Reproducible Science

Margherita Calderan

Replicability School

June 6, 2025

About me 👋

- Post-doctoral researcher in **Cognitive Psychology**, University of Padova.

- Research: Computational modeling of **cognitive** and **learning processes**, **Bayesian** hypothesis testing.

- PhD in Psychological Science, completed **March 6, 2025**.

- Passionate about reproducible science after struggling with disorganized datasets in my early research!

Our job is hard 🔥

- Running experiments

- Analyzing data

- Managing trainees

- Writing papers

- Responding to reviewers

Reproducibility helps!

- **Organizes** your workflow.

- **Saves time** by documenting steps.

- **Builds trust** in your findings.

- Enables others to **reproduce** and **extend** your work.

What is reproducible science?

At its core, reproducible science means that someone else, or even you, in the future, can **reproduce** your **results** from your **materials**: your data, your code, your documentation.

It means your workflow is **transparent**.

Keys to reproducible science 🔐

- **Data**: organize, document, and share your datasets in ways that are usable by others and understandable by you (even years later).

- **Code**: write analysis scripts that are clean, transparent, and reusable..

- **Literate programming**: combine code and text in the same document, so your reports are dynamic and replicable.

- **Version Control and Sharing**: track changes, collaborate, and make your work openly available using tools like GitHub and OSF.

So... Is reproducible science even harder?

At first, yes - but then…🧯🔥

- Helps you stay **organized**.

- Makes it **easier to remember** what you did.

- Allows others to **understand**, **reproduce**, and **build on** your work.

> *Learning the tools takes effort but once you do, your workflow becomes smoother, clearer, and more reliable.*

Outline

Data

Code

R projects

Literate Programming

Version Control

Data

Data types in research

Open Science Framework

- Free platform to **organize**, **document**, and **share** research.

- Supports **preregistration**, **archiving**, and **collaboration**.

- Integrates with GitHub, Dropbox, Google Drive.

Bad data sharing example

Imagine this scenario: you read a paper that seems really relevant to your research. At the end, you're excited to see they've shared their data on OSF. You go to the repository, and there's one file…

Bad data sharing example

You download it, open it, and you see this: . . .

x1	x2	x3	x4	x5	x6	x7
0.3981105	13.912435	a	0	-0.6775811	0.8759740	-0.2051604
-0.1434733	1.093743	c	0	0.7055193	0.2521987	1.8816947
-0.2526000	4.898035	c	0	0.4744651	-0.5628840	0.3245589
-1.2272588	14.717053	b	0	-0.5132792	-1.1368242	-0.1355150
-0.4360417	8.547025	c	1	-0.1736804	-0.7120962	-1.2714320

What do these variables mean? What's x3?

What do 0 and 1 represent? How are missing values coded?

Is x6 a z-score or raw data?

Good data sharing practices

- Use **plain-text formats** (e.g., `.csv`, `.txt`).

- Include a **data dictionary** with variable descriptions.

- Add a **README** with key details.

- Follow **FAIR principles** (Findable, Accessible, Interoperable, Reusable).

Data dictionary 📂

- A **data dictionary** defines each variable in your dataset.

- Boosts **transparency** and **collaboration**.

- Saves time for **collaborators** and **future-you**.

datadictionary 📦

```r
1  library(datadictionary)
2  df <- data.frame( id = factor(letters[1:5]),
3                    anxi = rnorm(5, 0, 1),
4                    edu = factor(c("PhD", "BSc", "MSc", "PhD", "BSc")))
5  df_labels <- list(
6    anxi = "Beck Anxiety Inventory, standardized",
7    edu = "Last degree obtained"
8  )
9  create_dictionary(df, id_var = "id", var_labels = df_labels)
```

	item	label	class	summary	value
1				Rows in dataset	5
2				Columns in dataset	3
3	id	Unique identifier		unique values	5
4				missing	0
5	anxi	Beck Anxiety Inventory, standardized	numeric	mean	0
6				median	0
7				min	-1.39
8				max	2.16
9				missing	0
10	edu	Last degree obtained	factor	BSc (1)	2
11				MSc (2)	1
12				PhD (3)	2
13				missing	0

Data dictionary - Good data sharing example

The daily costs of workaholism

Data-dictionary.txt

```
- `ID`: Factor indexing participants identification codes (e.g., "S001", "S002", etc.)

- `day`: Integer indexing the day of participation (from 1 to 10)

- `SBP_aft`, `DBP_aft`: Numeric indexing afternoon measurements (mmHg) of systolic and diastolic blood pressure, respectively. Note: these
  variables have been computed by averaging each pair of consecutive recordings

- `WHLSM1` ... `WHLSM6`: Item scores at the state workaholism measure (1–7) administered in the afternoon

- `SBP_eve`, `DBP_eve`: Numeric indexing evening measurements (mmHg) of systolic and diastolic blood pressure, respectively. Note: these
  variables have been computed by averaging each pair of consecutive recordings

- `EE1` ... `EE4`: Item scores at the emotional exhaustion measure (1–7) administered in the evening

- `R.det1` ... `R.det3`: Item scores at the psychological detachment measure (1–7) administered in the evening

- `SQ1` ... `SQ4`: Item scores at the sleep disturbances measure (1–7) administered in the morning. Note: higher values indicate worse
  sleep disturbances; these measures are related to the following day

- `gender`: Factor indexing participants' gender ("F" or "M")

- `age`: Integer indexing participants' age (years)

- `BMI`: Numeric indexing participants' body mass index ($kg/m^2$)

- `IN.resp`: Factor indexing whether the participant was included in the main sample based on the response rate inclusion criteria (TRUE)
  or not (FALSE)

- `IN.bp`: Factor indexing whether the participant was included in the sample considered for blood pressure analyses (TRUE) or not (FALSE)
```

README files

A **README** file is the first thing someone sees when they open your dataset/project folder. It should answer basic questions like:

- What is this dataset?

- How was it collected?

- What are the variables?

README - Good data sharing example

Wearing face masks when no longer mandatory: An exploratory study about a...

README.txt ⋮

This repository "data" contains a CSV file that serves as the data source for the network analysis and regression analysis discussed in the accompanying article. Each column within the CSV file corresponds to a node in the network analysis.

Run first the Network_analysis.R script and then the "Generalized linear mixed effect modeling.R script.

README - Good data sharing example

README.md ⋮

Tracking the Unconscious: Neural evidence for the retention of unaware information in visual working memory

This repository contains the code to reproduce the analysis, figures and tables of the paper *Tracking the Unconscious: Neural evidence for the retention of unaware information in visual working memory* by Gambarota et al. The repository contains also the code to run the experiment using Psychopy along with raw EEG and behavioral data.

Structure

- The `data` folder contains the raw and cleaned EEG and behavioral data:

<!-- -->

```
data
├── clean
│   ├── behavioral
│   └── eeg
└── raw
    ├── behavioral
    │   ├── csv
    │   └── pkl
    └── eeg
```

- The `experiment/` folder contains python scripts to run the experiment.

<!-- -->

```
experiment
├── experiment.py
└── triggers.py
```

FAIR data principles 🔍

- **Findable**: Use metadata and DOIs to make data easy to locate.

- **Accessible**: Ensure data is retrievable via open repositories.

- **Interoperable**: Use standard formats (e.g., `.csv`, `.txt`) for compatibility.

- **Reusable**: Include clear documentation and open licenses.

Findable (DOI, Metadata)	Accessible (Open Repos)	Interoperable (.csv, Standards)	Reusable (Docs, Licenses)

FAIR
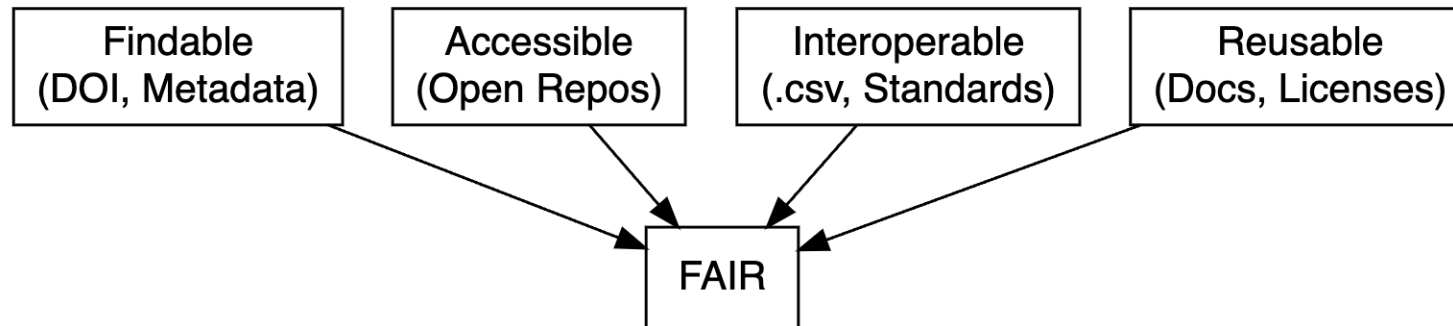
Data licensing 🔒

A license tells others what they can and can't do with your data. If you don't include one, legally speaking, people might not be allowed to use it, even if you meant to share it openly.

- **Licenses** clarify how others can use your data.

- Common licenses:

  - CC BY: Requires attribution.

  - CC0: No restrictions.

  - GNU-GPL: guarantees end users the freedom to run, study, share, and modify the software while requiring that all modified versions and derivative works also be distributed under the same license. ❤️

Code

Why scripting?

Scripting ensures **transparent** and **reproducible** workflows.

Reproducible: You can rerun them.

Documented: You can see what you did and when.

Shareable: Others can inspect and reproduce your analysis.

The SPSS Workflow

- Click menu items to run analysis

- "exclude <18"

- Click through everything again

- Forget a step? Round differently?

Stressful, error-prone, and undocumented.

R Workflow

```r
1   # Load data
2   data <- read.csv("data.csv")
3
4   # Filter age
5   data <- data[data$age >= 18, ]
6
7   # Analyze
8   summary(lm(score ~ condition, data = data))
9
10  # Make plot
11  ggplot(data, aes(x = condition, y = score)) +
12    geom_boxplot()
```

One line change, rerun, and everything updates.

R and RStudio 💻

- **R**: Free, open-source, with thousands of packages for analysis.

- **RStudio**: Intuitive interface for coding, plotting, and debugging.

- Vibrant **community** for support and resources.

Writing better code 📝

- **Name descriptively**: Use `snake_case` or `camelCase` for readability.

- **Comment clearly**: Document your logic for clarity.

- **Organize scripts**: Load packages and data upfront.

Use descriptive names

Another best practice: name your variables clearly.

```
1  # Bad
2  x1 <- c("unipd_psychology", "unipd_medicine", "unito_biology")
3  # Better
4  uni_dep <- c("unipd_psychology", "unipd_medicine", "unito_biology")
```

Consistency helps too. Use either snake_case or camelCase, but pick one and stick to it.

Comments, comments and comments...

Write the code for your future self and for others, not for yourself right now.

Try to open a (not well documented) old coding project after a couple of years and you will understand :)

Invest time in writing more comprehensible and documented code for you and others.

```r
1  # Remove participants with missing anxiety scores
2  dat <- dat %>% filter(!is.na(anxi))
3  #vs.
4  dat <- dat %>% filter(!is.na(anxi))  # <-- What is this doing?
```

Organized scripts

Global operations at the beginning of the script:

- loading packages

- loading datasets

- changing general options (`options()`)

```r
1  # packages
2  library(tidyverse)
3  library(lme4)
4
5  # options
6
7  options(scipen = 999)
8
9  # loading data
10 dat <- read.csv(...)
```

Summary

- Scripts beat point-and-click

- Structure matters

- Comment often

- Name things well

Functions to avoid repetition

Functions are the primary building blocks of your program. You write small, reusable, self-contained functions that do one thing well, and then you combine them.

Avoid repeating the same operation multiple times in the script. The rule is, if you are doing the same operation more than two times, write a function.

A function can be re-used, tested and changed just one time affecting the whole project.

Functional Programming, example...

We have a dataset (`mtcars`) and we want to calculate the mean, median, standard deviation, minimum and maximum of each column and store the result in a table.

```
1  head(mtcars, n = 3)
```

```
               mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4      21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag  21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710     22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
```

```
1  str(mtcars)
```

```
'data.frame':    32 obs. of   11 variables:
 $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
 $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
 $ disp: num  160 160 108 258 360 ...
 $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
 $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
 $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
 $ qsec: num  16.5 17 18.6 19.4 17 ...
 $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
 $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
 $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
 $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

Imperative Programming

The standard (~imperative) option is using a `for` loop, iterating through columns, calculate the values and store into another data structure.

```r
 1  ncols <- ncol(mtcars)
 2  means <- medians <- mins <- maxs <- rep(0, ncols)
 3
 4  for(i in 1:ncols){
 5    means[i] <- mean(mtcars[[i]])
 6    medians[i] <- median(mtcars[[i]])
 7    mins[i] <- min(mtcars[[i]])
 8    maxs[i] <- max(mtcars[[i]])
 9  }
10
11  results <- data.frame(means, medians, mins, maxs)
12  results$col <- names(mtcars)
13
14  head(results, n = 3)
```

```
    means medians mins  maxs  col
1  20.09062    19.2 10.4  33.9  mpg
2   6.18750     6.0  4.0   8.0  cyl
3 230.72188   196.3 71.1 472.0 disp
```

Functional Programming

The main idea is to decompose the problem writing a function and loop over the columns of the dataframe:

```r
1  summ <- function(x){
2    data.frame(means = mean(x),
3               medians = median(x),
4               mins = min(x),
5               maxs = max(x))
6  }
7  ncols <- ncol(mtcars)
8  dfs <- vector(mode = "list", length = ncols)
9
10 for(i in 1:ncols){
11   dfs[[i]] <- summ(mtcars[[i]])
12 }
```

Functional Programming

```r
results <- do.call(rbind, dfs)
head(results, n = 6)
```

```
    means  medians    mins    maxs
1  20.090625  19.200  10.400  33.900
2   6.187500   6.000   4.000   8.000
3 230.721875 196.300  71.100 472.000
4 146.687500 123.000  52.000 335.000
5   3.596563   3.695   2.760   4.930
6   3.217250   3.325   1.513   5.424
```

Functional Programming, *apply 📦

- The *apply family is one of the best tool in R. The idea is pretty simple: apply a function to each element of a list.

- The powerful side is that in R everything can be considered as a list. A vector is a list of single elements, a dataframe is a list of columns etc.

- Internally, R is still using a for loop but the verbose part (preallocation, choosing the iterator, indexing) is encapsulated into the *apply function.

```r
1  means <- rep(0, ncol(mtcars))
2  for(i in 1:length(means)){
3    means[i] <- mean(mtcars[[i]])
4  }
5
6  # the same with sapply
7  means <- sapply(mtcars, mean)
```

The *apply Family

Apply **your** function…

```
1   results <- lapply(mtcars, summ)
```

Now results is a list of data frames, one per column.

We can stack them into one big data frame:

```
1   results_df <- do.call(rbind, results)
```

This gives us a clean summary for every variable in just a few lines of code. No loops, no repetition.

Using **sapply**, **vapply**, and **apply**

- `lapply()` always returns a list.

- `sapply()` tries to simplify the result into a vector or matrix.

- `vapply()` is like `sapply()` but safer (you specify the return type).

- `apply()` is for applying functions over rows or columns of a matrix or data frame.

```
1  sapply(mtcars, mean) == apply(mtcars, MARGIN = 2, mean) # MARGIN = 2, apply to columns
```

```
 mpg  cyl disp   hp drat   wt qsec   vs   am gear carb
TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

`for` loops are bad?

`for` loops are the core of each operation in R (and in every programming language). For complex operation thery are more readable and effective compared to `*apply`. In R we need extra care for writing efficent `for` loops.

Extremely slow, no preallocation:

```
1  res <- c()
2  for(i in 1:1000){
3    # do something
4    res[i] <- i^2
5  }
```

Very fast:

```
1  res <- rep(0, 1000)
2  for(i in 1:length(res)){
3    # do something
4    res[i] <- i^2
5  }
```

microbenchmark 📦

```r
library(microbenchmark)

microbenchmark(
  grow_in_loop = {
    res <- c()
    for (i in 1:10000) {
      res[i] <- i^2
    }
  },
  preallocated = {
    res <- rep(0, 10000)
    for (i in 1:length(res)) {
      res[i] <- i^2
    }
  }, times = 100)
```

```
Unit: microseconds
        expr      min        lq       mean    median        uq      max neval cld
grow_in_loop 1140.948 1225.982 1352.5666 1257.532 1320.651 2904.153   100   a
preallocated  655.139  683.183  784.2455  697.943  714.671 6966.351   100    b
```

Going further: custom function lists

Let's define a list of functions:

```
1  funs <- list(mean = mean, sd = sd, min = min, max = max, median = median)
```

Now we can apply all of these to every column:

```
1  sapply(funs, function(f) apply(mtcars, 2, f))
```

```
          mean          sd     min     max  median
mpg    20.090625   6.0269481  10.400  33.900  19.200
cyl     6.187500   1.7859216   4.000   8.000   6.000
disp  230.721875 123.9386938  71.100 472.000 196.300
hp    146.687500  68.5628685  52.000 335.000 123.000
drat    3.596563   0.5346787   2.760   4.930   3.695
wt      3.217250   0.9784574   1.513   5.424   3.325
qsec   17.848750   1.7869432  14.500  22.900  17.710
vs      0.437500   0.5040161   0.000   1.000   0.000
am      0.406250   0.4989909   0.000   1.000   0.000
gear    3.687500   0.7378041   3.000   5.000   4.000
carb    2.812500   1.6152000   1.000   8.000   2.000
```

This gives you a matrix with rows as variables and columns as statistics.

Pure vs. Impure functions

Pure function

Same input, same output, no side effects.

```r
1  x = 4
2  add_pure<- function(x) {
3    return(x + 1)
4  }
5  add_pure(x)
```

```
[1] 5
```

```r
1  print(x)
```

```
[1] 4
```

Impure function

Modifies external variables.

```r
1  x = 4
2  add_impure <- function(x) {
3    x <<- x + 1
4  }
5  add_impure(x)
6  print(x)
```

```
[1] 5
```

Test your functions - `fuzzr` 📦

When you write your own functions, it's smart to test them. In R, we can use `fuzzr` to do **property-based testing**.

Define your function…

```
1  my_mean <- function(x, na.rm = TRUE) {
2    if (!is.numeric(x)) stop("`x` must be numeri
3    if (length(x) == 0) return(NA)
4    if (na.rm) x <- x[!is.na(x)]
5    if (length(x) == 0) return(NA)
6    sum(x) / length(x)
7  }
```

Define properties that should always hold true…

```
1  property_mean_correct <- function(x) {
2    x_no_na <- x[!is.na(x)] #remove NA
3    if (length(x_no_na) == 0) return(TRUE)
4    abs(my_mean(x) - mean(x, na.rm = TRUE)) < 1e
5  }
```

This runs the property on different random numeric vectors and checks whether it holds.

```r
# Property-based testing with 'fuzzr'
library(fuzzr)
test = fuzz_function(fun = property_mean_correct,
                     arg_name = "x",
                     tests = test_dbl())
lapply(test, function(res) res$test_result$value)
```

```
[[1]]
[1] TRUE

[[2]]
[1] TRUE

[[3]]
[1] TRUE

[[4]]
[1] TRUE

[[5]]
[1] TRUE

[[6]]
[1] TRUE
```

```r
fuzzr::test_dbl()
```

```
$dbl_empty
numeric(0)

$dbl_single
[1] 1.5

$dbl_mutliple
[1] 1.5 2.5 3.5

$dbl_with_na
[1] 1.5 2.5  NA

$dbl_single_na
[1] NA

$dbl_all_na
[1] NA NA NA
```

Why functional programming?

- We can write less and reusable code that can be shared and used in multiple projects.

- The scripts are more compact, easy to modify and less error prone (imagine that you want to improve the `summ` function, you only need to change it once instead of touching the `for` loop).

- Functions can be easily and consistently documented (see roxygen documentation) improving the reproducibility and readability of your code.

Functional programming in the wild

You can write some R scripts only with functions and `source()` them into the global environment.

```
project/
├── R/
│   ├── utils.R
│   ├── analysis.R
```

```
1  source("R/utils.R")
2  results <- lapply(mtcars, summ)
3  results_df <- do.call(rbind, results)
```

This is reproducible, modular, and maintainable.

More about functional programming in R

- Advanced R by Hadley Wickham, section on Functional Programming (https://adv-r.hadley.nz/fp.html)

- Hands-On Programming with R by Garrett Grolemund https://rstudio-education.github.io/hopr/

- Hadley Wickham: The Joy of Functional Programming (for Data Science) (https://www.youtube.com/watch?v=bzUmK0Y07ck)

Advanced R

Second Edition

Hadley Wickham

Hands-On Programming with R

WRITE YOUR OWN FUNCTIONS AND SIMULATIONS

Garrett Grolemund

Foreword by Hadley Wickham
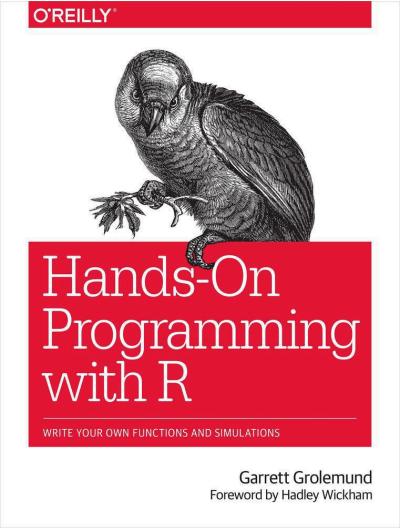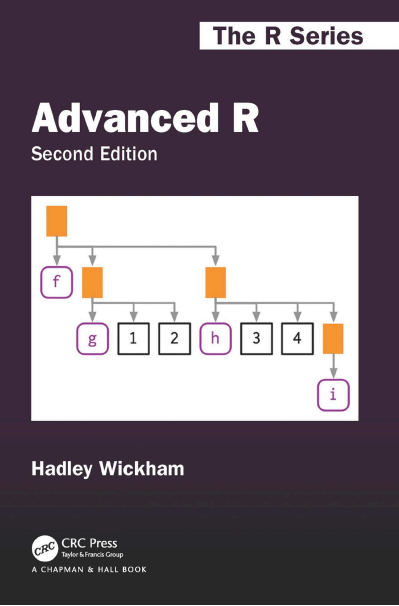
Wrapping up

- Avoid repetition by using functions.

- Favor pure functions.

- Test your functions.

- The `*apply` functions are your friends, or `*map` from `purrr` 📦

Organize your project

R Projects

R Projects are a feature implemented in RStudio to organize a working directory.

- They automatically set the working directory

- They allow the use of **relative paths** instead of **absolute paths**

- They provide quick access to a specific project

The Working Directory Problem

How many times have you opened an R script and seen this at the top?

setwd("C:/Users/margherita/Documents/PhD/final_data/mess") *#change*

Instead of hardcoding paths, we want to use projects with **relative paths**.

R Projects

An R Project (.Rproj) is a file that defines a self-contained workspace. When you open an R Project, your working directory is automatically set to the project root, no need to use setwd() ever again.

Open RStudio

```
File → New Project → New Directory → New Project
```

Relative Path (to the working directory)

Absolute path: read.csv("Users/tita/workinMemo/data/clean_data.csv")

Relative path: read.csv("data/clean_data.csv")

A Minimal Project Structure

```
1    my-project/
2        |
3        ├── data/
4        |       ├── raw/
5        |       └── processed/
6        ├── R/
7        |    └── analysis.R
8        |
9        ├── outputs/
10       |       ├── figures/
11       |       └── tables/
12       |
13       ├── my-project.Rproj
14       |
15       └── README.md
```

Project organization with `rrtools` 📦

To make this even "easier", you can use the `rrtools` package to create what's called a reproducible research compendium.

> … the goal is to provide a standard and easily recognisable way for organising the digital materials of a project to enable others to inspect, reproduce, and extend the research… (*Marwick et al., 2018*)
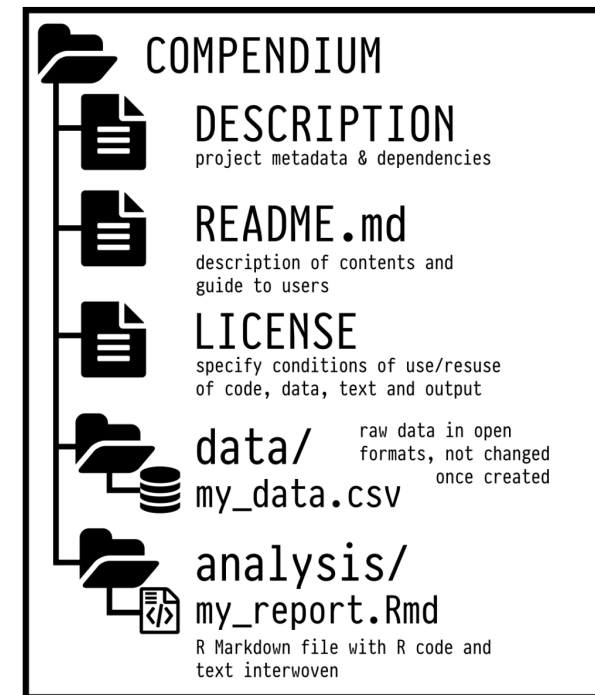
Research compendium `rrtools` 📦

- Organize its files according to the prevailing conventions.

- Maintain a clear separation of data (original data is untouched!), method, and output.

- Specify the computational environment that was used for the original analysis

> Tutorial

`rrtools::create_compendium()` builds the basic structure for a research compendium.

> Example

renv📦 : locking your R environment

Another challenge for reproducibility is package versions.

You write some code today using `dplyr 1.1.2`.

In six months, `dplyr` gets updated… 😢

`renv` helps you create reproducible environments for your R projects!

What does **renv** do?

- It records all the packages you use, with versions, in a lockfile

- It installs them in a project-specific library

- It ensures that anyone who runs your code gets exactly the same environment

Project specific library

```
install.packages("renv")

renv::init()

install.packages('bayesplot')

These packages will be installed into "~/repro-pre-school/example-renv/renv/library/macos/R-
4.4/aarch64-apple-darwin20".
```

> Example

renv commands
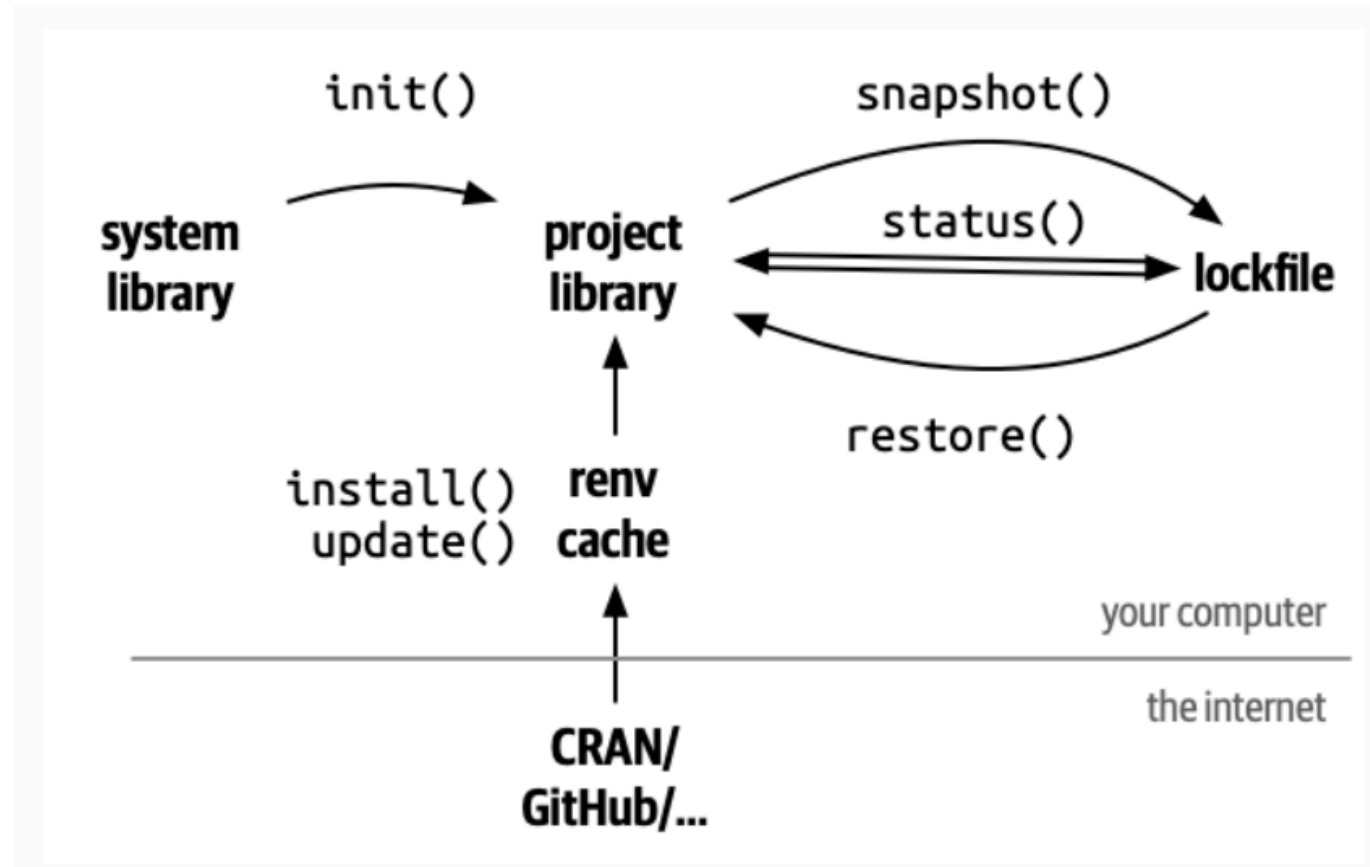
`renv::snapshot()` # update lockfile

`renv::restore()` # re-install from lockfile

Research **rrtools** + **renv** 💣

- **rrtools**: Organizes your project into a **reproducible compendium** with clear folders.

- **renv**: Locks **R package versions** for consistent environments.

- Together, they ensure **structure** and **reproducibility** across teams and time.

- Run `rrtools::create_compendium()` to start, then `renv::init()` to lock dependencies.

Docker

- Packages your project's **software**, **dependencies**, and **system settings** into a *container*.

- Ensures **consistency** across different computers or servers.

- Ideal for **sharing** complex analyses with others.

Documenting your environment ℹ️

- **`sessionInfo()`**: Captures your **R version**, **packages**, and **platform** in one command.

- Easy way to **document** and **share** your environment.

```
1  sessionInfo()
```

```
R version 4.4.2 (2024-10-31)
Platform: aarch64-apple-darwin20
Running under: macOS Sequoia 15.5

Matrix products: default
BLAS:   /Library/Frameworks/R.framework/Versions/4.4-arm64/Resources/lib/libRblas.0.dylib
LAPACK: /Library/Frameworks/R.framework/Versions/4.4-arm64/Resources/lib/libRlapack.dylib;  LAPACK
version 3.12.0

locale:
[1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8

time zone: Europe/Rome
tzcode source: internal

attached base packages:
[1] stats       graphics  grDevices utils     datasets  methods   base
```

Organizing for reproducibility

- Don't hardcode paths, use `R Projects`

- Create a logical folder structure for your project

- Use `rrtools` to scaffold a research compendium

- Use `renv` to lock your package versions

All of this structure isn't just for you, it makes it easier to work with others.

Literate Programming

What's wrong about Microsoft Word?

MS Word is a WYSIWYG (*what you see is what you get editor*) that force users to think about formatting, numbering, etc. Markup languages receive the content (plain text) and the rules and creates the final document.

The entire Microsoft word document when you slightly move an image by 1 mm

Moving a picture in Microsoft Word

It actually does what you want

You the

mess up whole document

Problems beyond formatting

Beyond the pure writing process, there are other aspects related to research data.

Think about the typical MW workflow

- You run your analysis in R

- You copy the results into a Word document

- You tweak the formatting

- You insert a figure generated with R manually

- You change your analysis, but forget to update the results in the text…

Literate Programming

A document where:

- The code is part of the text

- The results are generated *dynamically*

- The figures are rendered *automatically*

- Everything is in *sync*

For example **jupyter notebooks**, **R Markdown** and now **Quarto** are literate programming frameworks to integrate code and text.

Literate Programming, the markup language

Beyond the coding part, the markup language is the core element of a literate programming framework.

When you write in a markup language, you're writing **plain text** while also giving **instructions** for how to generate the final result.

- LaTeX

- HTML

- Markdown

- …

LaTeX

```latex
1  % This is a simple sample document.  For more complicated documents take a look
   in the exercise tab. Note that everything that comes after a % symbol is treated
   as comment and ignored when the code is compiled.
2
3  \documentclass{article} % \documentclass{} is the first command in any LaTeX
   code.  It is used to define what kind of document you are creating such as an
   article or a book, and begins the document preamble
4
5  \usepackage{amsmath} % \usepackage is a command that allows you to add
   functionality to your LaTeX code
6
7  \title{Simple Sample} % Sets article title
8  \author{My Name} % Sets authors name
9  \date{\today} % Sets date for date compiled
10
11 % The preamble ends with the command \begin{document}
12 \begin{document} % All begin commands must be paired with an end command
   somewhere
13     \maketitle % creates title using information in preamble (title, author,
       date)
14
15     \section{Hello World!} % creates a section
16
17     \textbf{Hello World!} Today I am learning \LaTeX. %notice how the command
       will end at the first non-alphabet charecter such as the . after \LaTeX
18      \LaTeX{} is a great program for writing math. I can write in line math such
       as $a^2+b^2=c^2$ %$ tells LaTexX to compile as math
19      . I can also give equations their own space:
20     \begin{equation} % Creates an equation environment and is compiled as math
21     \gamma^2+\theta^2=\omega^2
22     \end{equation}
23     If I do not leave any blank lines \LaTeX{} will continue  this text without
       making it into a new paragraph.  Notice how there was no indentation in the
       text after equation (1).
24     Also notice how even though I hit enter after that sentence and here
       $\downarrow$
25      \LaTeX{} formats the sentence without any break.  Also   look   how      it
        doesn't    matter       how    many  spaces    I put     between
        my    words.
26
```

Simple Sample

My Name

July 4, 2024

1 Hello World!

Hello World! Today I am learning LaTeX. LaTeX is a great program for writing math. I can write in line math such as $a^2 + b^2 = c^2$. I can also give equations their own space:

$$\gamma^2 + \theta^2 = \omega^2 \tag{1}$$

If I do not leave any blank lines LaTeX will continue this text without making it into a new paragraph. Notice how there was no indentation in the text after equation (1). Also notice how even though I hit enter after that sentence and here $\downarrow$ LaTeX formats the sentence without any break. Also look how it doesn't matter how many spaces I put between my words.

For a new paragraph I can leave a blank space in my code.

Markdown

```
1  # Markdown syntax guide
2
3  ## Headers
4
5  # This is a Heading h1
6  ## This is a Heading h2
7  ###### This is a Heading h6
8
9  ## Emphasis
10
11 *This text will be italic*
12 _This will also be italic_
13
14 **This text will be bold**
15 __This will also be bold__
16
17 _You **can** combine them_
18
19 ## Lists
20
21 ### Unordered
```

Markdown syntax guide

Headers

This is a Heading h1

This is a Heading h2

This is a Heading h6

Emphasis

This text will be italic
This will also be italic

Markdown

Markdown is one of the most popular markup languages for several reasons:

- easy to write and read compared to Latex and HTML

- easy to convert from Markdown to basically every other format using `pandoc`

- easy to implement new features

Markdown (source code)

Also the source code can be used to take notes and read.

```
1  ## My Section
2  - Write **bold** text.
3  - Include a [link](https://quarto.org)
4  - Run code: `r mean(mtcars$mpg)`.
```

Latex and HTML need to be compiled otherwise they are very hard to read.

Quarto

Quarto (https://quarto.org/) is the evolution of R Markdown that integrate a programming language with the Markdown markup language. It is very simple but quite powerful.

Basic Markdown

Markdown can be learned in minutes. You can go to the following link https://quarto.org/docs/authoring/markdown-basics.html and try to understand the syntax.

Markdown Basics

Overview

Quarto is based on Pandoc and uses its variation of markdown as its underlying document syntax. Pandoc markdown is an extended and slightly revised version of John Gruber's Markdown syntax.

Markdown is a plain text format that is designed to be easy to write, and, even more importantly, easy to read:

> A Markdown-formatted document should be publishable as-is, as plain text, without looking like it's been marked up with tags or formatting instructions. – John Gruber

Quarto

You write your documents in **Markdown**, and **Quarto** turns them into:

- HTML reports

- PDF articles

- Word documents

- Slides

- Website

- Academic manuscripts

- …

Quarto

> Example

- If your data changes, your summary table updates.

- If you update your model, your coefficients update.

- If you change a plot's colors, the new version appear, without having to re-export and re-insert anything.

This eliminates a huge source of human error: **manual updates**.

Outputs

Quarto can generate multiple output formats from the same source file.

With one command, you get three outputs:

- A PDF to send to your colleagues

- A Word document for your co-author who hates PDFs

- An HTML report for your own website

Everything from the same source. No duplication. **Synchronization**.

> Example

Extra Tools: citations and cross-referencing

- Citations with BibTeX or Zotero

- Cross-references for figures and tables

- Numbered equations with LaTeX syntax

- Footnotes, tables of contents, and more

You can write scientific documents that look and behave just like journal articles, without ever opening Word.

Writing Papers - APA quarto

APA Quarto is a Quarto extension that makes it easy to write documents in APA 7th edition style, with automatic formatting for title pages, headings, citations, references, tables, and figures.

A Quarto Extension for Creating APA 7 Style Documents

lifecycle experimental

This article template creates APA Style 7th Edition documents in .docx, .html. and .pdf. The .pdf format can be rendered via Latex (i.e., apaquarto-pdf) or via Typst (apaquarto-typst). The Typst output for this extension is still experimental and requires Quarto 1.5 or greater.

Because the .docx format is still widely used—and often required—my main priority was to ensure compatibility for .docx. This is still a work in progress, and I encourage filing a "New Issue" on GitHub if something does not work of if there is a feature missing.

See instructions and template options for apaquarto here.

Version History

Example Outputs

The apaquarto-docx form looks like this:

TEMPLATE FOR THE APAQUARTO EXTENSION   1

Let's see an example...

> Example

Quarto + Zotero

Choose your reference:

More about Quarto and R Markdown

The topic is extremely vast. You can do everything in Quarto, a website, thesis, your CV, etc.

- Yihui Xie - R Markdown Cookbook https://bookdown.org/yihui/rmarkdown-cookbook/

- Yihui Xie - R Markdown: The Definitive Guide https://bookdown.org/yihui/rmarkdown/

- Quarto documentation https://quarto.org/docs/guide/

Version Control

Why Version Control?

You're working on a project. You save your script as:

- `analysis.R`

- `analysis2.R`

- `analysis_final.R`

- `analysis_final_revised.R`

- `analysis_final_revised_OK_for_real.R`

What is Git?

Git is a version control system. It works like a time machine for your project.

```
1  git init
```

Then, save changes with commits:

```
1  git add analysis.R
2  git commit -m "Added initial analysis"
```

Commit message ✍️

- Write meaningful messages:

- ✅ `"Fix bug in anxiety scoring function"`

- ❌ `"stuff"`

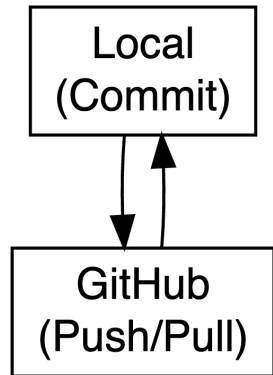- Use the imperative mood: `"Add README"`, `"Update plots"`

GitHub

Git works **locally**. GitHub is the **online platform** for:

- Backing up your project

- Sharing it publicly or privately

- Collaborating with others

- Tracking issues and progress

https://github.com

GitHub in Practice

Local
(Commit)

GitHub
(Push/Pull)

```
1  # 1. Initialize a Git repository in your current project folder
2  git init
3
4  # 2. Stage a file to be tracked (e.g., your script)
5  git add analysis.R
6
7  # 3. Save a snapshot of your work with a message
8  git commit -m "Initial commit"
9
10 # 4. Rename the default branch to 'main' (recommended)
11 git branch -M main
12
13 # 5. Connect your local project to a GitHub repo (change the URL)
14 git remote add origin https://github.com/yourname/repo.git
15
16 # 6. Upload your commits to GitHub
17 git push -u origin main
```

Branching & merging 🌱

- Try out new features

- Fix bugs safely

- Work on different versions in parallel

```
1   # Create and switch to a new branch called 'new-feature'
2   git checkout -b new-feature
3
4   # (Make your changes in code, then stage and commit them)
5   # Save those changes with a descriptive message
6   git commit -m "Add new plot"
7
8   # Switch back to the main branch
9   git checkout main
10
11  # Merge the changes from 'new-feature' into 'main'
12  git merge new-feature
```

Use branches to keep your `main` branch clean.

Handling conflicts

Sometimes, Git can't automatically merge changes. This happens when two branches modify the same line in a file.

```
1  <<<<<<< HEAD
2  plot(data)
3  =======
4  plot(data, col = "blue")
5  >>>>>>> new-feature
```

Git will insert conflict markers directly into the file:

The code between `<<<<<<< HEAD` and `=======` is from the current branch (e.g., `main`)

The code between `=======` and `>>>>>>> new-feature` is from the other branch you're merging (e.g., `new-feature`)

Handling conflicts

To resolve the conflict, choose the correct version (or combine them), delete the markers, and save the file.

For example:

```
1  plot(data, col = "blue")  # resolved version
```

Then:

```
1  git add file.R
2  git commit -m "Resolve merge conflict in file.R"
```

GitHub + RStudio Integration

- Clone repos with `File → New Project → Version Control`

- Commit and push from the **Git** tab in RStudio

- View commit history in **History** pane

Practice & resources

- Happy Git and GitHub for the useR

- GitHub Education: https://education.github.com

- Try GitHub Desktop (GUI client)

Start small. Use Git for one script. Then grow your skills from there.

If Git and GitHub feel too technical, or if your collaborators are less technical, the OSF is a fantastic alternative or complement.

- Upload data, code, and documents

- Create public or private projects

- Add collaborators

- Create preregistrations

- Generate DOIs for citation

- Track changes

> You can also connect OSF to GitHub.

Integrated workflow 🛠️

1. Develop your analysis using **R and Quarto**.

2. Track code and scripts using **Git**.

3. Host your code on **GitHub** (public or private).

4. Upload your data and materials to **OSF**, including a data dictionary.

5. Link your GitHub repository to your OSF project.

6. Use `renv` for reproducible R environments.

7. Share the OSF project and cite it in your paper.

Reproducibility

It's about **credibility** and **transparency**.

Reproducible science is **not** about being **perfect**.

It's about showing your work so that others can **follow**, **understand**, and **build upon** it.

Start simple 🐾

> "… anything you do — providing the raw data, posting any small scripts, detailing the versions of programs you used together with their parameters — will be tremendously welcome to anyone trying to validate or build off your paper…"

C. Titus Brown

Start simple, don't wait until you're "ready", and teach what you learn!

THANK YOU!

References

Marwick, B., Boettiger, C., & Mullen, L. (2018). Packaging data analytical work reproducibly using r (and friends). *The American Statistician*, *72*(1), 80–88. https://doi.org/10.1080/00031305.2017.1375986