Reproducible Science

Margherita Calderan
Replicability School

June 6, 2025

About me 👋

- I am a post-doctoral researcher in **Cognitive Psychology** at the Department of General Psychology, University of Padova.

- My research interests include the computational modeling of **cognitive** and **learning processes**, and **Bayesian** hypothesis testing.

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

Our job is hard 🔥

- Running experiments
- Conducting analyses
- Managing trainees
- Managing data
- Writing papers
- Preparing talks and abstracts
- Reading papers
- Responding to reviewers
- Collaborating with peers and supervisors

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.*

Keys to reproducible science

1. A general purpose programming language such as R or Python.

2. A literate programming framework to integrate code and text.

3. A version control system to track projects.

4. An online repository for future-proof sharing.

R and RStudio

- R packages allow to **do almost everything**.
- It is **free** and **open-source**.
- The **community** is wide, active thus solving problems is very easy.
- Force you to **learn scripting**.
- 

Writing better code

Naming variables

Be consistent and descriptive. Avoid cryptic names like `x1`. Use either:

```
x1 = rep(c("DPSS","DPG","DSS"), 4)  # What does 'x' mean?
DepUni = x1       # CamelCase
dep_uni = x1      # snake_case
```

Try to **stick to one style**.

Comment

Leave meaningful comments. You might not remember your own code in a few months, imagine someone else trying to read it!

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.

```r
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
```

```r
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 ...
```

Functional Programming

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

```r
ncols <- ncol(mtcars)
means <- medians <- mins <- maxs <- rep(0, ncols)

for(i in 1:ncols){
  means[i] <- mean(mtcars[[i]])
  medians[i] <- median(mtcars[[i]])
  mins[i] <- min(mtcars[[i]])
  maxs[i] <- max(mtcars[[i]])
}

results <- data.frame(means, medians, mins, maxs)
results$col <- names(mtcars)

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
summ <- function(x){
  data.frame(means = mean(x),
             medians = median(x),
             mins = min(x),
             maxs = max(x))
}
ncols <- ncol(mtcars)
dfs <- vector(mode = "list", length = ncols)

for(i in 1:ncols){
  dfs[[i]] <- summ(mtcars[[i]])
}
```

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

The actual real functional way require using the built-in iteration tools `*apply`. In this way you avoid writing the verbose `for` loop.

```
results <- lapply(mtcars, summ)
results <- do.call(rbind, results)
head(results, n = 6)
```

```
          means medians   mins    maxs
mpg   20.090625  19.200 10.400  33.900
cyl    6.187500   6.000  4.000   8.000
disp 230.721875 196.300 71.100 472.000
hp   146.687500 123.000 52.000 335.000
drat   3.596563   3.695  2.760   4.930
wt     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
means <- rep(0, ncol(mtcars))
for(i in 1:length(means)){
  means[i] <- mean(mtcars[[i]])
}

# the same with sapply
means <- sapply(mtcars, mean)
```

`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:

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

Very fast:

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

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 1133.568 1232.993 1382.2059 1284.5915 1352.364 5106.099   100  a 
 preallocated  645.627  669.899  747.9499  691.1165  737.098 1899.243   100   b
```

With `*apply` you can do crazy stuff!

```r
funs <- list(mean = mean, sd = sd, min = min, max = max, median = median)
sapply(funs, function(f) lapply(mtcars, function(x) f(x)))
```

```
     mean     sd        min   max   median
mpg  20.09062 6.026948  10.4  33.9  19.2
cyl  6.1875   1.785922  4     8     6
disp 230.7219 123.9387  71.1  472   196.3
hp   146.6875 68.56287  52    335   123
drat 3.596563 0.5346787 2.76  4.93  3.695
wt   3.21725  0.9784574 1.513 5.424 3.325
qsec 17.84875 1.786943  14.5  22.9  17.71
vs   0.4375   0.5040161 0     1     0
am   0.40625  0.4989909 0     1     0
gear 3.6875   0.7378041 3     5     4
carb 2.8125   1.6152    1     8     2
```

Pure functions

Pure functions have no side effects and always return the same output for a given input.

Pure function

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

```
[1] 3
```

```r
print(x)
```

```
[1] 4
```

Impure function

```r
add_impure <- function(x) {
  x <<- x + 1
}
add_impure(x)
print(x)
```

```
[1] 5
```

Test your functions - `fuzzr`

Define your function…

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

Define properties that should always hold true…

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

Test the function across many randomly generated inputs…

```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.

If your functions are project-specific you can define them into your scripts or write some R scripts only with functions and `source()` them into the global environment.

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

And inside `utils.R` you have some functions:

```r
myfun <- function(x) {
  # something
}
```

Then you can load the function using `source("R/utils.R)` at the beginning of `analysis.R`:

```r
source("R/utils.R")
```

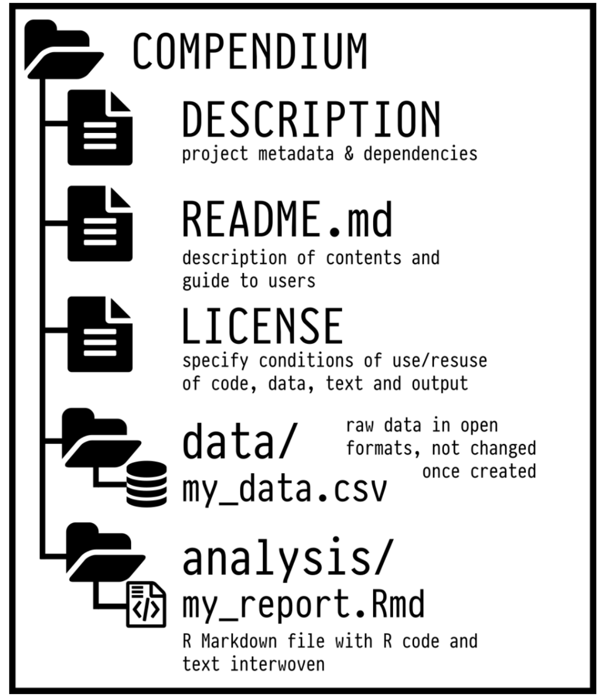
More about functional programming in R

- 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)

Organize your work - R projects

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, method, and output, while unambiguously expressing the relationship between those three (original data is untouched!).
- Specify the computational environment that was used for the original analysis

Tutorial

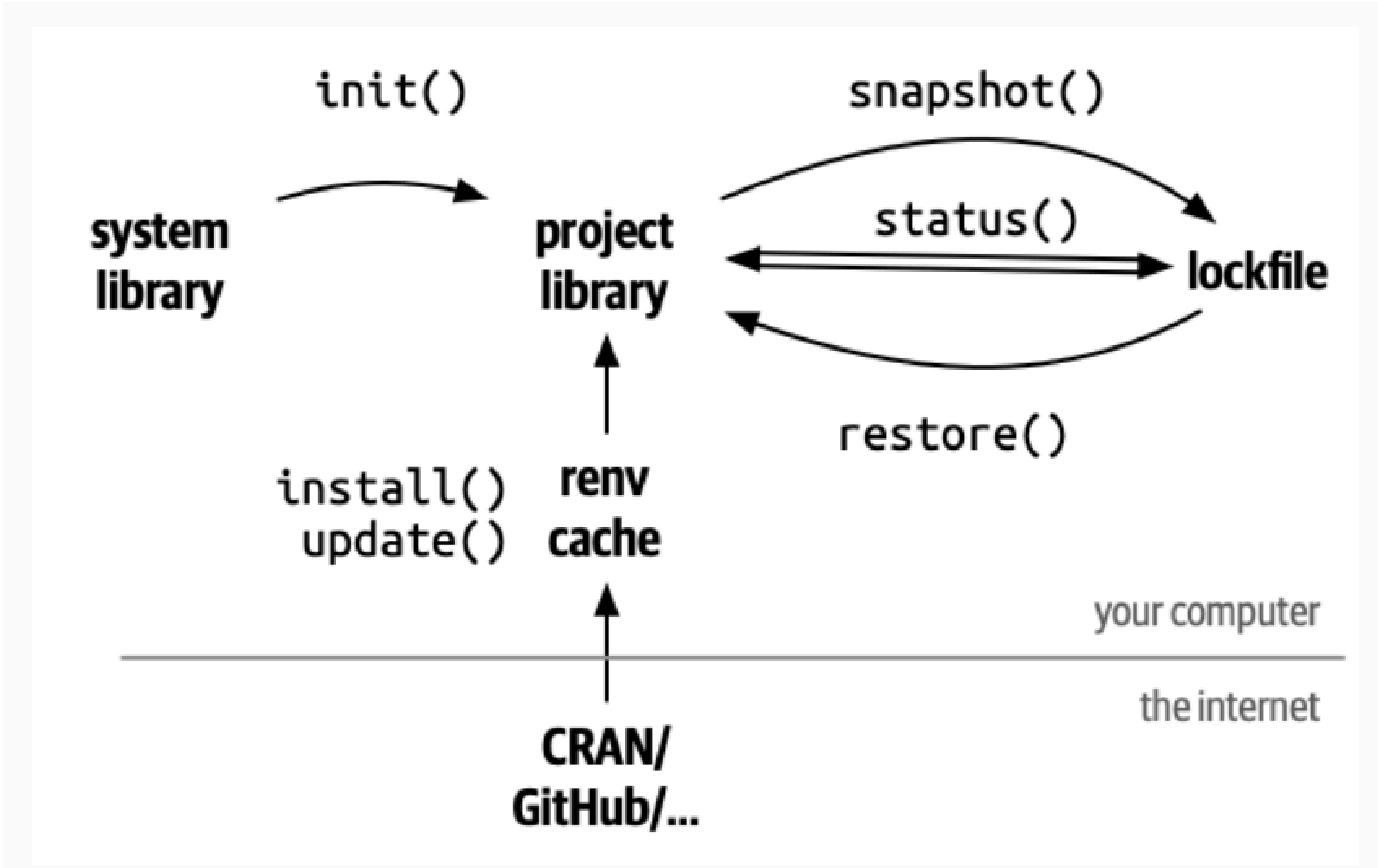
`rrtools::create_compendium()` builds the basic structure for a research compendium.

- Storage for general metadata (e.g., citation details)
- Dependency management via **DESCRIPTION** file
- Function storage and documentation in R/ folder

These features enable managing, installing, and sharing project-related functionality.

Research `renv` 📦

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



renv workflow

Project specific library

```
install.packages('microbenchmark')
```

The following package(s) will be installed:

- microbenchmark [1.5.0]

These packages will be installed into

“~/repro-pre-school/example-renv/renv/library/macos/R-4.4/aarch64-apple-darwin20”.

Research `rrtools` + `renv` 📦

Quarto

Apa quarto https://wjschne.github.io/apaquarto/

PDF output

docx output

Better code

Use consistent naming and comments ✅

Break long scripts into functions! ✅

Use clear structure

Use version control (e.g., Git)

Data sharing

Reproducible Documents

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