Remove at exit
           -it       # Interactive terminal
           -d        # Detached mode
           -v <host-path>:<container-path>  # Attach a volume or bind mount
           -e <name>=<value>                # Define environment variable
           -p <host-port>:<container-port>  # Map port

docker ps -a              # List all containers
docker container ls  -a     # List all containers

docker stop <container-name>    # Stop container
docker start <container-name>   # Start container
docker rm <container-name>        # Remove container

docker cp <container-name>:<path> <host-path> # Copy from container to host
docker cp <host-path> <container-name>:<path> # Copy from host to container
docker exec -it <name-container> bin/bash     # Connect to container bin/bash
```

Container

```
docker volume create <name-volume>  # Create named volume
docker volume ls                    # List all volumes
docker volume rm <volume id>        # Remove volume
```

Volumes

```
docker-compose up    # Start service
docker-compose down  # Stop service
```

Docker Compose

```
docker pull  <username>/<name-repo>:<tag>   # Pull image
docker push  <username>/<name-repo>         # Push image

docker login --username=<username> --email=<email@com>  # Login
docker tag <image-id> <username>/<name-repo>:<tag> # Tag image for repository
```

Documentation-Box

Containers and Virtual Machines

- Container
  https://www.docker.com/resources/what-container
- Container vs VM
  https://www.backblaze.com/blog/vm-vs-containers/
- Docker vs VM
  https://stackoverflow.com/questions/16047306/how-is-docker-different-from-a-virtual-machine

Docker

- Official website
  https://www.docker.com/
- Official documentation
  https://docs.docker.com/
- DockerHub
  https://hub.docker.com/

Docker Tutorials

- Official *Get Started*
  https://docs.docker.com/get-started/
- *Docker Tutorials and Labs*
  https://github.com/docker/labs
- R Docker
  http://jsta.github.io/r-docker-tutorial/
- Compose
  https://docs.docker.com/get-started/08_using_compose/

Docker Elements

- Dockerfile
  https://docs.docker.com/engine/reference/builder/
- Dockerfile best practices
  https://docs.docker.com/develop/develop-images/dockerfile_best-practices/
- Caching layers
  https://docs.docker.com/get-started/09_image_best/#layer-caching
- Dockerignore
  https://docs.docker.com/engine/reference/builder/#dockerignore-file
- Build image
  https://docs.docker.com/engine/reference/commandline/build/
- Images
  https://docs.docker.com/engine/reference/commandline/image/
- Run image
  https://docs.docker.com/engine/reference/run/
- Storage
  https://docs.docker.com/storage/

- Bind mounts
  https://docs.docker.com/storage/bind-mounts/
- Volumes
  https://docs.docker.com/storage/volumes/
- Compose
  https://docs.docker.com/compose/
- Dockerhub
  https://docs.docker.com/docker-hub/
- Pull image
  https://docs.docker.com/engine/reference/commandline/pull/
- Docker GUI
  https://docs.docker.com/desktop/dashboard/
- Extension VSC
  https://marketplace.visualstudio.com/items?itemName=ms-azuretools.vscode-docker

Extra

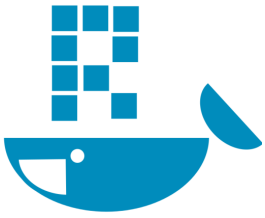
- Docker Windows containers
  https://docs.microsoft.com/en-us/virtualization/windowscontainers/
- Linux permissions
  https://ryanstutorials.net/linuxtutorial/permissions.php
- Docker root access
  https://stackoverflow.com/questions/41991905/docker-root-access-to-host-system
- Usernames and groups
  https://medium.com/@mccode/understanding-how-uid-and-gid-work-in-docker-containers-c37a01d01cf
- Usernames and groups (II)
  https://github.com/docker/labs/tree/master/security/userns
- Isolated namespace
  https://docs.docker.com/engine/security/userns-remap/
- Docker file permission
  https://medium.com/@nielssj/docker-volumes-and-file-system-permissions-772c1aee23ca
- Linux `sudo docker`
  https://docs.docker.com/engine/install/linux-postinstall/

- Attach detach container
  https://www.baeldung.com/ops/docker-attach-detach-container
- Stopped container
  https://www.tutorialworks.com/why-containers-stop/
- Moving files host-container
  https://docs.docker.com/engine/reference/commandline/cp/
- Running commands
  https://docs.docker.com/engine/reference/commandline/exec/
- Access volume from host
  https://forums.docker.com/t/how-to-access-docker-volume-data-from-host-machine/

13 Rocker

In Chapter 12, we discussed Docker, the *Holy Grail* of reproducibility. In this chapter, we introduce the Rocker Project which provides Docker Containers for the R Environment.

The Rocker Project (https://www.rocker-project.org/) provides and maintains several Docker images very useful when working with R programming language. Starting from simple images where only base-R is available, Rocker provides gradually more complex images stacking new elements and tools on top of the previous images. The main images stack is composed by,



- **`r-ver`** with Base-R.
- **`rstudio`** adds RStudio.
- **`tidyverse`** adds tidyverse and devtools packages.
- **`verse`** adds tex and publishing-related packages.

We only need to find the base image that better suits our needs. Moreover, by specifying the version tag we can define the desired version of R. For example, by indicating `rocker/rstudio:4.1.0` we obtain the R version `4.1.0`.

In addition to this images stack, Rocker provides several other images for specific use cases (e.g, Shiny, Stan, Keras, CUDA). For a list of all available images provided by Rocker, see https://hub.docker.com/u/rocker.

13.1 Rocker Getting Started

Let's see how we can use R and Rstudio inside a Docker container. To do that we choose the image `rocker/rstudio`. Note that this image is not based on RStudio Desktop (i.e.,

our usual desktop application) but is based on RStudio Server (i.e., a web-browser-based interface). Don't worry, they look almost the same. The main difference is just that we open RStudio Server through our web browser. To dig a little bit deeper regarding the differences between Desktop and Server versions, see https://support.rstudio.com/hc/en-us/articles/217799198-What-is-the-difference-between-RStudio-Desktop-RStudio-Workbench-and-RStudio-Server-.

If we are on macOS or Windows, first, remember to start the Docker daemon by opening the Docker Desktop application. Next, to create a container with the `rocker/rstudio` image, run the following command,

```
$ docker run --rm -p 8787:8787 -e PASSWORD=my-password rocker/rstudio:4.1.0
```

In the command we specified several options:

- **`--rm` (optional).** Remove the container once it exits.
- **`-p 8787:8787`.** Map the port on which we can access RStudio Server from the browser. `rocker/rstudio` uses the `8787` port, thus, for consistency, we map on the same port on our machine.
- **`-e PASSWORD=my-password`.** Specify password for RStudio Servers login. The default username is **`rstudio`**. Alternatively, we can disable authentication by setting the options-e DISABLE_AUTH=true‘.
- **`4.1.0`.** By specifying the image tag, we define the desired R version, in this case, R `4.1.0`. Remember that it is always recommended to specify the image tag.

The first time we run this command it can take a few minutes (and gigabytes) for Docker to download the `rocker/rstudio` image from Docker Hub. We will see something similar to this on our terminal,

```
Unable to find image 'rocker/rstudio:4.1.0' locally
4.1.0: Pulling from rocker/rstudio
7b1a6ab2e44d: Pull complete
34cb923ed704: Pull complete
f2f213d01c8c: Downloading [======================>          ]  200.7MB/295.8MB
7c05c07f0160: Download complete
f72cf49d9462: Download complete
6abf17a5ebed: Downloading [============================>     ]  171.7MB/218.4MB
9a11ac2d5af1: Download complete
```

Once downloaded, the container will be initialized and a few other lines will be printed on the terminal. Now, the terminal will become non-responsive: what is going on? Where is my container? Actually, the container is alive and running, the terminal has become not responsive because we attached it to the container without specifying other commands. Using `rocker/rstudio` we do not need the terminal but only our web browser.

Trick-Box: Attach Options

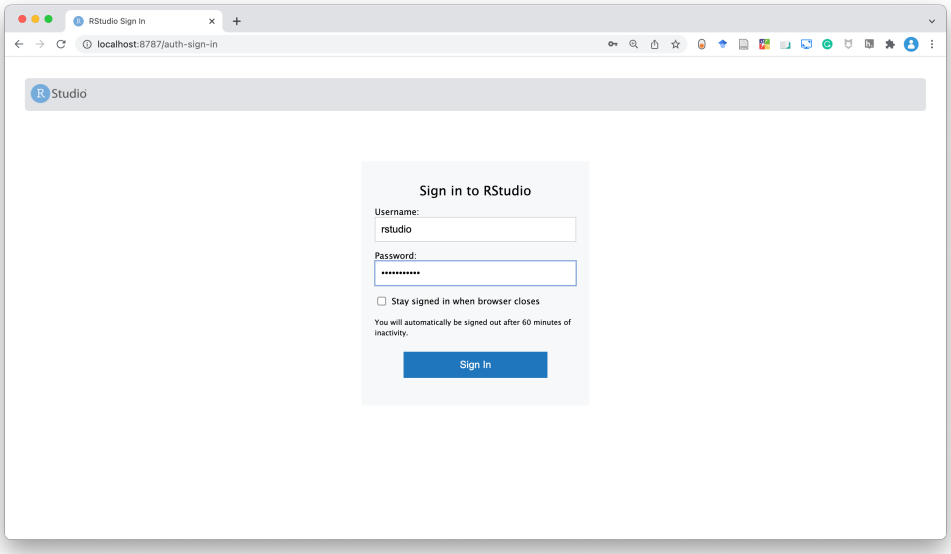
Running a `rocker/rstudio` container will leave our terminal not responsive. If we need the terminal we can simply open a new tab or terminal window

Alternatively, we can specify the `-d` flag to run the container in detached mode (i.e., in a background process) allowing us to continue working on the current terminal (connected to the host machine).

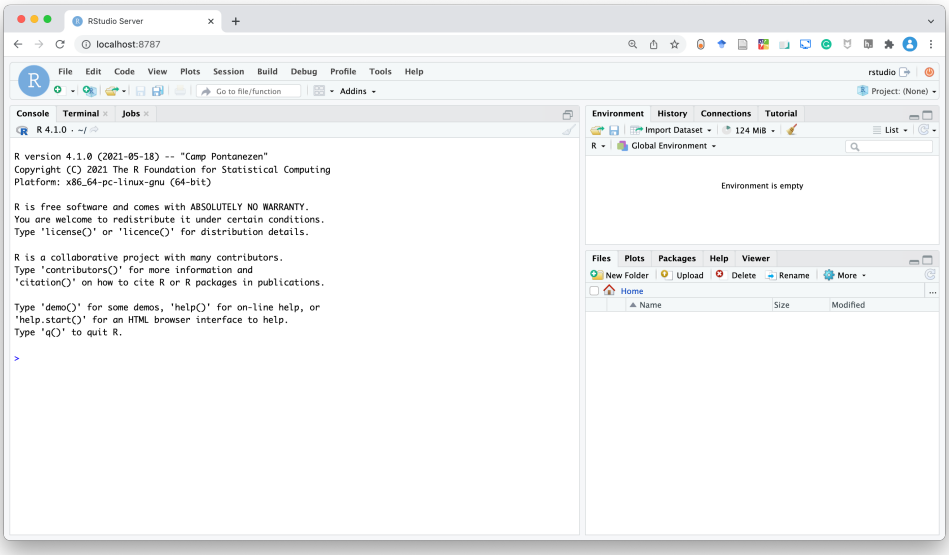
Note that, using the `-it rocker/rstudio:4.1.0 bin/bash` option, we can connect directly to the container terminal.

To connect to RStudio Server, open the web browser and go to http://localhost:8787/ (or http://127.0.0.1:8787) [TODO: check windows]. Wait! What's that? http://localhost (or http://127.0.0.1) is analogous to our home address in computer networks terms. We are saying, connect to our computer at the port `8787`, which is the one specified in the Docker command. And, *"Knock, knock! Who's there?"*, surprise surprise, RStudio Server is waiting for us. On the login page, use the following credentials:

- username: **rstudio**
- password: the password we have previously specified



Next, here we are, doesn't it look familiar? RStudio Server interface looks almost the same as our RStudio Desktop interface.



In RStudio Server we can work as we would normally do in RStudio Desktop. Writing scripts, running code, creating graphs, everything is the same, but remember, we are in a container.

If we close the browser and open a new page at http://localhost:8787/, we will still find all our files and analysis because the container is still running. But what happens if we stop the container? Well as we know (see Chapter 12.3.1), we will lose everything. Running a new container would not allow us to retrieve our files as the container is initialized from the image. Therefore, in RStudio Server we need to manage all our files and data appropriately, see Section 13.2.

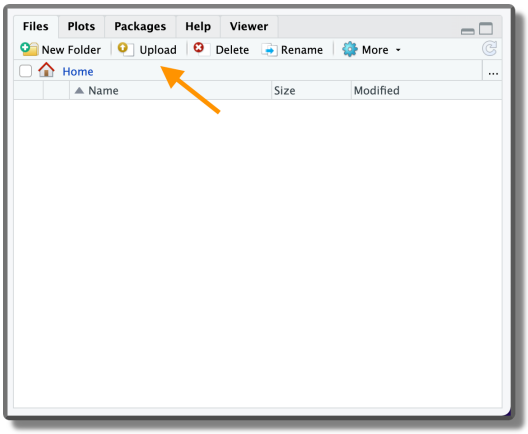
To stop the container, go to the terminal window from where we launched the Docker container and press `control + C` (or run `docker stop <container-name>`). As we specified the `--rm` flag, once stopped the container is automatically removed as well. Alternatively, we can use `docker rm <container-name>`.

13.2 Data Storage

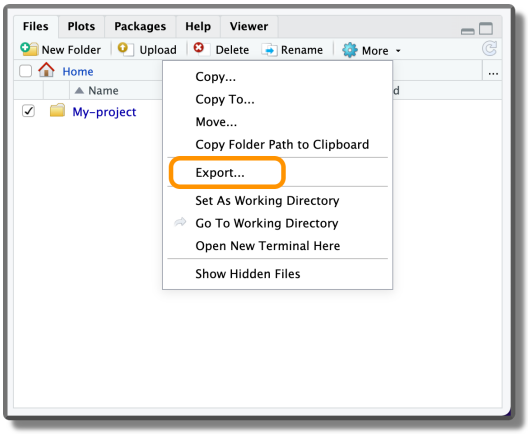
Again, containers are isolated from the host machine, thus all changes made to the files or the results we obtain are available only in the container. If we remove the container, we will lose everything. In Chapter 12.3.1, we introduced *bind mounts* and *volumes* to overcome this issue. Let's see how we can apply them and other solutions for persistent data storage when using `rocker/rstudio`.

- **Upload/Export.** A naive solution is to upload and export the data at each session form and to the host machine. Of course, this is the worst approach and it is not recommended. However, upload/export functions may still be useful for moving a few files during our working session.

To upload a file, use the *"Upload"* button in the File RStudio panel. To upload multiple files or a directory, create a zip file first.



To export a file, select the file and from the *"More"* menu in the File RStudio panel, click *"Export..."*.



- **Bind Mounts.** We can mount a local directory when running the container using the option `-v <host-absolute-path>:<container-absolute-path>`. The default user in the `rocker/rstudio` image is named `rstudio`. Therefore, the container absolute path would be `/home/rstudio/<path>`. For example, to mount the directory `/Users/<user-name>/my-project/`, we would specify,

```
-v /Users/<user-name>/my-project/:/home/rstudio/my-project/
```

- **Volumes.** We can mount a named volume when running the container specifying the option `-v <name-volume>:<container-absolute-path>`. Again, the default user is `rstudio`, thus we would specify,

```
-v My-volume:/home/rstudio/my-volume/
```

Instructions-Box: Rocker User Permissions

As always, when we deal with files we also deal with permissions. Mounting **bind mounts** at the runtime will automatically assign ownerships to the default users (`rstudio`), thus there should be no problem when modifying and saving files.

On the contrary, named **volumes** permissions depend on the settings specified when they are created. If mounting a new named volume at the runtime, this will be created but owned by the root user. Therefore, the `rstudio` user would not be able to modify and save files.

Assigning volume ownership to the `rstudio` user can be done, for example, by creating a new image using the following `Dockerfile`,

```
#----    Dockerfile    ----#
# Base image
FROM rocker/rstudio:4.1.0

# Create folder and change owner (user:group)
RUN mkdir /home/rstudio/My-project \
      && chown -R rstudio:rstudio /home/rstudio/my-project

# Define volume
VOLUME /home/rstudio/my-project
```

Note that it is important to change the directory ownership before the `VOLUME` instruction. Changes done to the directory (e.g., assign ownership) or to its content after the `VOLUME` instruction are ignored (see https://docs.docker.com/engine/reference/builder/#volume).

Next, to create a named volume, the first time we run the new image we have to specify the option

```
-v my-project:/home/rstudio/my-project
```

Doing this, a new volume named `my-project` owned by `rstudio` user will be created.

To summarize, volume permissions are slightly more difficult to manage. If we do not need to care about security, bind mounts are a much easier solution.

To learn more about file permission in Rocker containers, see the discussion at https://github.com/rocker-org/rocker/issues/246#issuecomment-316124359.

To learn more about user management (e.g., custom users, run as root group) in Rocker containers, see https://www.rocker-project.org/use/managing_users/.

13.3 Custom `Dockerfile`

Now we know how to start a container with RStudio Server using `rocker/rstudio` and how to manage bind mounts (or volumes) for persistent data storage. However, we note that each time we run the container we need to reinstall all the required R packages as they are not available in the base image. Moreover, we can not install dependencies external to the R ecosystem as we have no root permission.

If none of the images provided by Rocker satisfies our requirements, we can create our own custom image. As we have already seen in Chapter 12.2.3, we simply need to define a `Dockerfile` for our project. A possible `Dockerfile` may look like the following one,

```
#----    Dockerfile    ----#
# Base image
FROM rocker/rstudio:4.1.0

# Install external dependencies
RUN apt-get update \
    && apt-get install -y curl

# Install R packages
RUN R -e "install.packages('trackdown')"
```

Note the particular combined use of `"` and `'` to correctly specify the command.

Alternatively, we could run an R script (or a bash script) created ad-hoc with the instructions to install all the R packages (or external dependencies). In this case, the `Dockerfile` will be similar to

```
#----    Dockerfile    ----#
...

# Install R packages
COPY my-install.R /my-install.R
RUN Rscript my-install.R
```

However, as we have described in Chapter 11.3, there are better solutions to manage dependencies and guarantee reproducibility. In Section 13.4, we discuss how to manage the workflow introduced in the previous chapters when using Docker containers.

For more suggestions and best practices when using Docker with R, see https://www.r-bloggers.com/2021/05/best-practices-for-r-with-docker/.

Trick-Box: Commit Changes

There is an alternative approach that allows us to permanently modify an image without changing the `Dockerfile`. We can commit the changes made (this should sound familiar if we are already using git).

Following this approach, we simply install all the required R-packages from RStudio as we are used to.

In the case of external dependencies, instead, we can overcome the non-root user issue by running the following command `docker exec -it <container-name> bin/bash` on a new terminal window in our host machine. This command allows us to connect to the container terminal as the root user.

Once we have installed all the required dependencies, we can commit our changes using the command,

```
docker commit -m "<a descriptive message>" <container-name>  <image-name>:<tag>
```

For more details and options, see https://docs.docker.com/engine/reference/commandline/commit/.

This approach, however, is not recommended as it is not reproducible. Of course, we can share our custom image, but colleagues would have a hard time figuring out what exactly the image contains and how it was created. Using a `Dockerfile` is recommended as it indicates all the instructions allowing reproducibility and facilitating debugging.

13.4 Docker Workflow

Let's see how we can combine the workflow introduced in the previous chapters together with Docker containers to guarantee results reproducibility.

Of course, the optimal approach will depend on the specific project aims and needs. However, we can think of two main scenarios in which Docker containers are used in our projects:

- **After the Development.** Imagine that we have finished the development of our project and we have already obtained our results. However, as we did everything on our personal machine, we want to use a Docker container to guarantee results reproducibility. Therefore, we create a `Dockerfile`, in which we define the project dependencies, add all the required files and set everything to reproduce the results. Next, we re-run the analysis from within the container to obtain the *official* results.

Except for small variations, results obtained from within the container should be essentially identical to those obtained on our host machine.

- **During the Development.** Alternatively, we might want to manage the whole development of our project directly from within the container. In this case, the first thing we do is to create a container. During the development, we use RStudio Server provided by Rocker as our preferred integrated development environment (IDE) adding the required dependencies and all the relevant files for our project. Once we obtain the results, we are already sure these will be reproducible from within the container (as long as we correctly followed the workflow). Nevertheless, a second run of the analysis to check that everything flows smoothly never hurts.

Of course, there is not a strict line between the two scenarios and many other solutions are possible. We are free to choose (or create) the solution that better suits our needs. The only important thing to keep in mind is that to guarantee reproducibility, the final results should be obtained by running the whole analysis from within the container.

In most cases, creating the container after the development is fine. Most projects do not have strict requirements that could potentially hinder results replicability (we would obtain essentially the same results on the host machine and from within the container) and rerunning the whole analysis should not be a problem in terms of time and power required. On the other side, following this approach, we have the advantage of working with our familiar tools on the host system. Moreover, we do a double check of the results by re-running the analysis from within the container.

Developing the project directly from within the container could be necessary if there are specific requirements (e.g., OS, specific software, compilers) or to avoid unexpected (rare) issues when re-running the analysis in the container instead of on our host machine. However, managing the development from within the container requires some extra tricks about how to deal with GitHub authentication or the `renv` caching system.

Let's clarify everything by considering two examples.

13.4.1 After the Development

Imagine that we structured `my-project/` following all recommendations from the previous chapters: we created an `.Rproj`, Git is used to track our project, the analysis workflow is managed by `targets`, and dependencies are recorded by `renv`. At this point, to guarantee results reproducibility, we add a `Dockerfile` defining a Docker container for our analysis. The project is structured as follows,

```
My-project/
    |-- .gitignore
    |-- .Rprofile
    |-- _targets.yaml
    |-- DESCRIPTION
    |-- Dockerfile
```

```
    |-- my-project.Rproj
    |-- README
    |-- renv.lock
    |-- .git/
    |-- analysis/
    |       |-- _taregts/
    |       |-- targets-analysis.R
    |       |-- targets-workflow.R
    |-- data/
    |       |-- raw-data.csv
    |-- documents/
    |       |-- _targets.yaml
    |       |-- report.Rmd
    |-- R/
    |       |-- my-functions.R
    |       |-- targets-utils.R
    |-- renv/
```

13.4.1.1 Fix R-version

The first step is to fix the R-version used in the analysis and copy all the required files in the image. The `Dockerfile` will look similar to

```
#----    Dockerfile    ----
FROM rocker/rstudio:4.1.2

# Copy project files
COPY . /home/rstudio/my-project

# Change ownership
RUN chown -R rstudio:rstudio /home/rstudio/
```

In particular, in the `Dockerfile`:

- First, we define as base image `rocker/rstudio:4.1.2`.
- Next, we copy all the files from the project folder to the `/home/rstudio/my-project` directory in the image. Note that we need to copy all the files required to run the analysis but not the results themselves. Ideally, this command is executed on the project downloaded from GitHub (or other online repositories). As discussed in Chapter 7, we push on the online repository only the code and files required to obtain the analysis using the `.gitignore` file to avoid Git tracking the results. Alternatively, we can also use the analogues `.dockerignore` file to prevent Docker from copying in the image unwanted files (see Section 12.2.3.4).

- Finally, we ensure that all files and folders added inside `/home/rstudio/` are owned by the `rstudio` user. In this way, we avoid any permission issues.

Moreover, we need to provide appropriate instructions in the `README` to allow other colleagues to reproduce the analysis. In this case, it could be something similar to:

> To reproduce the analysis:
>
> 1. Download the project repository at or by running,
>
>    ```
>    git clone <link-repository>
>    ```
>
> 2. Build the image by running,
>
>    ```
>    docker build -t my-project:1.0.0 .
>    ```
>
> 3. Run the container by running,
>
>    ```
>    docker run --rm -p 8787:8787 -e PASSWORD="<your-PW>" my-project:1.0.0
>    ```
>
> 4. In your browser, go to http://localhost:8787 and login to RStudio Server (username: rstudio; PW:).
> 5. Open the R project by double-clicking the file `my-project/my-project.Rproj` file you can find in the project. A new R-studio session will open.
> 6. Run `renv::restore()` to install the project's dependencies (have a coffee, it takes some time).
> 7. Run `targets::tar_make()` to run the analysis using `targets`. (Note that this will properly work because we defined targets script and store locations in the `_targets.yaml` file).
> 8. To compile the report, open `documents/report.Rmd` and compile the file clicking the `Knitr` button (or run `rmarkdown::render(input = 'documents/report.Rmd')`).

Colleagues not familiar with the workflow may require some more details, but following the instructions step by step, they should always be able to reproduce the results.

13.4.1.2 Add R-packages

The previous image would require collaborators to re-install all the dependencies each time they run the container. This is very time-consuming. Moreover, we can not guarantee

that all R-packages will be available in the future. A better approach is to execute the `renv::restore()` step directly in the `Dockerfile`. To do that, we can use the following instructions,

```
#----    Dockerfile    ----
FROM rocker/rstudio:4.1.2

# Copy project files
COPY . /home/rstudio/my-project

# Install renv
ENV RENV_VERSION 0.15.1
ENV RENV_PATHS_CACHE /home/rstudio/.cache/R/renv
RUN R -e "install.packages('remotes', repos = c(CRAN = 'https://cloud.r-project.org'))"
RUN R -e "remotes::install_github('rstudio/renv@${RENV_VERSION}')"

# Install packages
WORKDIR /home/rstudio/my-project
RUN R -e "renv::restore()"

# Change ownership
RUN chown -R rstudio:rstudio /home/rstudio/
```

Without going into details, first, we install `renv` specifying the required version. Next, we run `renv::restore()` to install all the required R-package inside the image. Note that, when opening the project in RStudio Server, it could be still required to run `renv::restore()` to install the package inside the project. This operation, however, will now require less than a second as packages are linked directly from the cache.

To know more about how to use `renv` inside a Docker container, see https://rstudio.github.io/renv/articles/docker.html. For more details about `renv` workflow path customization options, see https://rstudio.github.io/renv/reference/paths.html.

13.4.1.3 Run the Analysis and Report

Finally, we could also include in the `Dockerfile` the instructions `targets::tar_make()` and `rmarkdown::render(input = 'Documents/Report.Rmd')` to run the analysis and compile the report, respectively. In this case, the `Dockerfile` will looks like as follow,

```
#----    Dockerfile    ----
FROM rocker/rstudio:4.1.2

# Copy project files
COPY . /home/rstudio/My-project
```

```
# Install renv
ENV RENV_VERSION 0.15.1
ENV RENV_PATHS_CACHE /home/rstudio/.cache/R/renv
RUN R -e "install.packages('remotes', repos = c(CRAN = 'https://cloud.r-project.org'))"
RUN R -e "remotes::install_github('rstudio/renv@${RENV_VERSION}')"

# Install packages
WORKDIR /home/rstudio/My-project
RUN R -e "renv::restore()"

# Run targets
RUN R -e "targets::tar_make()"

# Render report
RUN R -e "rmarkdown::render(input = 'documents/report.Rmd')"

# Change ownership
RUN chown -R rstudio:rstudio /home/rstudio/
```

Note the particular use of ' and " to correctly define the instructions. Moreover, the command `targets::tar_make()` will properly work because we defined the targets script and the store locations in the `_targets.yaml` file.

In this way, collaborators will automatically run the analysis and compile the report when building the image. Our workflow is entirely automatized.

13.4.1.4 Considerations

A downside of the presented approach is that we are required to rebuild the whole image each time we make a change inside the project folder. We can mitigate this issue by modifying the `Dockerfile` to take advantage of the Docker layers caching system (see Section 12.2.3.1). For this reason, the present approach is recommended once the project development is finished.

Moreover, following the presented approach, we create a specific image for each project. Having too many images could result in a waste of storage space. Of course, we can remove and rebuild the image when needed, but a good idea is also to publish the resulting image on Docker Hub (see Chapter 12.3.3). In this way, collages can download the already built image, and, no matter what, it will be always possible to reproduce our analysis (even if R-packages were removed from CRAN).

As we have seen, we can run the analysis and compile the documents directly in the container allowing us to completely automate our workflow. In the case of PDF, however, we have to deal with LaTeX installation. That may require some extra tricks to make everything work. A good starting point is to use as base image `rocker:verse`, which already implements tex and publishing-related packages. Moreover, we may require some

specific settings in RStudio Server (see https://support.rstudio.com/hc/en-us/articles/200532247-Weaving-Rnw-Files-in-the-RStudio-IDE and https://support.rstudio.com/hc/en-us/articles/200532257-Customizing-LaTeX-Options-in-the-RStudio-IDE) and we may need to learn how the `tinytex` R-package works (see https://yihui.org/tinytex/).

13.4.2 Rocker as IDE

Finally, let's say we want to use RStudio Server as our IDE and develop our projects from within a container. In this case, we could simply use one of the Rocker images and mount a bind mount with all the project files (see Chapter 12.3.1). For example, we can use `rocker/verse` (which has more features) by running the command,

```
docker run --rm -p 8787:8787 -e DISABLE_AUTH=true \
    -v  ~/Desktop/My-project:/home/rstudio/my-project \
    rocker/verse:4.1.2
```

Opening the browser at http://localhost:8787, we will find the `my-project` directory inside RStudio Server ready to keep developing our project. Note that ownership of bind mounts (or volumes) mounted at the running time is automatically assigned to the current user, in this case, `rstudio`.

However, when developing our project from within the container, we will find two main annoying issues: `renv` cache and GitHub authentication.

13.4.2.1 `renv` Cache

Packages installed in a session are lost when removing the container. Therefore, we are required to reinstall all R-packages each session, a huge waste of time. An easy solution is to get advantage of the `renv` cache system. To do that, we need to save `renv` caches in a bind mount (or a volume) and indicate `renv` to use that directory by setting the environment variable `RENV_PATHS_CACHE`. For example, we can create a folder in our host system `~/Docker/renv-cache` and use it as a bind mount. To run the container, we use the command

```
docker run --rm -p 8787:8787 -e DISABLE_AUTH=true \
    -v  ~/Docker/renv-cache:/home/rstudio/.cache/R/renv \
    -e  RENV_PATHS_CACHE=home/rstudio/.cache/R/renv \
    -v  ~/Desktop/my-project:/home/rstudio/my-project \
    rocker/verse:4.1.2
```

In this way, running `renv::restore()` to install the package inside the project will now require less than a second as packages are linked directly from the cache. To know more about how to use `renv` inside a Docker container, see https://rstudio.github.io/renv/articles/docker.html. For more details about `renv` workflow path customization options, see https://rstudio.github.io/renv/reference/paths.html.

13.4.2.2 GitHub Authentication

If we try to clone, push, or pull our commits from/to GitHub from within the container, we are required to authenticate at each session. To avoid this, we can add our SSH keys and `gitconfig` file to the container.

Rather than using our local host computer SSH keys, we can create a new pair of SSH keys and register it to GitHub (see Chapter 8.1.2). In this way, we can set different permissions when forking from the host computer or the Docker container. Next, we save the new SSH keys in a folder `~/Docker/ssh-key` and we use it as a bind mount.

We also need a `gitconfig` file with username, email, and other settings. For example,

```
#----    gitconfig    ----#
[user]
        name = <username>
        email = <user-email>
[merge]
        conflictstyle = diff3
```

and we save the file as `~/Docker/docker-gitconfig.txt`. Therefore, we end up having a folder with all the files needed to work with Docker between different sessions. Something like,

```
~/Docker/
    |-- docker-gitconfig.txt
    |-- renv-cache/
    |-- ssh-key
```

To run the container, we use the command

```
docker run --rm -p 8787:8787 -e DISABLE_AUTH=true \
    -v  ~/Docker/renv-cache:/home/rstudio/.cache/R/renv \
    -e  RENV_PATHS_CACHE=home/rstudio/.cache/R/renv \
    -v ~/Docker/ssh-key:/home/rstudio/.ssh\
    -v ~/Docker/docker-gitconfig.txt:/etc/gitconfig\
    -v  ~/Desktop/my-project:/home/rstudio/my-project \
    rocker/verse:4.1.2
```

Note that here the `rocker/verse` is necessary, as in this image some Linux packages required to deal with SSH authentication are already installed.

When dealing with credentials and containers, we have to pay a little bit of attention to avoid sharing our private SSH keys with others. Using bind mounts (or volumes) allows us to mount private information at runtime, without including them directly in the image.

Finally, as the commands became quite long, this may be a good occasion to have a look at Docker Compose (see Chapter 12.3.2) or try to create our first Bash script (see https://ryanstutorials.net/bash-scripting-tutorial/).

13.4.2.3 Considerations

Following this approach, we develop our project directly from within the container. The important thing to keep in mind is that, to guarantee reproducibility and avoid strange issues, if we start working from within the container we should consistently develop the project working from the container.

An advantage of this approach is that, except for projects that require specific system requirements (i.e., libraries, software), we can use the same base image to work on many different projects. We simply change the bind mounts to mount at runtime. However, once the project is finished, it is recommended to create a Docker image with all the dependencies already installed to guarantee the reproducibility of the results...FOREVER.

Documentation-Box

Rocker

- The Rocker Project
  https://www.rocker-project.org/
- Available images
  https://hub.docker.com/u/rocker
- Best practices
  https://www.r-bloggers.com/2021/05/best-practices-for-r-with-docker/
- Permissions Rocker
  https://github.com/rocker-org/rocker/issues/246#issuecomment-316124359
- Users Rocker
  https://www.rocker-project.org/use/managing_users/

`renv` and Docker

- Guidelines
  https://rstudio.github.io/renv/articles/docker.html
- Paths
  https://rstudio.github.io/renv/reference/paths.html

Extra

- Rstudio Server and Desktop
  https://support.rstudio.com/hc/en-us/articles/217799198-What-is-the-difference-between-RStudio-Desktop-RStudio-Workbench-and-RStudio-Server
- Docker commit
  https://docs.docker.com/engine/reference/commandline/commit/

- Rocker and Latex
  https://support.rstudio.com/hc/en-us/articles/200532247-Weaving-Rnw-Files-in-the-RStudio-IDE
- Rocker and Latex (II)
  https://support.rstudio.com/hc/en-us/articles/200532257-Customizing-LaTeX-Options-in-the-RStudio-IDE
- tinytex
  https://yihui.org/tinytex/

References

Blischak, J., Carbonetto, P., & Stephens, M. (2021). *Workflowr: A framework for reproducible and collaborative data science*. https://github.com/workflowr/workflowr

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

Kothe, E., Zandonella Callegher, C., Gambarota, F., Linkersdörfer, J., & Ling, M. (2021). *Trackdown: Collaborative editing of rmd (or rnw) documents in google drive*. https://CRAN.R-project.org/package=trackdown

Kubilius, J. (2014). Sharing Code. *Iperception*, *5*(1), 75–78. https://doi.org/10.1068/i004ir

Landau, W. M. (2022a). *Tarchetypes: Archetypes for targets*. https://CRAN.R-project.org/package=tarchetypes

Landau, W. M. (2022b). *Targets: Dynamic function-oriented make-like declarative workflows*. https://CRAN.R-project.org/package=targets

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

Richard McElreath (Director). (2020, September 26). *Science as Amateur Software Development*. https://www.youtube.com/watch?v=zwRdO9_GGhY

Ushey, K. (2022). *Renv: Project environments*. https://rstudio.github.io/renv/

Wickham, H. (2021). *Conflicted: An alternative conflict resolution strategy*. https://CRAN.R-project.org/package=conflicted

Wickham, H. (2022). *Testthat: Unit testing for r*. https://CRAN.R-project.org/package=testthat

Wickham, H., Danenberg, P., Csárdi, G., & Eugster, M. (2021). *roxygen2: In-line documentation for r*. https://CRAN.R-project.org/package=roxygen2

Wickham, H., Hester, J., Chang, W., & Bryan, J. (2021). *Devtools: Tools to make developing r packages easier*. https://CRAN.R-project.org/package=devtools