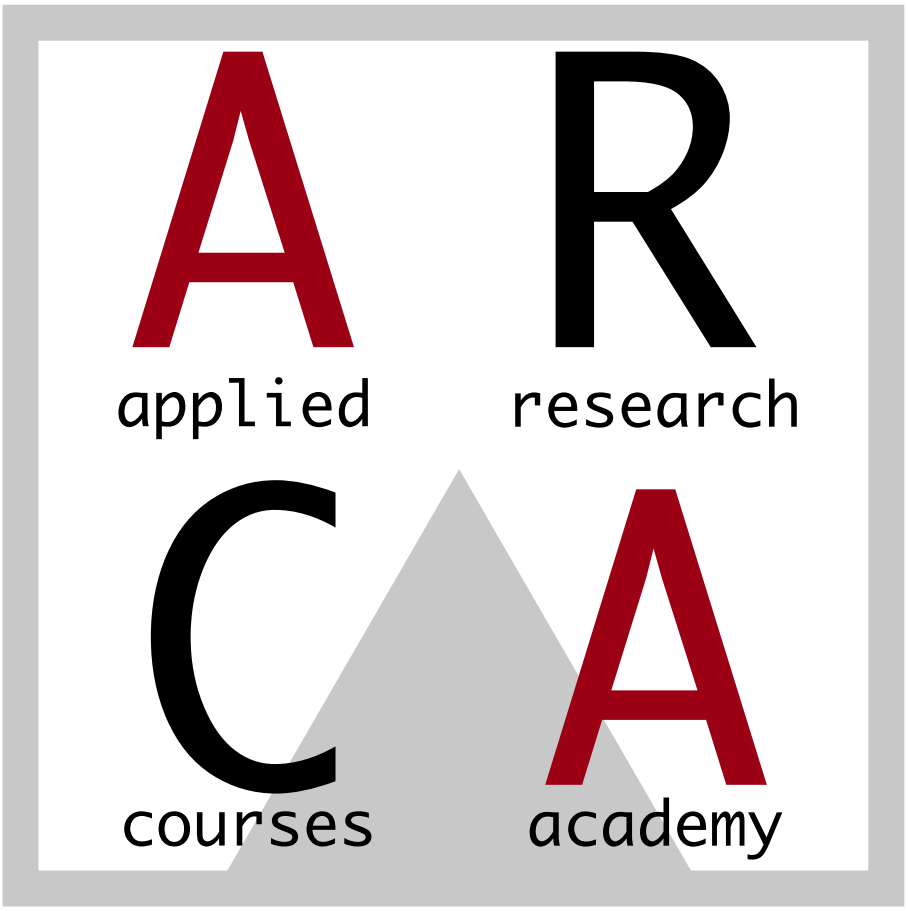
The Open Science Manual

Make Your Scientific Research Accessible and Reproducible

Claudio Zandonella Callegher and Davide Massidda

04 May, 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

If you think there is something missing, something should be described better, or something is wrong, please, feel free to contribute to this book. Anyone is welcome to contribute to this book.

This is the hearth of open-source: contribution. We will understand the real value of this book not by the number of people that will read it but by the number of people who will invest their own time trying to improve it.

For typos (the probability of typos per page is always above 1) just send a pull request with all the corrections. Instead, if you like to add new chapters or paragraphs to include new arguments or discuss more in detail some aspects, open an issue so we can find together the best way to organize the structure of the book.

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, 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 **??**, we discuss the main guidelines regarding organizing, documenting, and sharing data.
- In Chapter **??**, we provide general good practices to create readable and maintainable code and we describe the functional style approach.
- In Chapter **??**, we provide a basic tutorial about the use of the terminal.
- In Chapter **??**, we introduce Git software for tracking changes in any file during the development of our project.
- In Chapter **??**, we introduce GitHub for managing collaboration using remote repositories.
- In Chapter 4, we discuss how to manage the analysis workflow to enhance results reproducibility and code maintainability.

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

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

Warning-Box:

Warning-Boxes are used to provide important warnings.

Instructions-Box:

Instructions-Boxes are used to provide detailed instructions.

Details-Box:

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

Trick-Box:

Trick-Boxes are used to describe special useful tricks.

Command Cheatsheet:

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

In this chapter, we introduce the Open Science Framework (OSF), an open-source project that enhances open collaboration in scientific research.

2.1 OSF Introduction

The OSF is a free, open-source web application that connects and supports the research workflow, enabling scientists to increase the efficiency and effectiveness of their research (Elliott et al., 2021). Researchers can use OSF to collaborate, document, archive, share, and register research projects, materials, and data.

Although the OSF was initially used to work on a project in the reproducibility of psychology research, it has subsequently become multidisciplinary (from Wikipedia https://en.wikipedia.org/wiki/Center_for_Open_Science). The OSF can help us during any part of the research lifecycle (Elliott et al., 2021):

- **Store research materials.** The OSF allows us to store all research materials in a manageable, secure cloud environment. The OSF ensures we and our colleagues can access materials, when needed, reducing the likelihood of losing study materials.
- **Collaborate with colleagues.** With the OSF, our work is preserved and easily accessible to only those people that should have access. By sharing materials via OSF, we ensure that everyone is working with the correct, up-to-date file versions and we never lose files again.

- **Cite research materials.** The OSF can be used to share manuscripts, unpublished findings, materials, and in-progress work. The OSF makes all of it citable so that our impact is measured and we get credit for our work.
- **Measure research impact.** The OSF provides tools to help us measure our impact (e.g., download and visit counts).

In Sections 2.2, we provide a guide on how to create an account and start a project on the OSF. In Section 2.3, we describe further features useful to researchers.

Details-Box: Title

Center for Open Science

The OSF is released by the Center for Open Science.

The Center for Open Science (COS; https://www.cos.io) is a non-profit technology startup founded in 2013 with a mission to increase the openness, integrity, and reproducibility of scientific research (Elliott et al., 2021).

COS pursues this mission by building communities around open science practices, supporting metascience research, and developing and maintaining free, open-source software tools.

2.2 Getting Started

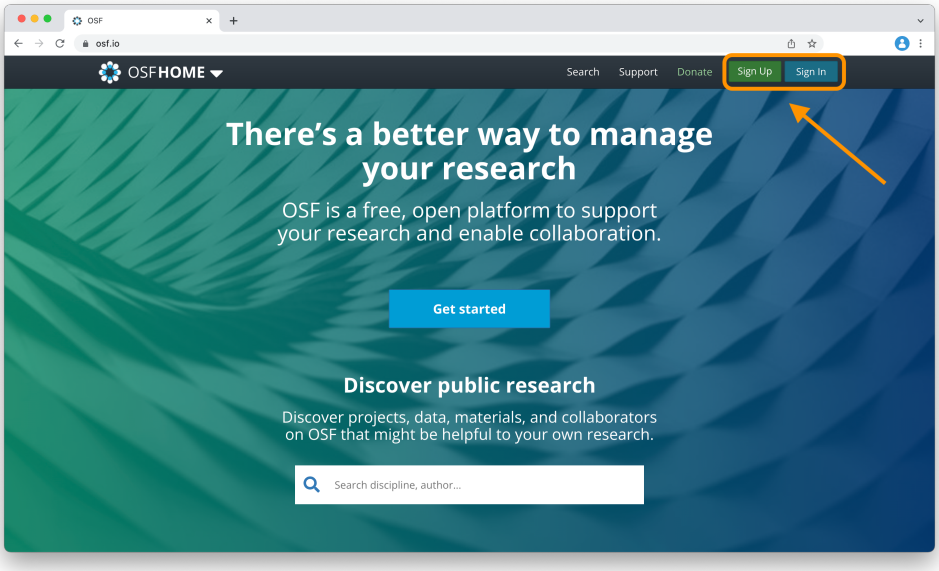
This is an introductory tutorial. For a complete guide of the OSF and a description of all features, see:

- **OSF Support** https://help.osf.io/
- **OSF Wiki** https://osf.io/4znzp/wiki/home/

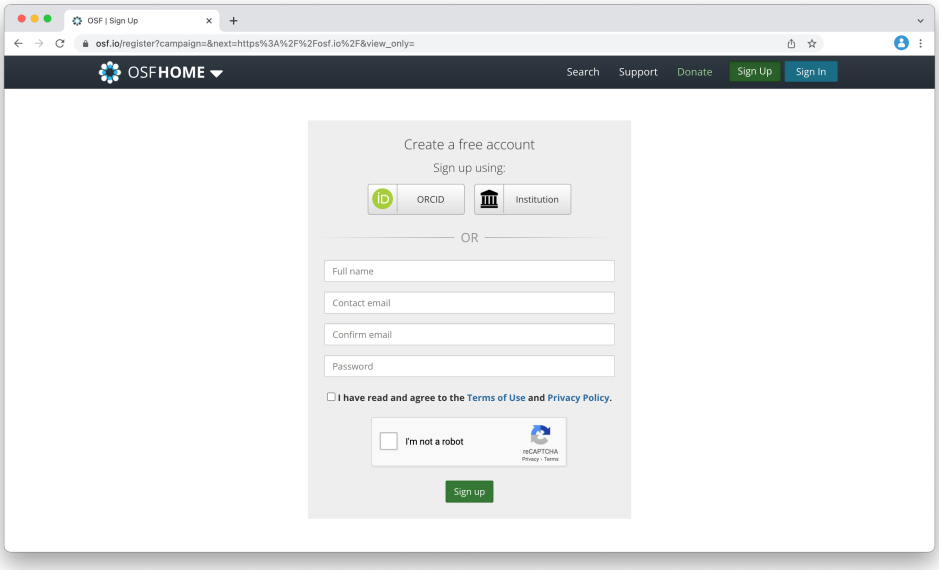
2.2.1 Subscribe

To subscribe to the OSF:

- Go to the OSF homepage (https://osf.io) and click the *"Sign Up"* (or *"Sign In"*) button in the top right corner.



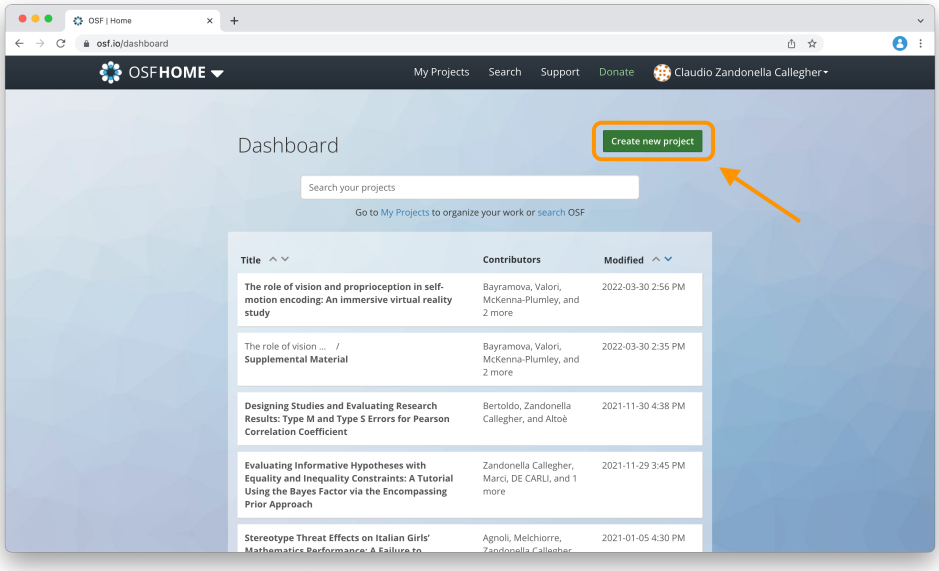
- Create a new account or use the ORCID or Institution credential to register.



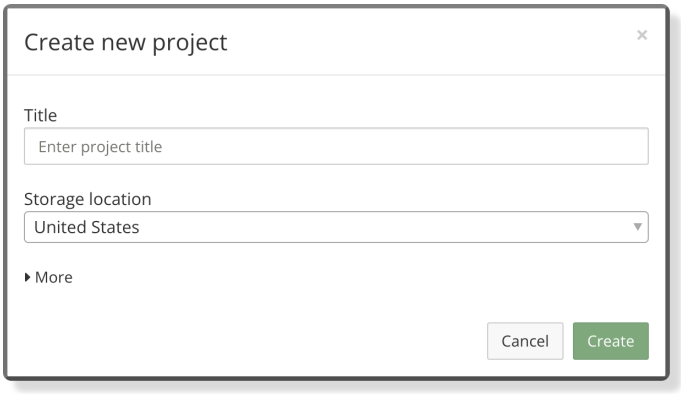
2.2.2 Create a Project

To create a new project:

- From the user Dashboard, click the *"Create new project"* button.



- Specify the project title and store location. Next, press the *"Create"* button. Note that institutions may require that data be stored in the EU to comply with storage regulations.

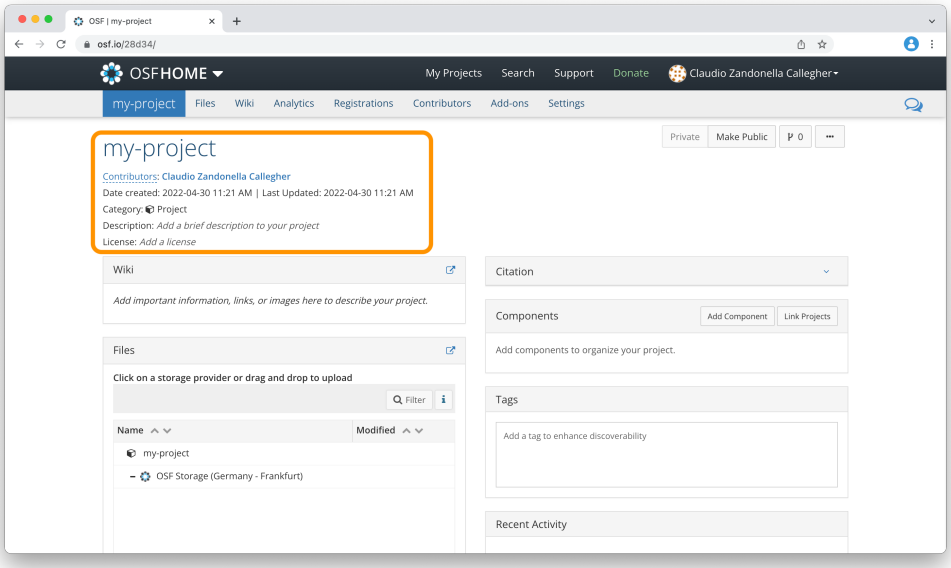


2.2.3 Project Homepage

The project homepage provides summary information about the project. In particular,

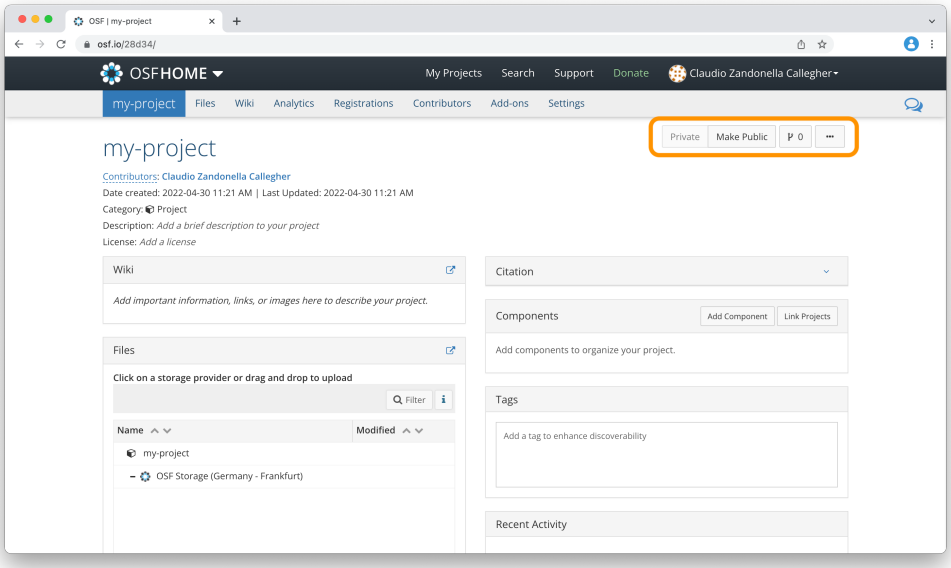
- **Project title.** The project title.
- **Contributors.** List of all contributors. See Section 2.3.2, to add new contributors to the project.
- **Date created.** Date of creation and last updated.
- **Category.** Specify the type of project.

- **Description.** Provide a brief description of the project
- **License.** Specify the project license. To know more about licenses, see Chapter 3.1.1.6.



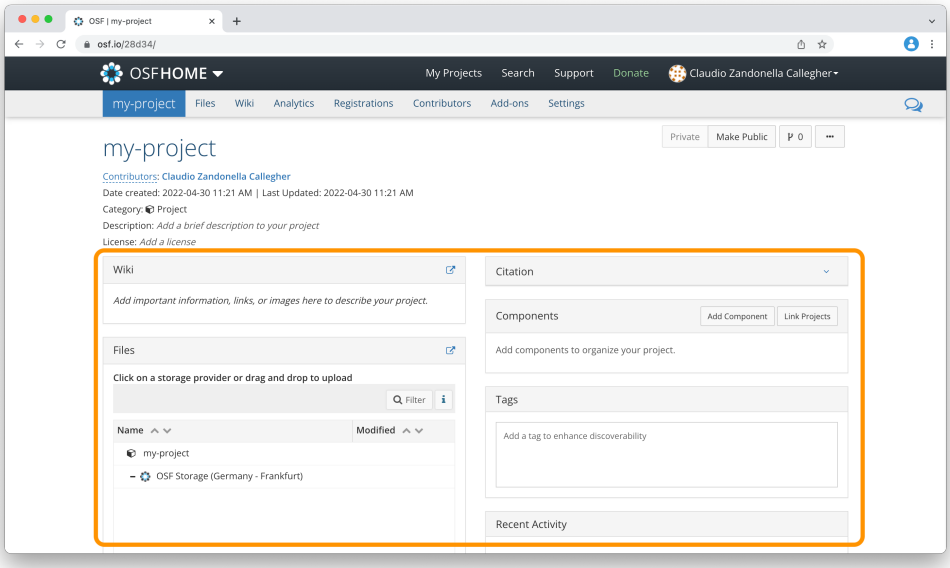
On the top right corner, we find further options about,

- **Visibility.** Check and modify the project visibility (public or private).
- **Fork.** Created copies of the project.
- Share or add a bookmark to the project.

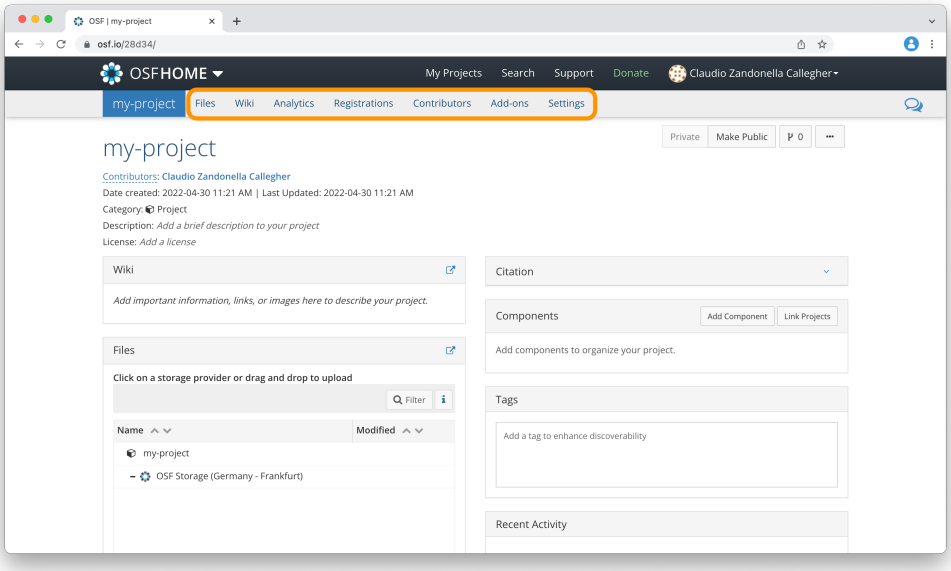


Moreover, multiple panels provide detailed information about the project. In particular,

- **Wiki.** Provide a detailed description of the project.
- **Files.** List all project files and materials.
- **Citation.** Specify how other colleagues should cite the project.
- **Components.** Add sub-projects to create a hierarchy structure.
- **Tags.** Add tags to the project to facilitate discoverability.
- **Recent Activity.** History of the project activity.



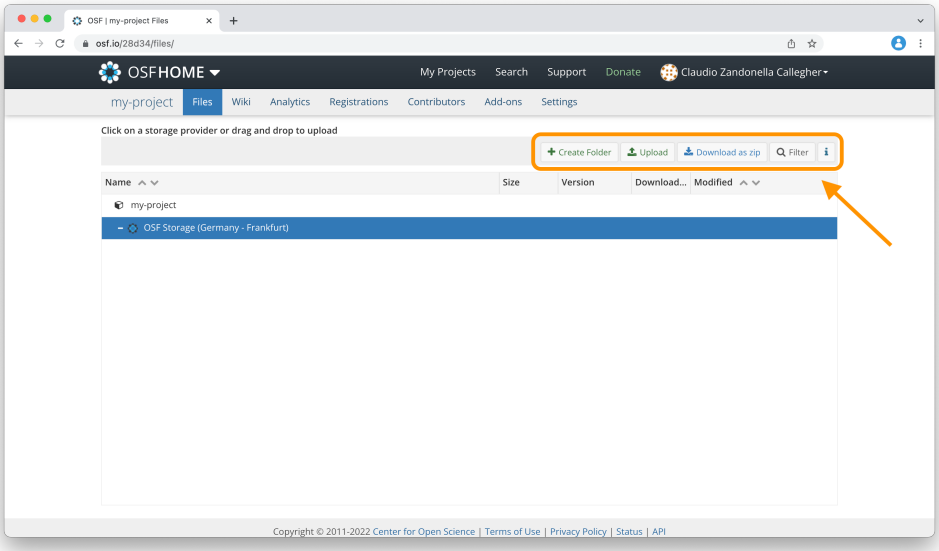
Finally, on the top of the project homepage, we find all the menu tabs to modify and manage our project.



2.2.4 Adding Files

To add a file,

- From the *"Files"* tab, select the desired OSF Storage.
- Use the buttons at the top to create folders, upload/download files.

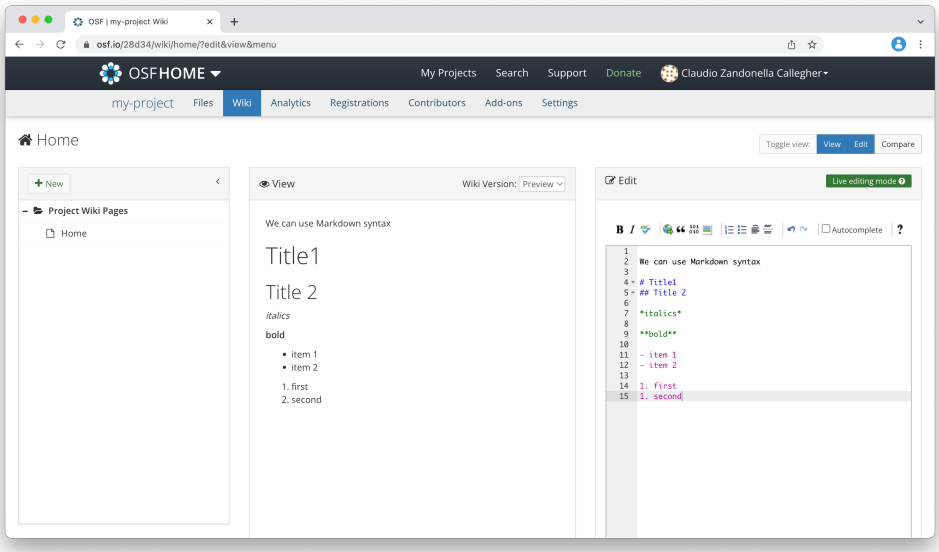


Note that we can add files also directly from the *"Files"* panel on the project homepage.

2.2.5 Documentation

To create a detailed description of the project,

- Open the *"Wiki"* tab from the top menu.
- Edit the wiki document. Note that Markdown syntax is supported (for an introduction to the Markdown syntax, consider the *"Markdown Guide"* available at https://www.markdownguide.org).



- Save the changes (button at the bottom) to update the project homepage.

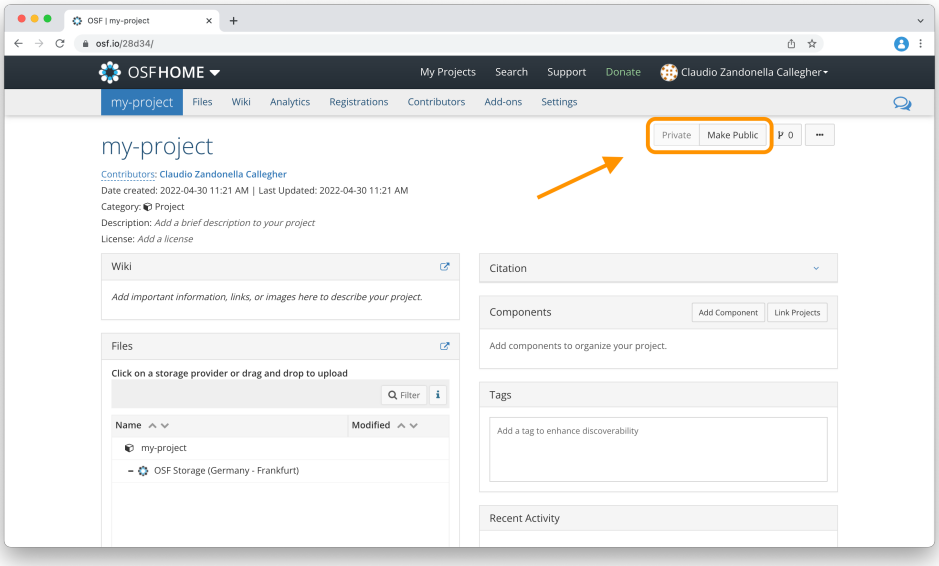
2.3 Research Features

In this section, we describe further features useful to researchers.

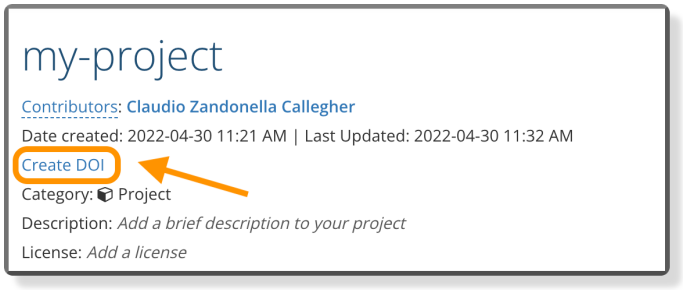
2.3.1 DOI

To obtain a DOI for the project,

- From the project homepage, change project visibility to public by pressing the *"Make Public"* button.



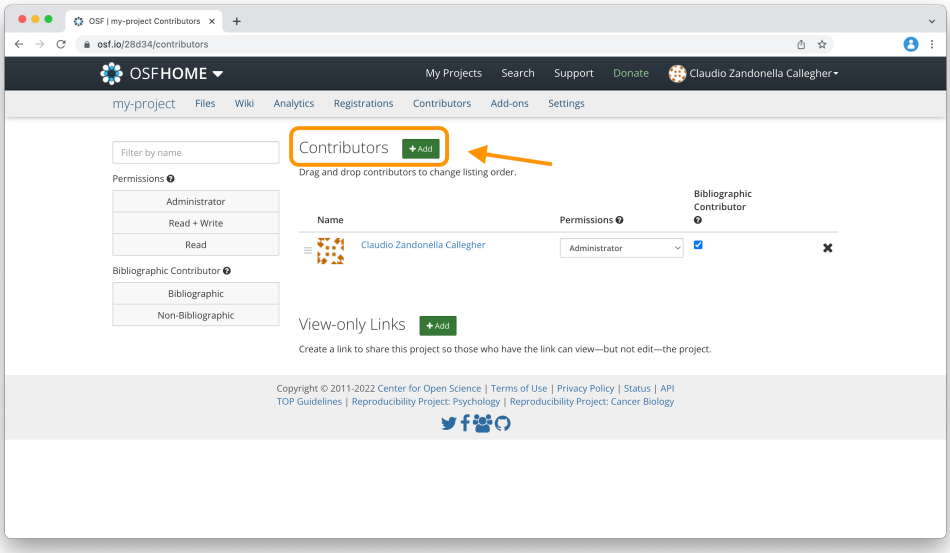
- Now the option *"Create DOI"* will be available.



2.3.2 Contributors

To add new contributors to the project,

- Open the *"Contributors"* tab from the top menu.
- Click the *"+ Add"* button in the *"Contributors"* section.

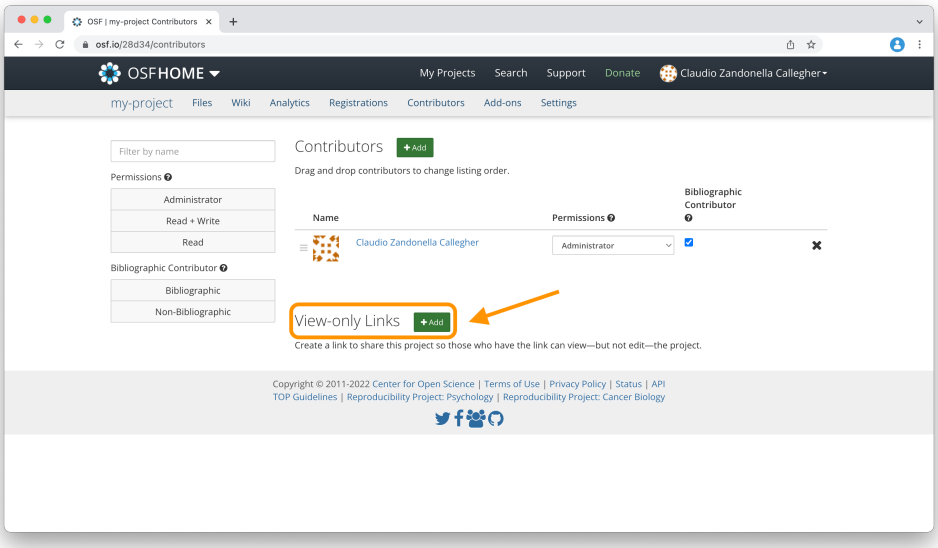


- Search users by name and add them as contributors.

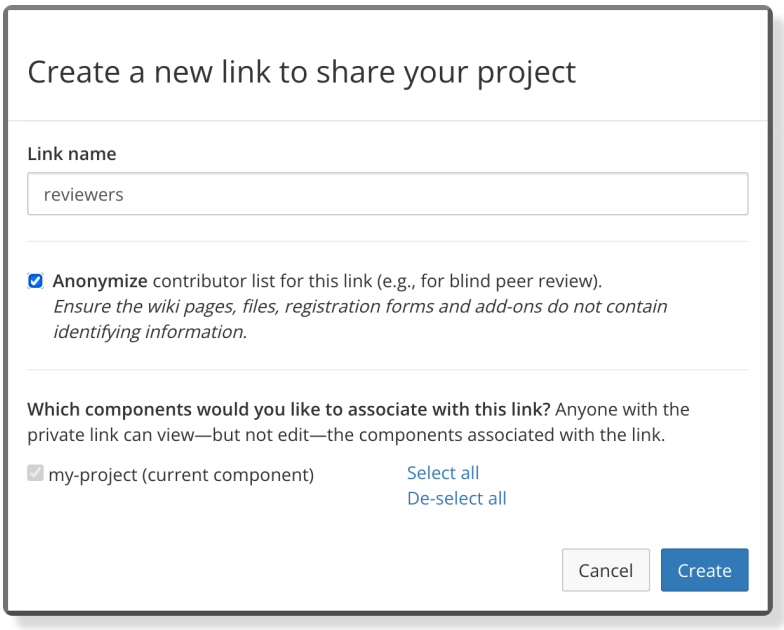
2.3.3 Sharable Anonymized Links

We can share anonymized links (hide contributor names in the project). A particularly useful feature for blinded peer review. To create an anonymized link,

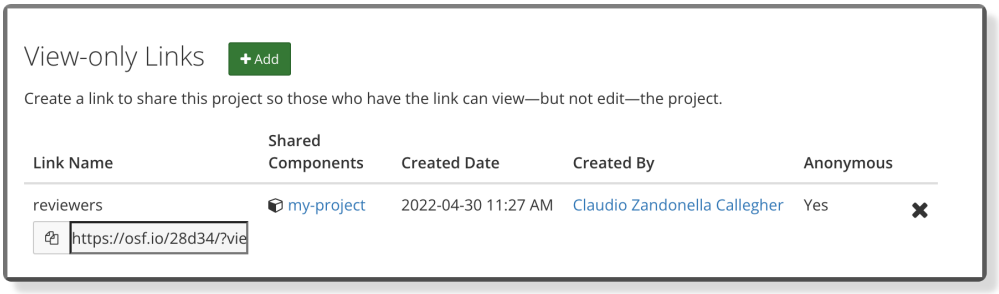
- Open the *"Contributors"* tab from the top menu.
- Click the *"+ Add"* button in the *"View-only Links"* section.



- Provide a name to identify the link and select the *"Anonymize"* option.



- The link is created and listed in the *"View-only Links"* section.



Warning-Box: Anonymization

In a blinded peer review process, we are required to remove any information about the authors.

Anonymized Links hide contributor names in the project, but it is our responsibility to check that project files contain no references to the authors.

Moreover, we need to keep private project visibility. Anonymized links are of type `https://osf.io/<id-project>/?view_only=<token>`. If the project is public, reviewers can still retrieve the original repository using the link

`https://osf.io/<id-project>` obtained by simply removing the last part of the anonymized link (i.e., `?view_only=<token>`).

2.3.4 Preprints

We can share preprints using the OSF Preprints service (https://osf.io/preprints/). The OSF Preprints is an aggregator of various preprint servers (e.g., PsyArXiv, and ArXiv).

To create a preprint, follow instructions at https://help.osf.io/article/376-preprints-home-page.

2.3.5 Add-ons and API

The OSF provides Add-ons and APIs that allow us to integrate third-party services. These include citation software (e.g., Zotero and Mendeley), storage services (e.g., Dropbox and Google Drive), Git repository-hosting services (e.g., GitHub and GitLab, see Chapter **??**), and API services to access files and project info.

For all details about Add-ons and API, see https://help.osf.io/article/377-add-ons-api-home-page

Details-Box: The osfr R Package

The `osfr` R Package (Wolen & Hartgerink, 2020) provides a suite of functions for interacting R with the Open Science Framework. In particular, using the `osfr` R Package, we can:

- *Access Open Research Materials.* Explore publicly accessible projects and download the associated files and materials
- *Manage Projects.* Create projects, add components, and manage directories and files.

Package documentation with all details is available at https://docs.ropensci.org/osfr/.

Documentation-Box

The OSF

- Homepage
  https://osf.io/
- Support
  https://help.osf.io/
- Wiki
  https://osf.io/4znzp/wiki/home/
- Preprints Service
  https://osf.io/preprints/
- Preprint Guide
  https://help.osf.io/article/376-preprints-home-page
- Add-ons and API
  https://help.osf.io/article/377-add-ons-api-home-page
- osfr R Package
  https://docs.ropensci.org/osfr/

Extra

- The Center for Open Science
  https://www.cos.io
- Markdown Guide
  https://www.markdownguide.org

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 **??**). 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 **??**, 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 **??**. 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 **??**, we discuss in detail the functional style approach, considering general good practices to write tidy, documented, and efficient code. In Chapter 4, 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 4, 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 **??**, 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 **??**).
  - *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.

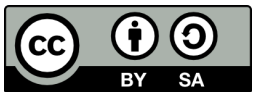
In the case of **software** or **code releasing** the most popular open-source license are:

- **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 **??**, we discuss specific licenses related to sharing data/databases. Further information about licenses can be found at:

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

Instructions-Box: Adding a License

To add a license to our project:

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

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

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

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

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

3.1.2 Naming Files and Directories

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

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

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

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

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

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

- **Prefer *lower-case* Names.** There is nothing wrong with capital letters. However, case sensitivity depends on the specific 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 **??** and Chapter 4, we describe how to adopt the functional style approach and how to manage the analysis workflow, respectively. In Chapter **??**, 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 **??** and Chapter **??**, we introduce the use of Git and GitHub to track changes and collaborate with others on the development of our project.

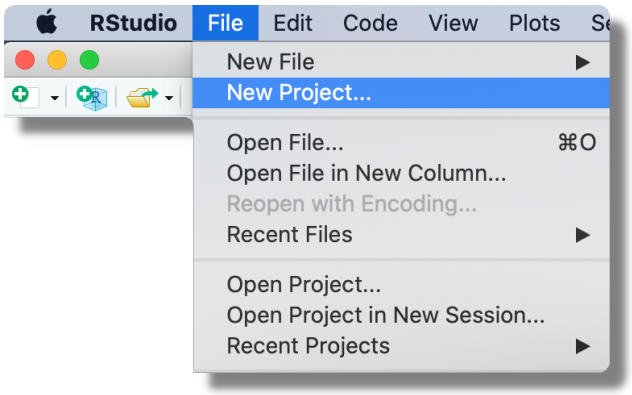
3.2 RStudio Projects

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.

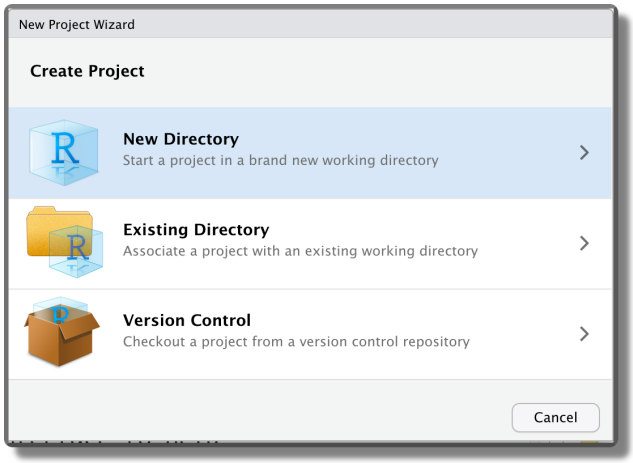
3.2.1 Creating a New Project

To create a new RStudio project:

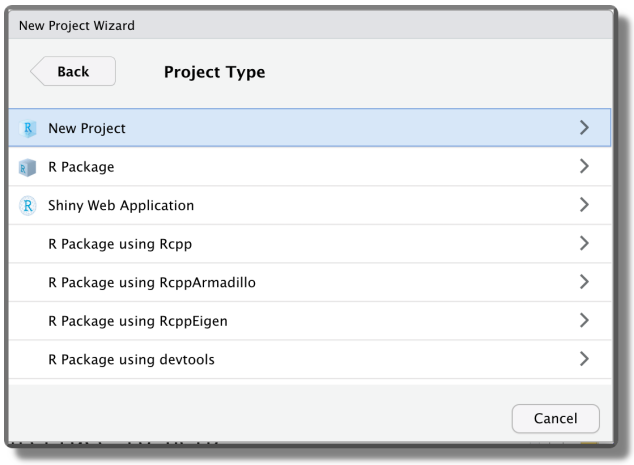
1. From the top bar menu, click *"File > New Project"*. Note that from this menu, we can also open already created projects or see a list of recent projects by clicking *"Open Project..."* or *"Recent Projects"*, respectively.



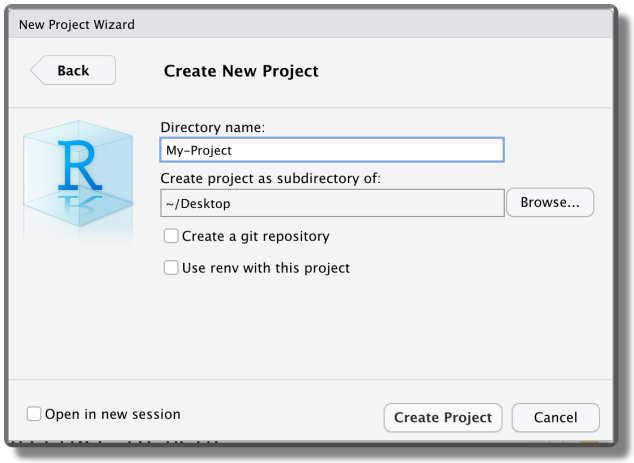
2. When creating a new project, we can decide between starting from a new directory (*"New Directory"*; the most common approach), associating a project with an already existing directory (*"Existing Directory"*), or associating a project to an online repository (*"Version Control"*; for more information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



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"*.

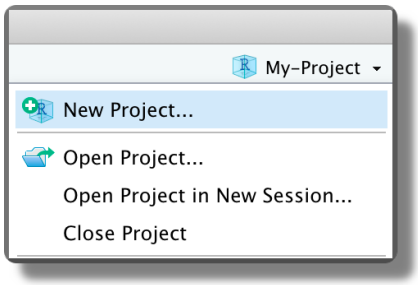


4. Finally, we can indicate the location where to create the project directory and specify the sdirectory name. This will be used also as the project name. Note that two more options are available *"Create a git repository"* and *"Use renv with this project"*. We discuss these options in Chapter **??**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

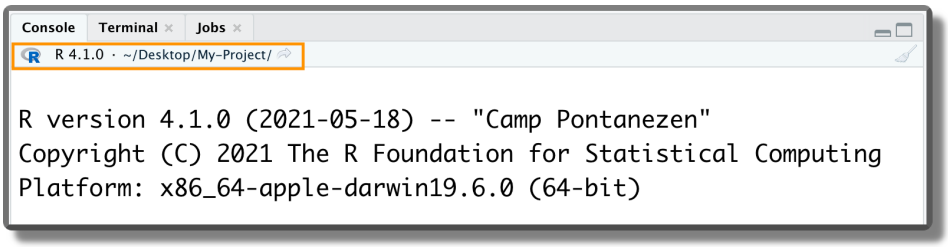
. The current project is also indicated in the top right corner. Click on it to see other project options and to close the project (*"Close Project"*; or from the top bar menu *"File > Close Project"*).



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()`.



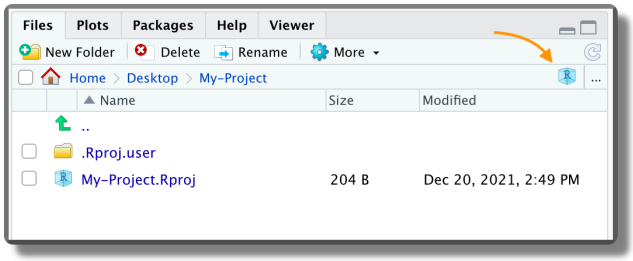
  As discussed in Section 3.1.3.1, this is a very useful feature because now we no longer have to bother about setting the working directory manually (we can finally forget about the `setwd()` command). Moreover, we can refer to any file using relative paths considering as reference the project root.

Details-Box: File Paths in R

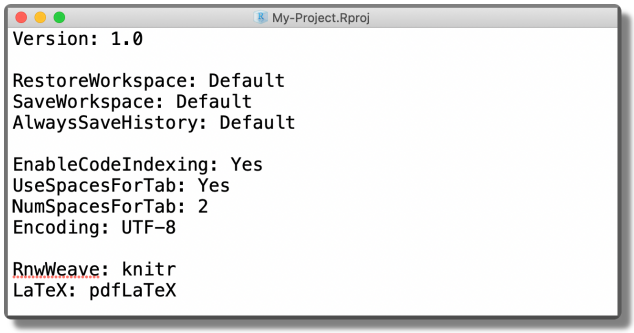
In R, we can specify file path using forward slash `"/"` or backslash `"\"` independently of the operating system we are currently using. However, the backslash has a special meaning in R as it is used as *escape character*. Thus, to specify a file path using backslash we have to double them (e.g., "my-project\\data\\my-data.csv").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

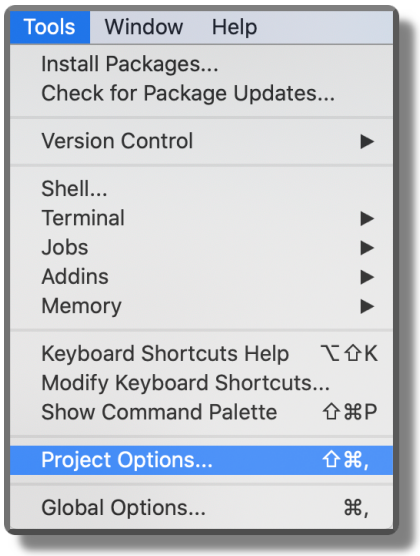
- **`<project-name>.Rproj` File.** The default Project template creates an empty directory with a single file named `<project-name>.Rproj` (plus some hidden files). Clicking on the `<project-name>.Rproj` file from the file manager (not from RStudio), we can open the selected project in a new RStudio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



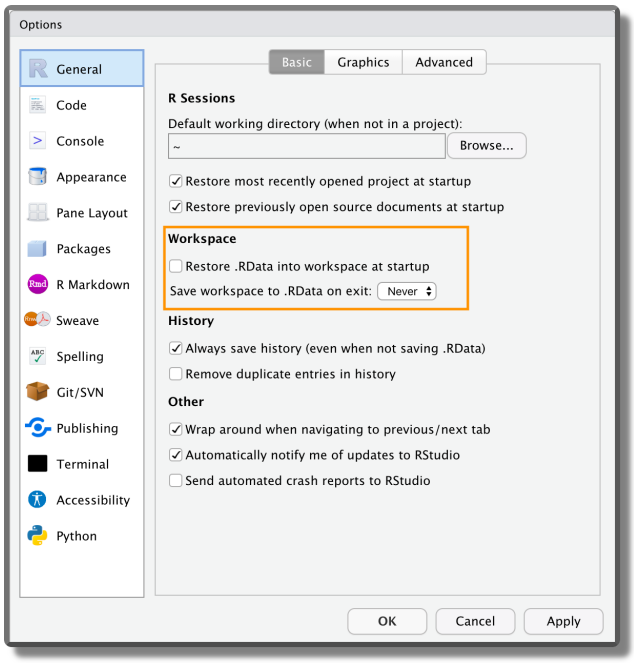
The `<project-name>.Rproj` file is simply a text file with the project settings. Using a text editor, we can see the actual content.



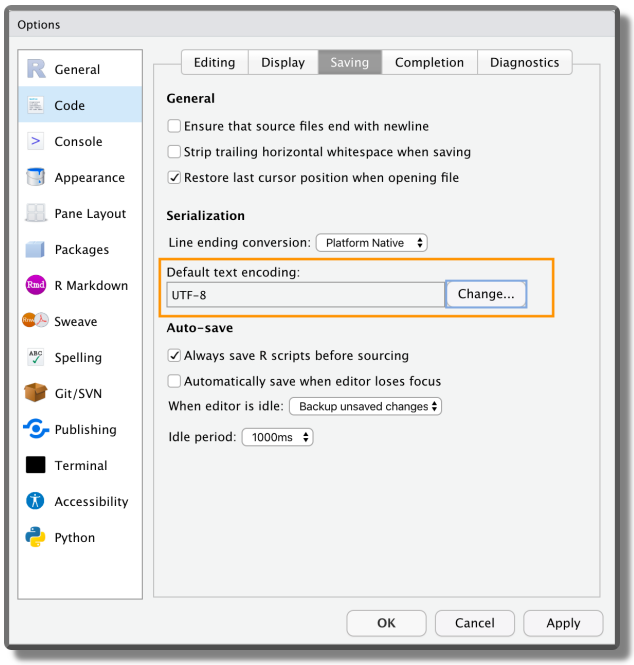
- **Project Settings.** We can specify specific settings for each project. From the top bar menu, select *"Tools > Project Options"* and specify the required options according to our needs.



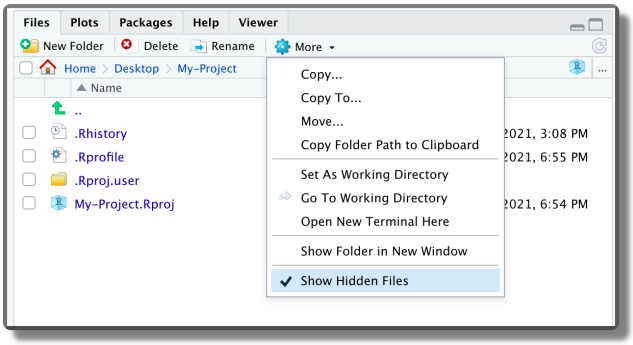
Some recommended settings should be applied as default to all projects. To do that, select from the top bar menu *"Tools > Global Options"*. From the panel *"General"* in the *"Workspace"* section, ensure the box *"Restore..."* is **not** selected and the option *"Save..."* is set to **Never** (see figure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



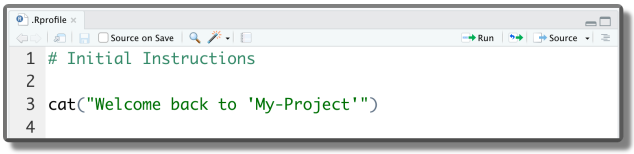
Another important setting is the encoding. From the panel *"Code"* ensure that the default text encoding is set to *"UTF-8"*. Encoding defines how the characters are represented by the computer. This could be problematic for different alphabets or special characters (e.g., accented characters). The UTF-8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



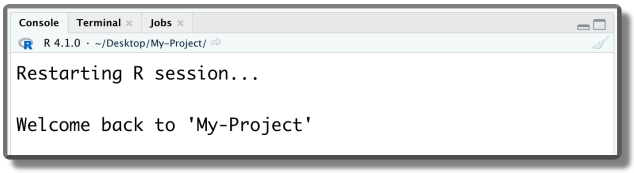
- **`.Rprofile` File.** This file is a special script that is automatically executed at the beginning of each session (or when the session is restarted). This file is not available by default but we can create it by naming an R script as `.Rprofile` (**without extension!**) [TODO: check in windows https://stackoverflow.com/questions/28664852/saving-a-file-as-rpr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

- **`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 **??**, 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
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.

4.1 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.

4.1.1 Run the Analysis

In Chapter **??**, 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
```

4.1.2 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 **??**, 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!

4.1.3 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 **??** and Chapter **??**, we discuss how to manage project requirements to guarantee results reproducibility.

4.1.4 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 4.1.4.1, we introduce Make, a Unix utility that allows us to automate tasks execution for general purposes. In Section 4.3, we present specific solutions for the R programming language.

4.1.4.1 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 4.3, 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.

4.2 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.

4.2.1 Analysis Workflow

In Chapter **??**, 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
```

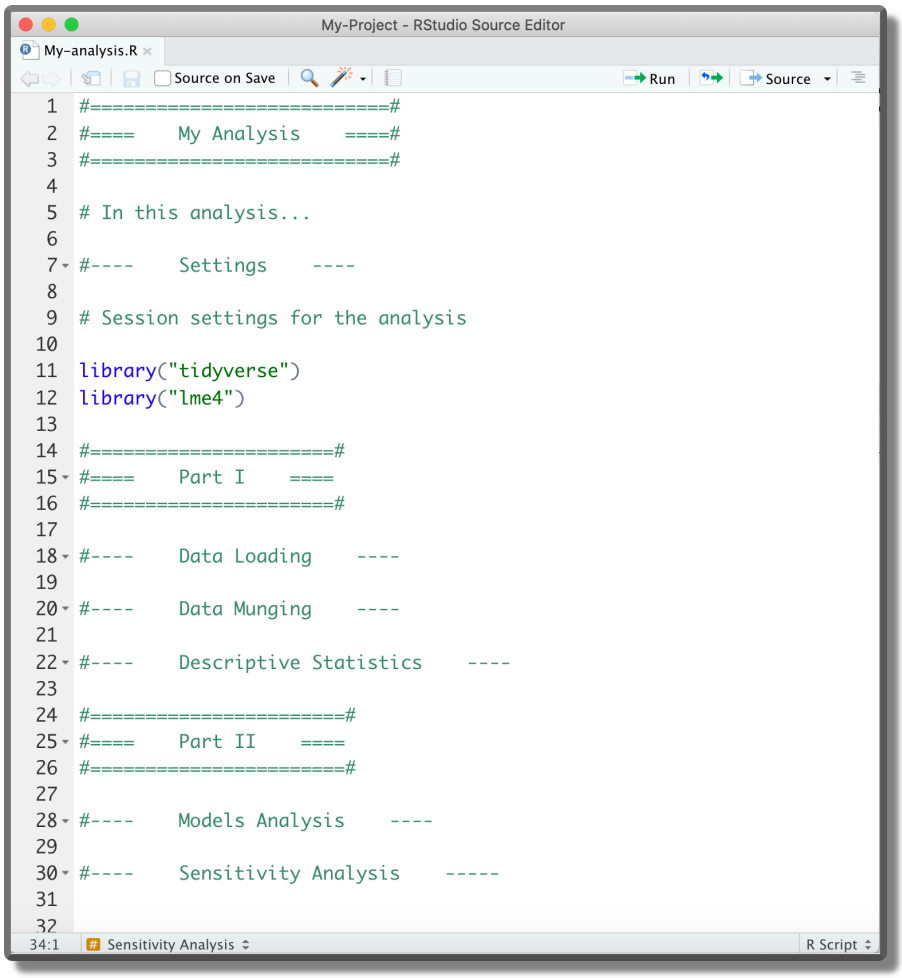
```
    |     |-- 04-comparison-experiments.R
    |     |-- 05-sensitivity-analysis.R
    |     |-- main.R
    |-- R/
    |     |-- data-munging.R
    |     |-- models.R
    |     |-- plots-tables.R
    |     |-- utils.R
```

4.2.1.1 Script Sections

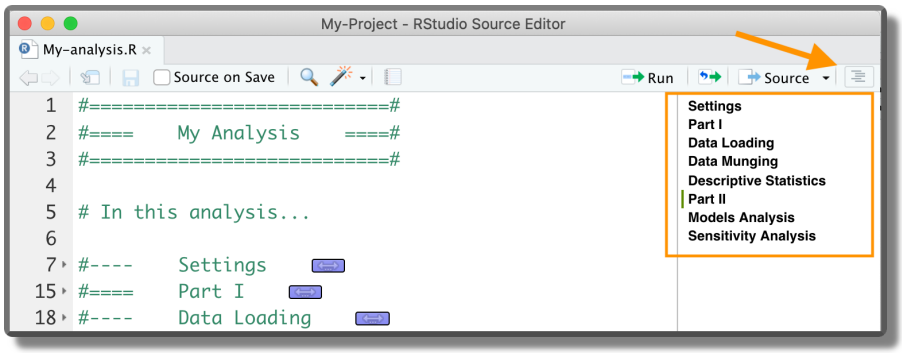
To enhance the readability of the analysis scripts, we can divide the code into sections. In RStudio, it is possible to create sections adding at the end of a comment line four (or more) consecutive symbols `####` (alternatively, `----` or `====`).

```
# Section 1 ####

# Section 2 ----

#----    Section 3    ----

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

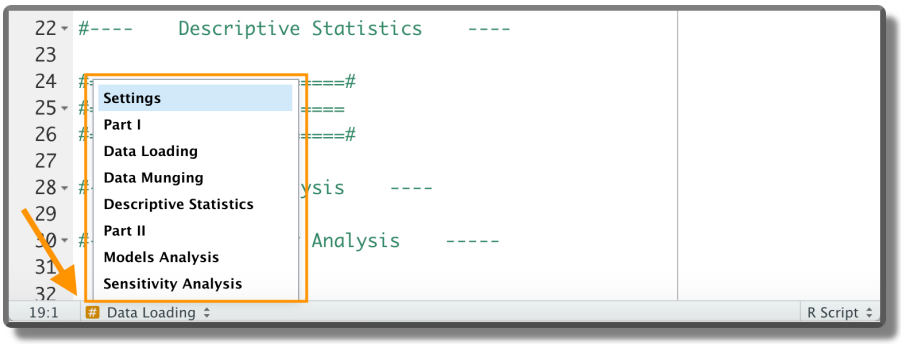
Using the available characters, it is possible to create different styles. The important thing is to finish the line with four (or more) identical symbols. As an example, we could organize our script as presented below



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.



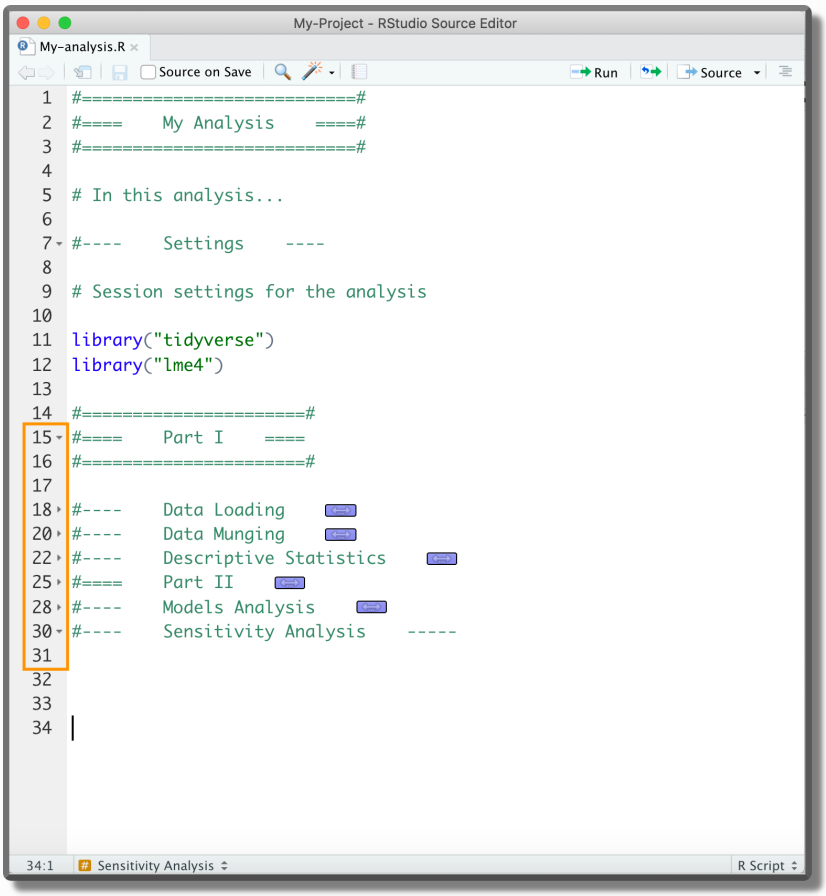
Another navigation bar is also available at the bottom left corner.



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.



4.2.1.2 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 **??**. 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 4.3) automatically runs the code in a new, non-interactive session.

4.2.1.3 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 4.2.1.2), 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()`

4.2.1.4 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 **??**, 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

4.2.2 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.

4.3 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.

4.3.1 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/`.

4.3.1.1 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 4.3.3.1 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.

4.3.1.2 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.

4.3.1.3 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).

4.3.2 The targets Workflow

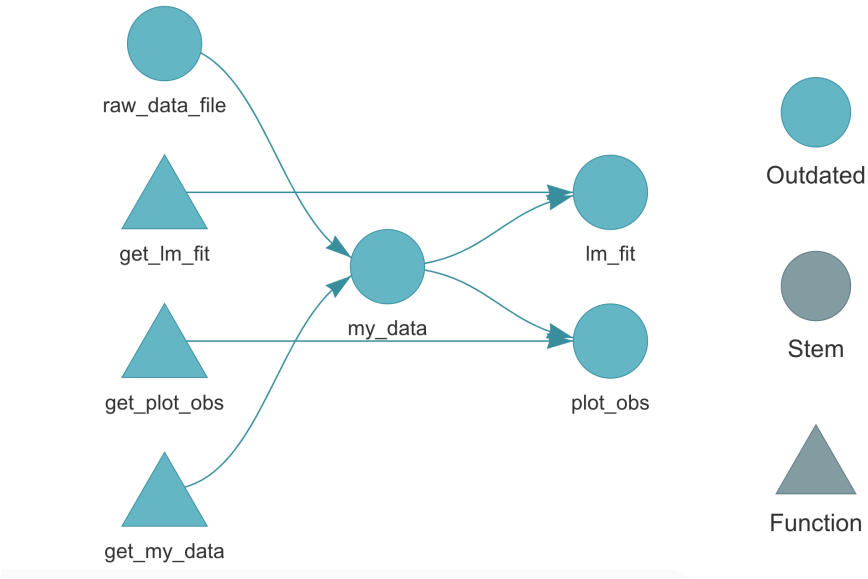
At this point, we have everything we need to run the analysis. Let's start the `targets` workflow.

4.3.2.1 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.

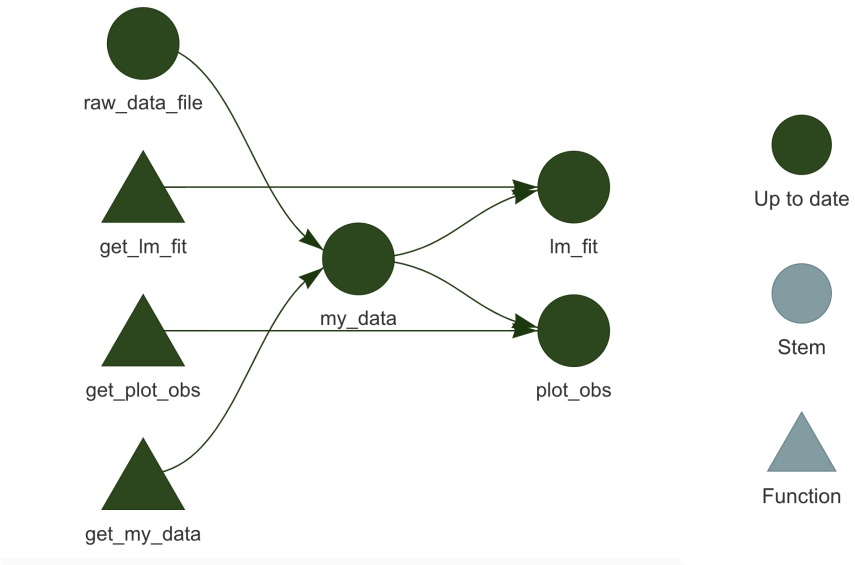


4.3.2.2 Run the Pipeline

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.

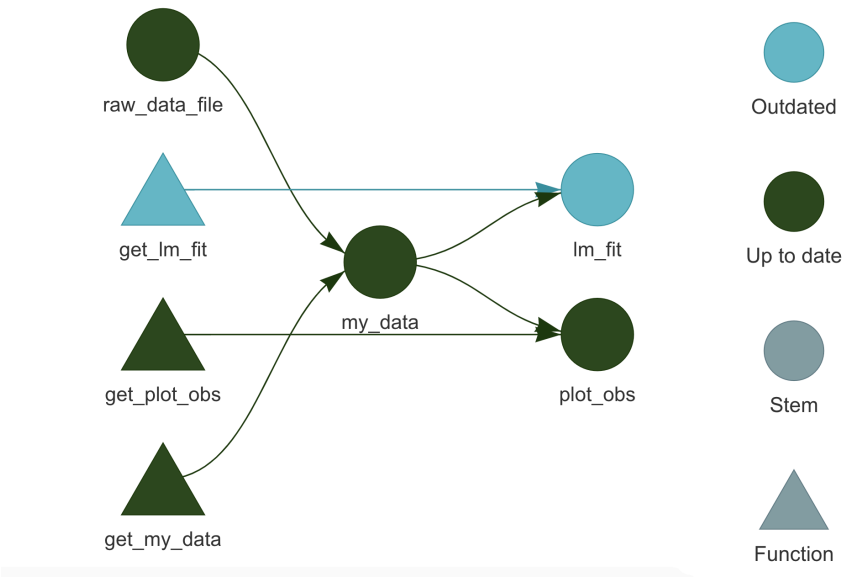
```
tar_make()
## * start target raw_data_file
## * built target raw_data_file
## * start target my_data
## * built target my_data
## * start target plot_obs
## * built target plot_obs
## * start target lm_fit
## * built target lm_fit
## * end pipeline
```

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



4.3.2.3 Make Changes

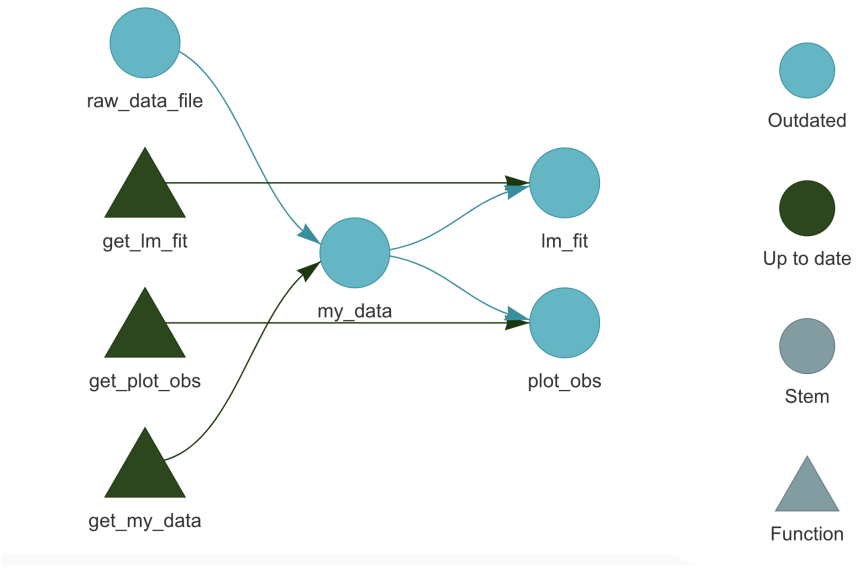
Let's say we make some changes to the function used to fit the linear model. `targets` will notice that and it will identify the invalidated targets that require to be updated. Looking at the `targets::tar_visnetwork()` graph, we can see which targets are affected by the changes made.



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.



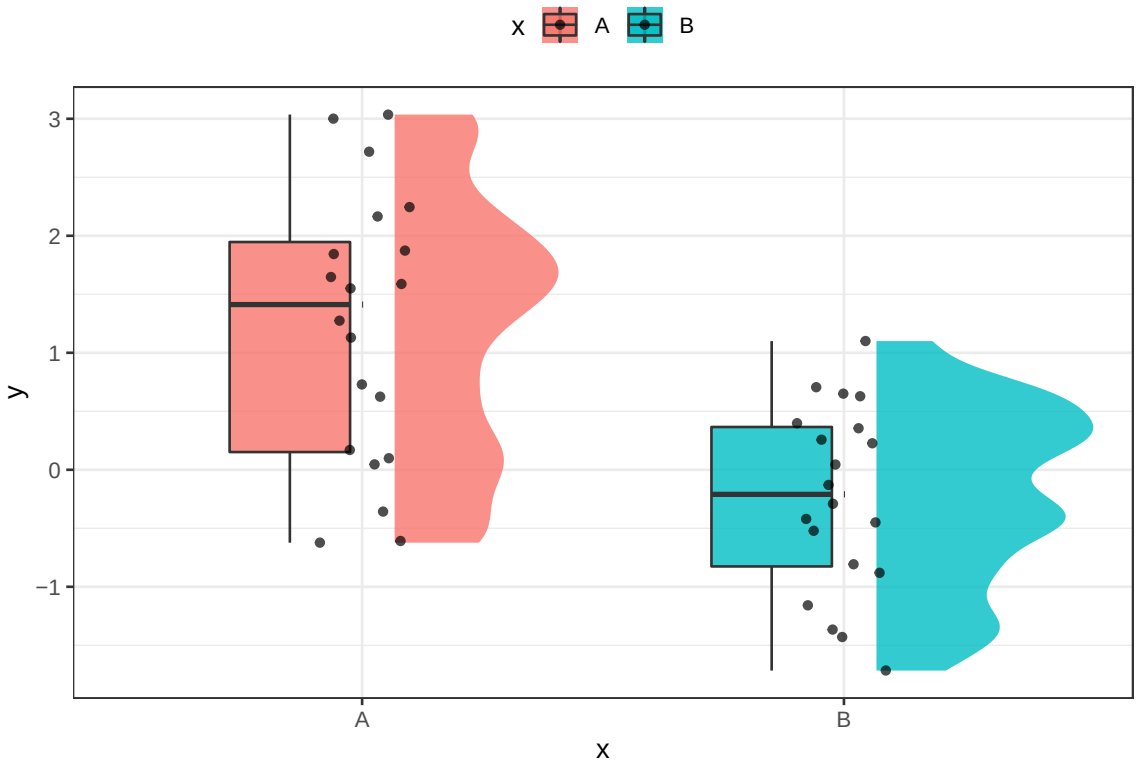
4.3.2.4 Get the Results

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
```

4.3.3 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.

4.3.3.1 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 4.3.1.1.
- **`analysis/_targets/`.** The directory where `targets` stores all the results and pipeline information. See Section 4.3.1.3.
- **`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/
```

4.3.3.2 `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.

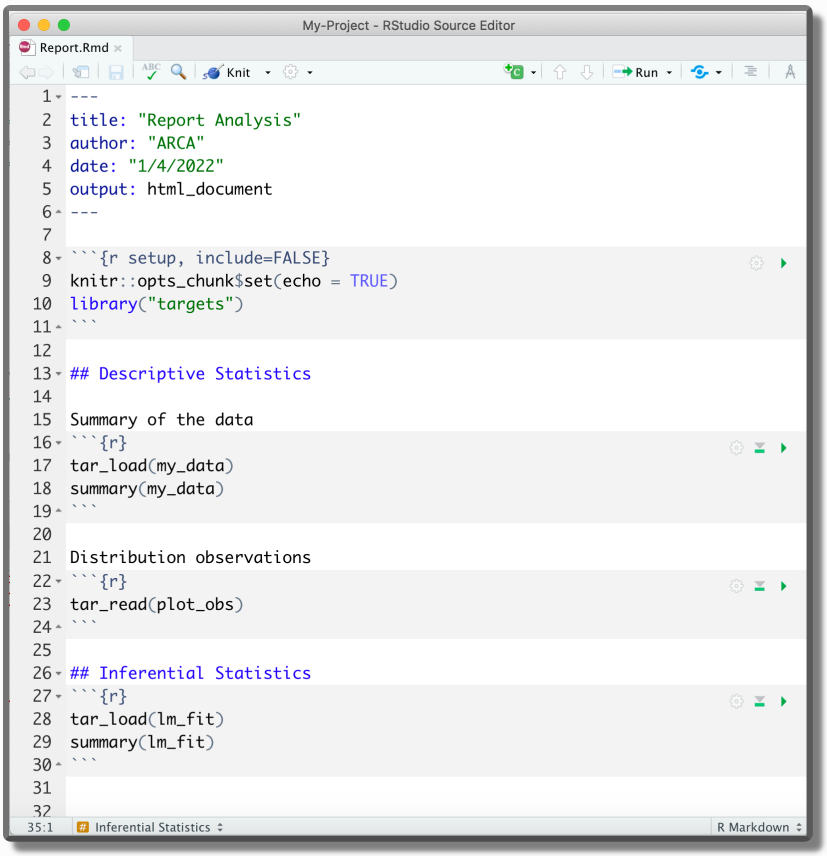
Among the two approaches, we recommend the second one. Using R Markdown documents as primary scripts to manage the analysis is fine in the case of simple projects. However, in the case of more complex projects, it is better to keep the actual analysis and the presentation of the results separate. In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

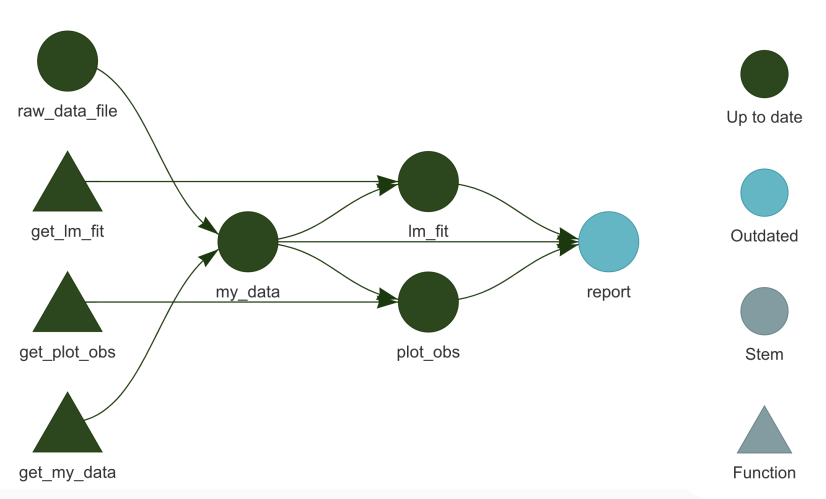


Next, we add it to the workflow pipeline,

```
#----    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/
```

4.3.3.3 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 4.3.1.1). 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 **??**). `targets` and `renv` can be used together in the same project workflow without problems.

4.3.3.4 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).

4.3.3.5 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.

4.3.3.6 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`.

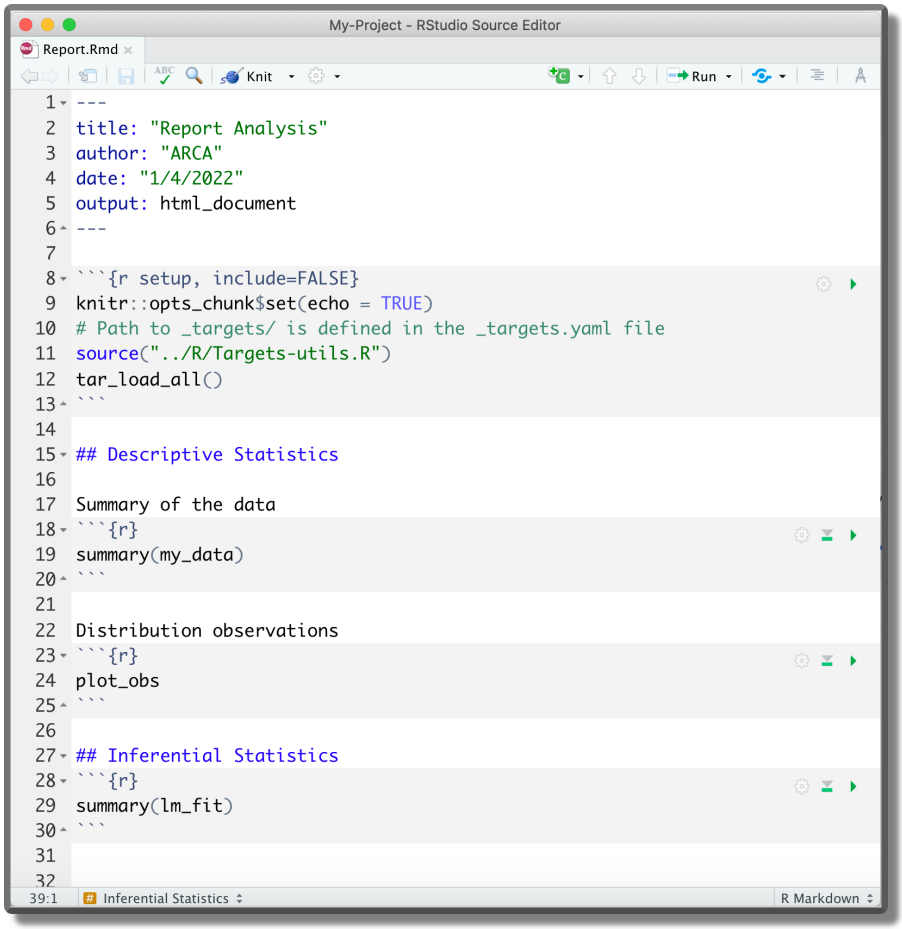
```
#----    R/Targets-utils.R    ----#

#----    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"))
}
```

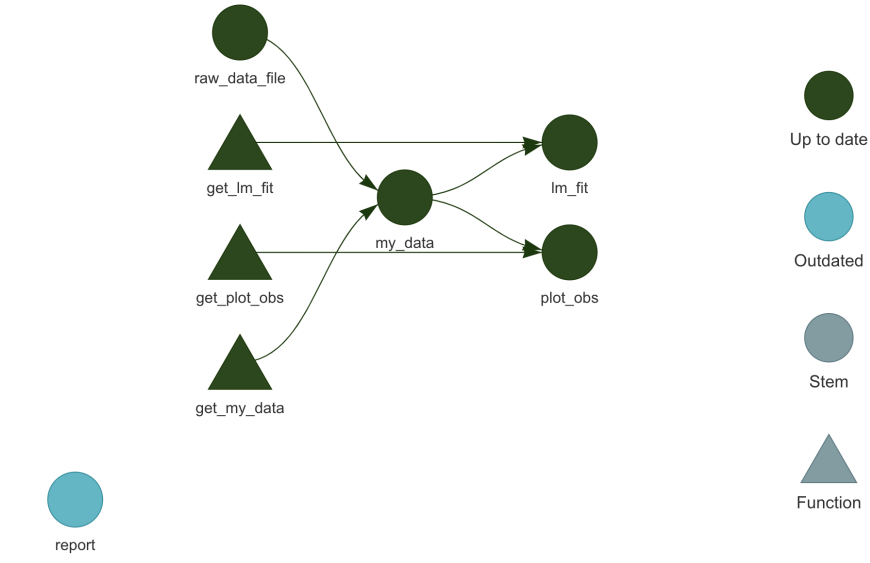
Where:

- **`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

References

Blischak, J., Carbonetto, P., & Stephens, M. (2021). *Workflowr: A framework for reproducible and collaborative data science*. https://github.com/workflowr/workflowr

Elliott, F., Errington, T. M., Bagby, C., Frazier, M., Geiger, B. J., Liu, Y., Mellor, D. T., Nosek, B. A., Pfeiffer, N., Tordoff, J. E., & al, et. (2021). *OSF*. OSF. osf.io/4znzp

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

Wickham, H. (2021). *Conflicted: An alternative conflict resolution strategy*. https://CRAN.R-project.org/package=conflicted

Wolen, A., & Hartgerink, C. (2020). *Osfr: Interface to the open science framework (OSF)*. https://CRAN.R-project.org/package=osfr