Kurt Lewin Instituut
March 2025

Sarah Ashcroft-Jones
Columbia University
sa4422@cumc.columbia.edu

Sofia Pelica
Vrije Universiteit Amsterdam
s.i.gomes.pelica@vu.nl

Best Practices for Reproducible Data Processing and Analysis using R and Quarto

Housekeeping

- ➔ **R and RStudio should be downloaded already**
- ➔ We have uploaded some example scripts to GitHub to help guide you - here. We have also uploaded **solution scripts - do not rely on those yet - make mistakes to learn!**
- ➔ We will respect breaks and will shift content around to do so
- ➔ We will provide the slides after the workshop is complete

Overview - Day 1 Schedule

- ➔ 14.00-14.30: Introduction
- ➔ ***Break***
- ➔ 14.40-15.30: Getting Started with R and Quarto
- ➔ ***Break***
- ➔ 16.00-17.00: How to Code Understandably

Overview - Day 2 Schedule

- ➔ 14.00-14.50: How to Code Defensively
- ➔ ***Break***
- ➔ 15.00-15.30: Publishing Reproducible Scripts
- ➔ ***Break***
- ➔ 15.30-16.40: Hands-on Practice (***Break*** included)
- ➔ 16:40-17.00: Closing Remarks and Resources

Why are we here?

Because maybe …

- ➔ You studied the language
- ➔ Learnt about all the libraries
- ➔ Practiced (struggled with) coding for weeks
- ➔ Value open science principles and practices
- ➔ But your code still feels like a 7 headed apocalyptic monster?

No worries - this course is designed to help with exactly that!

Introduction and Aims

> “Gaining confidence and clarity on how and why to process data reproducibly in R and Quarto for easier analysis, better collaborations, and better quality research!”

Workshop Objectives and Learning Outcomes

Day 1: The “why” of reproducible data processing/analysis

Understanding the principles of computational reproducibility

Using free/open-source software to enhance reproducibility/accessibility.

Learning how to improve coding readability.

Day 2: the “how” of reproducible data processing

Practicing defensive coding techniques.

Understanding portability in coding practices

Reproducibility as the bedrock of Open Science

Open data is not enough.

If you want to be able to reproduce your data (for yourself or others), or to be transparent about your data - you need to go further.

What did you do to get to the point that your data looks so perfect and tidy in that key plot? How much polishing was done? What did that process look like?

To commit to openness and transparency in science is to commit to reproducible research practices including clear documentation of data processing and analysis.

Reproducibility and open data processing and analysis

Reproducible analytics:

When someone else can take your data and create a reproduction of your output values and plots from start to finish

How does someone else (or you in 5 years/months time) know what you did at each step unless it is fully recorded somewhere in detail?

Reproducibility and open data processing and analysis

In the past - point and click software:

SPSS or Excel - the “hidden” changes cannot be tracked, no idea of how did they get from the raw data to the output values?

Now open and reproducible approaches:

R as a free open-source and fully available programming language

The importance of minimising manual analysis steps

“ Manual procedures are not only inefficient and error-prone, they are also difficult to reproduce”

Sandve, Nekrutenko, Taylor & Hodvig (2013)

What could go wrong and how would we know?

Copy and paste - e.g.“sticking” together two datafiles

Changing single values - e.g. adjusting “NA” markers manually

Transforming a variable - e.g. log transformation of a column

Removing data - e.g. “weird” values, incomplete data, outliers

Relabelling data - e.g. changing scale values from 0-4 to 1-5

The solution to manual analysis and data processing?

Just don’t do it - stop, please, now!

Instead use reproducible and traceable approaches that document your work - like R and/or Quarto!

If manual operations cannot be avoided, you should as a minimum note down which data files were modified or moved, and for what purpose.

(Shameless Plug: lab e-notebooks are great!)

Our main argument and takeaway

<u>R and Quarto are excellent tools to increase your analysis reproducibility.</u>

However, when used correctly:

- ➔ They will also make your life easier (less headaches!)
- ➔ They will also increase your ability to collaborate effectively (teamwork!)
- ➔ They can also increase the quality of your work (low chance of error!)

BREAK

Getting started (again) in R and Quarto

AKA if you aren’t ashamed of your old code you aren’t growing

What is R? *and* What is Quarto?

Open-source software for analysis and data processing - a toolkit!

R - a programming language, usually interacted with via RStudio

Quarto - RMarkdown but with multi-language options and “easier” text formatting (less need for LaTeX knowledge).

What is R? *and* What is Quarto?

We want our work to be:

Reliable – error free and accurate, i.e. you can rely on it

Replicable – you can recreate the “clean” data again given the raw data

Ideally we want both: R and/or Quarto can help us achieve those goals!

Side bar - the importance of version control

Combining use of R and Quarto with version control like GitHub can be invaluable.

Beyond the scope of today’s session but consider the value of being bale to return to old (working) versions of your code if you made a lot of changes and now you decide you want to return to the original approach

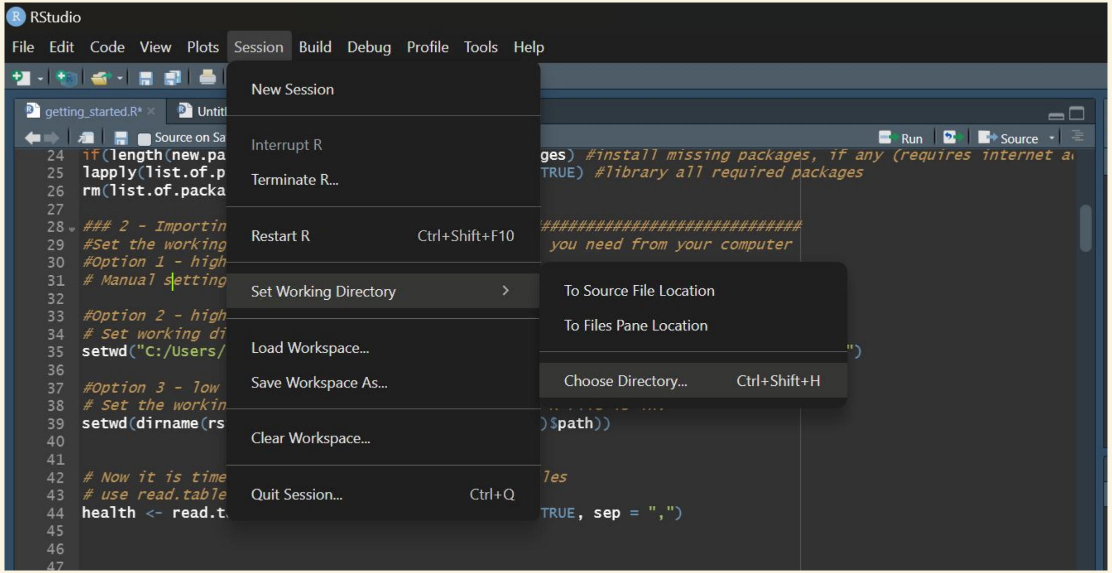
GitHub docs “Hello World” tutorial a good place to start

Hands on exploration!

Getting started with data processing in R: setwd()

Option 1 - high friction and low reproducibility

Manual setting of the working directory using the menu settings



```
if(length(new.pa                                   ges) #install missing packages, if any (requires internet a
lapply(list.of.p                                   TRUE) #library all required packages
rm(list.of.packa

### 2 - Importin                                   ##############################
#Set the working                                    you need from your computer
#Option 1 - high
# Manual setting

#Option 2 - high
# Set working di
setwd("C:/Users/                                                                 ")

#Option 3 - low
# Set the workin
setwd(dirname(rs                                   )$path))


# Now it is time                                   les
# use read.table
health <- read.t                                   TRUE, sep = ",")
```

Getting started with data processing in R: setwd()

Option 2 - high friction and risk of error in collaboration

Set working directory with a manual file path to a specific folder in your computer, e.g.:

```
setwd("C:/Users/sa4422/OneDrive - Columbia University
Irving Medical Center/Documents")
```

Must be adjusted on each version of the script and for every file

Getting started with data processing in R: setwd()

Option 3 - low friction, high portability easy collaboration

Set the working directory to the folder that this R file is in:

```
setwd(dirname(rstudioapi::getActiveDocumentContext()$path))
```

Getting started with data processing in R: packages

Option 1 - high friction, low editability, and little collaborative ease

Install and library each package across the script one by one, e.g.:

```
install.packages(“tidyverse”)

library(tidyverse)
```

Bonus points for no comments on what each package is used for in the script and if the are distributed across the whole script.

Getting started with data processing in R: packages

Option 2 - highly portable, flexible, and collaborator friendly

```r
list.of.packages <- c(
    "tidyverse", # handles data wrangling
    "lavaan" # running the models )

new.packages <- list.of.packages[!(list.of.packages %in%
installed.packages()[,"Package"])]

if(length(new.packages)) install.packages(new.packages)
lapply(list.of.packages, require, character.only = TRUE)
```

Getting started with data processing in R: loading data

Option 1 - standard, but not immune to error!

```
data <- read.table(file = "data.csv", header = TRUE, sep = ",")
data <- read.csv("data.csv")
```

If your data is housed in the same folder as your script this should work smoothly.

read.csv() works slightly more economically so more time efficient for large datasets

However - does your collaborator have that file in the right version?

Getting started with data processing in R: loading data

Option 2 - download data directly, better?

```
download.file(url='http://www.website.com/data.csv',
              destfile='data.csv', method='curl')
data <- read.csv("data.csv")
```

Will need to have installed RCurl to do so - but consider this if your datafiles are updating regularly. Easier for collaboration and ensuring you are working with the right file each time.

Getting started with data processing in R: loading data

Option 3 - save and use Rdata files, even better?

```
save(clean_dataframe, plot_data, file="data.RData")
load(file="data.RData")
```

OR - for single objects

```
write_rds(clean_dataframe,"data.rds")
data <- read_rds("data.rds")
```

Getting started with data processing in R: loading data

Option 3 - save and use Rdata files, even better?

➔ Preserves the data types (e.g. lose factor data types in readr)

➔ Preserves metadata (e.g. factor levels and labels)

➔ But - not transferable to Python for example.

The 80:20 Rule

“ 80% of a data scientists valuable time is spent simply finding, cleansing, and organizing data, leaving only 20% to actually perform the analysis”

IBM Data Analytics

Data processing in R

Code allows you to automate repetitive tasks and while reducing errors that emerge during manual edits.

GUI-based alternatives (like Excel) don’t offer the flexibility, transparency, reliability, and replicability that R does

Code tells you exactly how a piece of data was transformed and executing it will give you the same results every time.

“ Tidy dataset are all alike but every messy dataset is messy in its own way”

Hadley Wickham

Data processing in R

Even when your data is “structured,” this doesn’t mean it’s “clean”

“clean”: when data is structured, edited, and/or formatted such that the desired use/analysis can be performed on it without further preparation

We need to have this automated and clearly documented as this “clean” data is the foundation of our analysis and therefore our conclusions.

Proof of concept - outlier issues

You notice that 7 participants out of your 120 have a very long completion time. What do you do?

Remove their data?

Overwrite their completion times with the average?

Winsorise their times?

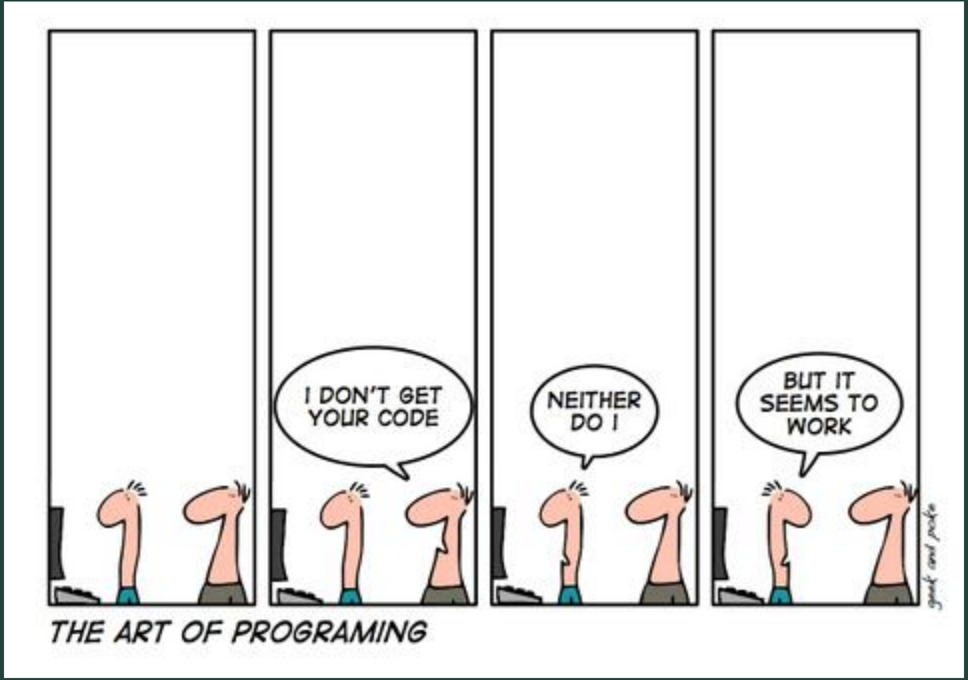
Flag their rows and perform the analysis wit and without inclusion to understand analysis sensitivity?

If you don’t automate this and record it nobody knows.

Hands on demonstration!

BREAK

Writing Clear and Comprehensible Code



(Geek & Poked, 2008)

- ➔ **Write clean and well-structured code** .
- ➔ Add **clear comments** that explain what each section of the code does.
- ➔ Ensure that **dependencies and the environment are well documented** .

Writing clean and well-structured code

Use Meaningful Variable and Function Names.

Names should be long and descriptive but not too long.

Bad Examples:

```
x <- 100

df <- read.csv("data.csv")

data <- read.csv("data_april.csv")

temp <- filter(data, var1 > 10)

regular_sessions_in_red_room <- ggplot(p)
```

Good Examples:

```
sample_size <- 100

raw_data <- read.csv("data.csv")

filtered_data <- filter(data, var1 > 10)

regular_sessions_red <- ggplot(var2)
```

The goal is "Don't make me think"

Writing clean and well-structured code

Naming Conventions for Variables

Use snake_case (recommended in R!) instead of spaces or special characters

Yes: `mean_age` No`: `mean age`` No: `mean-age` Use `janitor::clean_names(raw_data)`

Be consistent. If you prefer camelCase, then use it throughout (e.g., responseTime).

Avoid using "." for variable names because R uses it in the background.

Avoid using certain R reserved words for built-in functions for variable names (e.g., mean, sum, true, false, filter).

Writing clean and well-structured code

Each function and variable should have one clear purpose.

Avoid reusing variable names for different purposes, even if the original variable is deleted earlier in the script.

Bad example:

```
temp <- filter(data, age > 18)
rm(temp)

[...]

temp <- read.csv("identification.csv")
```

Writing clean and well-structured code

Follow a Consistent Coding Style

Use consistent indentation (two or four spaces per level). Never mix tabs and spaces.

Messy code:

```
if(mean(x)>10){print("High value")
} else{print("Low value")}
```

Clean code:

```
if (mean(x) > 10) {
   print("High value")
} else {
   print("Low value")
}
```

Use RStudio's Reindent Lines tool (Ctrl/Cmd + I) to auto-format code!

Writing clean and well-structured code

Follow a Consistent Coding Style

Keep lines short (below 80 characters). If a line gets too long, break it.

Hard to read:

```
summary <- filter(data, age > 18 & income > 50000 & gender == "male")
```

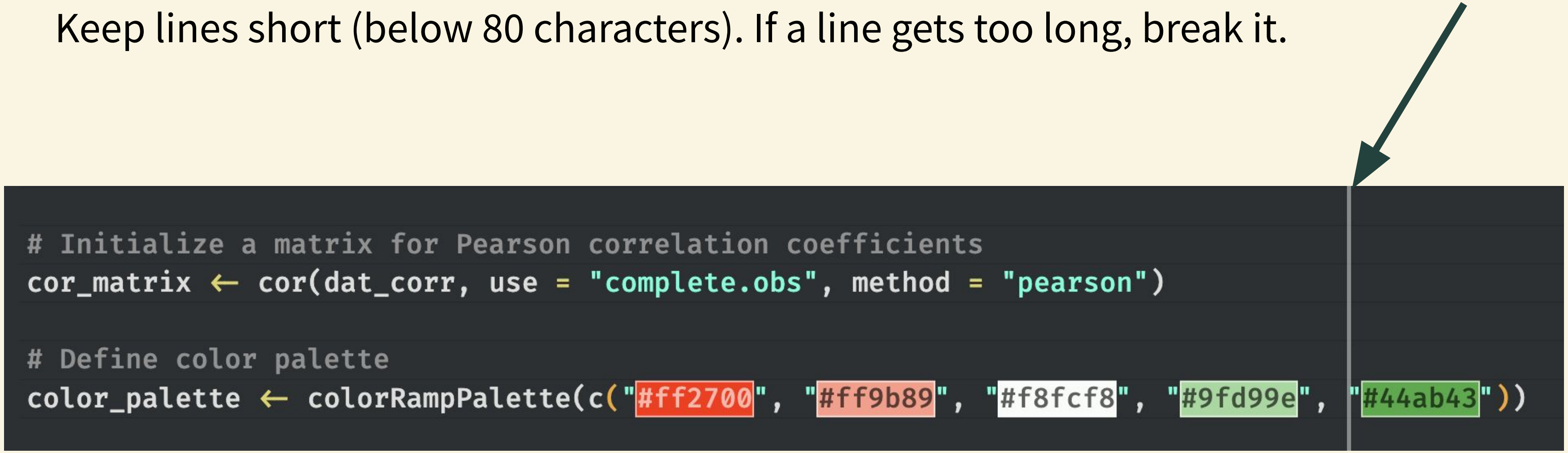
More readable:

```
summary <- filter(
  data,
  age > 18 &
  income > 50000 &
  gender == "male"
)
```

Writing clean and well-structured code

Follow a Consistent Coding Style

Keep lines short (below 80 characters). If a line gets too long, break it.



```r
# Initialize a matrix for Pearson correlation coefficients
cor_matrix ← cor(dat_corr, use = "complete.obs", method = "pearson")

# Define color palette
color_palette ← colorRampPalette(c("#ff2700", "#ff9b89", "#f8fcf8", "#9fd99e", "#44ab43"))
```

Writing clean and well-structured code

Follow a Consistent Coding Style

Place spaces around operators (=, +, -, ==, <-, etc.).

Hard to read:

```
mean(x,na.rm=TRUE)
df<-data.frame(x=1:10,y=runif(10))
```

More readable:

```
mean(x, na.rm = TRUE)
df <- data.frame(x = 1:10, y = runif(10))
```

Writing clean and well-structured code

Follow a Consistent Coding Style

Avoid long scripts with repetitive code. Instead, define functions or use piping.

Bad example:

```r
data_cleaned <- filter(data, age > 18)
data_cleaned <- select(data_cleaned, age, income, gender, country)
data_cleaned <- arrange(data_cleaned, desc(income))
```

Good example:

```r
data_cleaned <- data %>%
  filter(age > 18) %>%
  select(age, income, gender, country) %>%
  arrange(desc(income))
```

Writing clean and well-structured code

Cleaning the Enviroment

If you know which variables you no longer need, you can remove them using `rm()`

Bad example:

```
# Remove all variables except raw_data and filtered_data
rm(list = setdiff(ls(), c("raw_data", "filtered_data")))
```

Good example:

```
# Remove var1 and var2
rm(var1, var2)

# Remove all starting with "test_"
rm(list = ls(pattern = "^test_"))
```

Writing clean and well-structured code

Don’t leave commented code.

They are dead code since they don’t compile or run.

Bad example:

```
# ggplot(data, aes(x = x, y = y)) + geom_point() +
#     geom_smooth(method = "lm", col = "blue") +
#     labs(title = "Scatter Plot with Regression Line",
#          x = "Independent Variable (x)",
#          y = "Dependent Variable (y)")
```

Writing clean and well-structured code

Disable scientific notation for easier readability of large and small numbers.

Bad example:

```
[1] 1.234568e+08
[1] 1.23e-05
```

Good example:

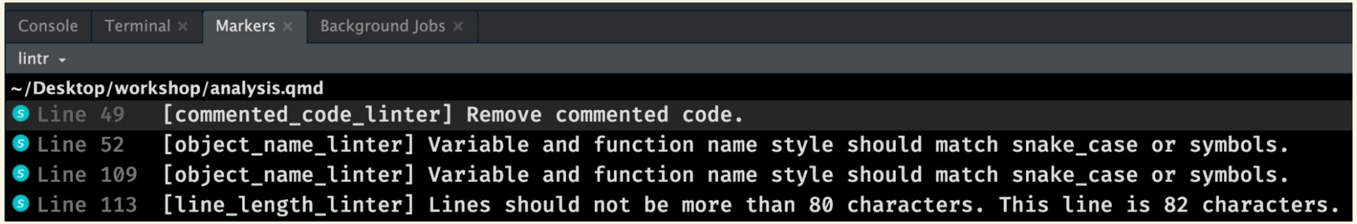
```
options(scipen = 999)
[1] 123456789
[1] 0.0000123
```

Writing clean and well-structured code

Tip:

Use **Lintr** package to check if your script follows these guidelines (lintr.r-lib.org).

```
install.packages("lintr")
library("lintr")
lint("location/my_script.qmd")
```



```
~/Desktop/workshop/analysis.qmd
Line 49   [commented_code_linter] Remove commented code.
Line 52   [object_name_linter] Variable and function name style should match snake_case or symbols.
Line 109  [object_name_linter] Variable and function name style should match snake_case or symbols.
Line 113  [line_length_linter] Lines should not be more than 80 characters. This line is 82 characters.
```

Adding clear comments and documentation

Write comments to explain **why** something is done, not just **what** is happening.
Comments should not duplicate the code (watch out for GenAI-generated comments).

Bad comments:

```r
# Compute the mean of x
mean(x)
```

Explain the why!

Good comments:

```r
# Removing participants who did not complete at least 80% of questions
cleaned_data <- filter(survey_data, completion_rate >= 0.8)
```

Adding clear comments and documentation

Avoid writing unnecessary or overly detailed explanations. Keep comments brief and meaningful.

Bad comments:

```r
# The following line of code will take the variable "temperature"
# and apply a function called "mean()" to it, which will compute
# the arithmetic average of all the numbers contained in this variable.
mean_temperature <- mean(temperature)
```

Good comments:

```r
# We need to know the mean temperature for the next step
mean_temperature <- mean(temperature)
```

Adding clear comments and documentation

Use comments to break down complex logic.

Write comments at the same time you’re writing the code.

```
cleaned_data <- data %>%

  # Step 1: Remove missing values from key variables
  filter(!is.na(age) & !is.na(income) & !is.na(gender)) %>%

  # Step 2: Exclude participants under 18
  filter(age >= 18) %>%

  # Step 3: Create an income category variable
  mutate(
    income_bracket = case_when(
      income < 30000 ~ "Low",
      income >= 30000 & income <= 70000 ~ "Middle",
      income > 70000 ~ "High"
    )
  )
```

Adding clear comments and documentation

Place the comment **above** the code block. Be consistent.

Bad comments:

```r
filtered_data <- filter(data, age > 18)
# Keep only adults


filtered_data <- filter(data, age > 18)  # Keep only adults
```

Good comments:

```r
# Keep only adults
filtered_data <- filter(data, age > 18)
```

Adding clear comments and documentation

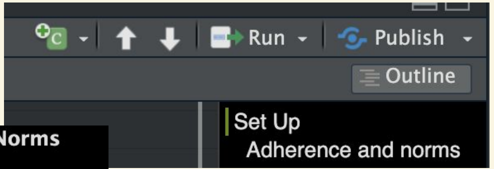
When working with large scripts, use block comments to **separate sections**.

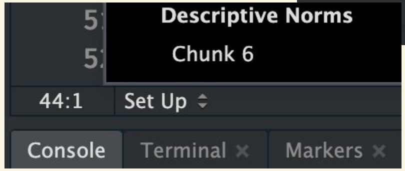
R Scripts:

```r
##### Load Libraries #####
library(dplyr)

##### Load Data #####
data <- read.csv("survey_results.csv")

##### Data Cleaning #####
data <- data %>%
  # Remove rows with missing age
  filter(!is.na(age)) %>%
  # Categorize income
  mutate(income_bracket = ifelse(income > 50000, "High", "Low"))

##### Data Visualization #####
ggplot(data, aes(x = income_bracket, fill = gender)) +
  geom_bar() +
  labs(title = "Income Distribution by Gender")
```

Quarto Scripts:

~~~
# Set Up
```{r}
#| output: false
library(dplyr)
```


## Adherence and norms

```{r}
#| output: false
data <- read.csv("survey_results.csv")
```
~~~





Adding clear comments and documentation

When working with large scripts, use block comments to **separate sections**.

You can also use the **ARTofR** package for R Scripts:

```
##~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
##                                                                            --
##----------------------------------- TITLE-------------------------------------
##                                                                            --
##~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


##~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
##                                    Title                                 ----
##~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~


##~~~~~~~~~~~~~~~
##  ~ Title  ----
##~~~~~~~~~~~~~~~

#...............................................................................
#                                                                              .
#  I used ARTofR everyday and it makes my R script so neat. I used ARTofR      .
#  everyday and it makes my R script so neat. I used ARTofR everyday and it    .
#  makes my R script so neat.                                                  .
#                                                                              .
#...............................................................................
```

Documenting dependencies and environment settings

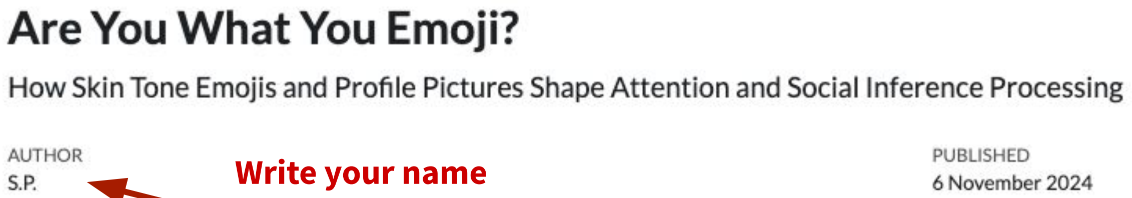
It’s crucial to document your environment because **in about 10 years**

1) **some of the packages you used may be deprecated,**
2) **the software as we know it may go through drastic changes.**

Example:

```
Warning message:
funs() is soft deprecated as of dplyr 0.8.0
please use list() instead
```

At the start of the script…



Are You What You Emoji?

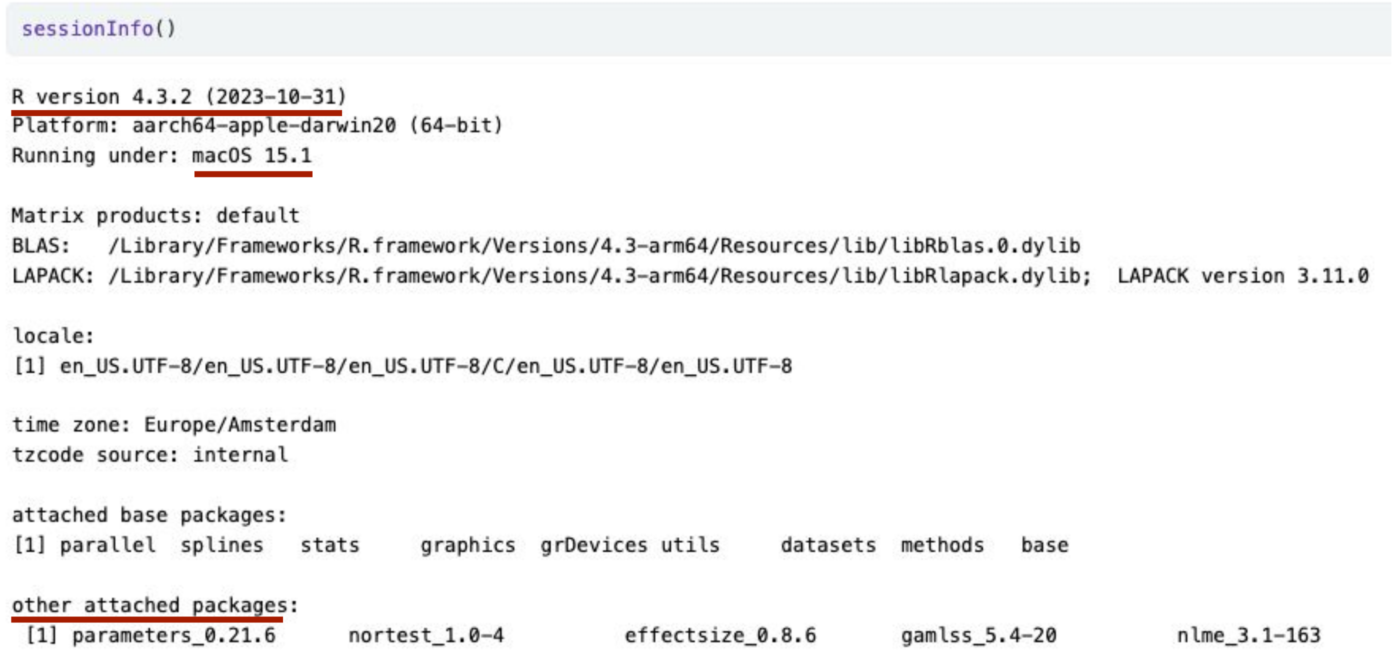
How Skin Tone Emojis and Profile Pictures Shape Attention and Social Inference Processing

AUTHOR
S.P.

PUBLISHED
6 November 2024

Write your name using only initials for peer-review!

At the end of the script…



```r
sessionInfo()
```

```
R version 4.3.2 (2023-10-31)
Platform: aarch64-apple-darwin20 (64-bit)
Running under: macOS 15.1

Matrix products: default
BLAS:   /Library/Frameworks/R.framework/Versions/4.3-arm64/Resources/lib/libRblas.0.dylib
LAPACK: /Library/Frameworks/R.framework/Versions/4.3-arm64/Resources/lib/libRlapack.dylib;  LAPACK version 3.11.0

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/Amsterdam
tzcode source: internal

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

other attached packages:
 [1] parameters_0.21.6     nortest_1.0-4         effectsize_0.8.6      gamlss_5.4-20         nlme_3.1-163
```

Leverage Quarto capabilities to increase readability

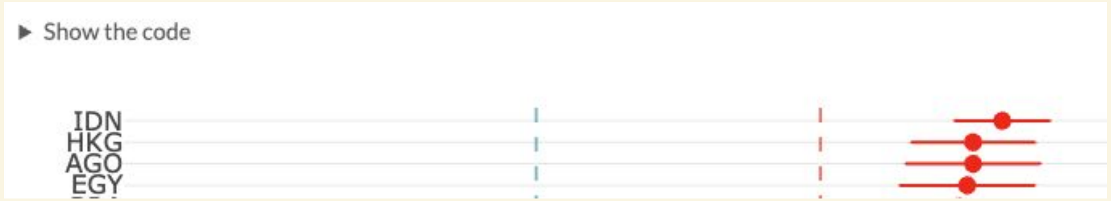
Avoid printing unnecessary content when using Quarto.

Output: False runs the code but hides its results.

Echo: False runs the code but hides the code while displaying the results.

Code-fold: Hides the code but allows it to be accessed when unfolded.

~~~
```{r}

#| code-fold: true

library(dplyr)
```
~~~

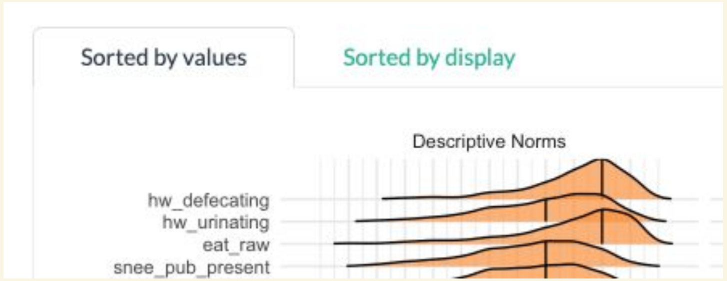


Avoid using **warning: false**. Instead, explain why the warning message is not relevant.

Leverage Quarto capabilities to increase readability

Use **tabset panels** to allow users to switch between different visualizations without cluttering the report.

~~~
::: {.panel-tabset}

## Sorted by values

```{r}
plot_values
```


## Sorted by display

```{r}
plot_display
```


:::
~~~



“We consider a paper to be an advertisement, and for the associated code, data, and environment to be the actual work.”

Buckheit and Donoho (1995)

Approaches to Error-Resistant Coding

Defensive Coding is very similar to Defensive Driving:

- An approach that emphasizes **safety to reduce the risk** of errors.
- It **goes beyond basic guidelines** and focuses on unforeseen code behaviors.
- Writing code with the **awareness that unexpected errors may occur unnoticed**.

Don’t assume your code is correct just because it didn’t give a warning message and looks right.

Expect the Unexpected and Be Prepared for the Illogical

More than 70% of researchers

have failed to reproduce

another scientist's experiments

(in a total of 1,576 researchers)

(Baker, 2016)

More than **50%** of researchers

have failed to reproduce

their **own** experiments

(in a total of 1,576 researchers)

(Baker, 2016)

More than 40% of the retracted papers

were due to honest errors and problems with reproducibility

(in a total of 10,500 retracted papers)

(Brainard & You, 2018)

Approaches to Error-Resistant Coding

A **sanity check** is a quick, basic test to ensure that a piece of code or a characteristic of the data makes sense and meets with expectations.

It helps catch errors or inconsistencies early in the development process.

The goal is to do perform it continuously as you work,
especially during preprocessing.

Approaches to Error-Resistant Coding

Sanity check: ALWAYS check the unique values of the variables of interest.

Manually reviewing the full data is not viable in large datasets. However, this small test able us to have complete overview of the variables you are interested in.

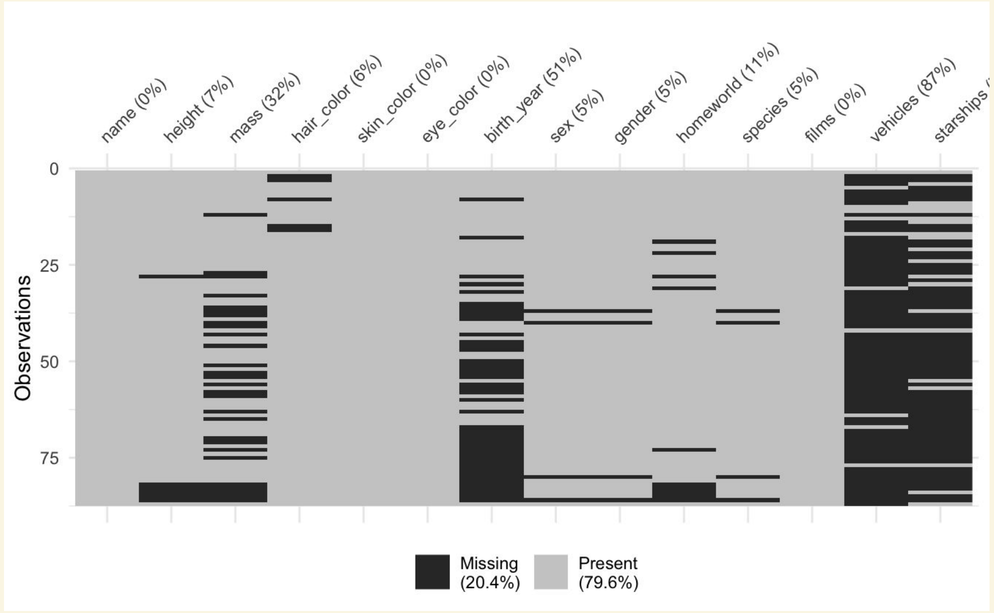
```
table(raw_data$fses1)
  1   2   3   4   5   6   7 
130 109 105 108 110 117 112 
```

Approaches to Error-Resistant Coding

Sanity check: Missing values.

Even if you believe you configured all items as forced responses in Qualtrics, errors can still occur.

```
vis_miss(raw_data)
```



Approaches to Error-Resistant Coding

Sanity check: Reversing variables.
Even if you believe the code executed as expected, always verify if the changes were applied correctly.

```
table(raw_data$freq_emoji_usage)
  1   2   3   4   5   6   7   8
108  92  92  90  89 104 105 111

raw_data$freq_emoji_usage <- 9 - raw_data$freq_emoji_usage

table(raw_data$freq_emoji_usage)
  1   2   3   4   5   6   7   8
111 105 104  89  90  92  92 108
```

Approaches to Error-Resistant Coding

Sanity check: Computing indices.
When calculating indices across multiple columns, **ALWAYS use the column names** and never their index positions.

Bad example:

```
data$fses_avg <- rowMeans(data[, 24:29])
```

To verify the code, manually compute the index for one participant and compare it.

Good example:

```
raw_data$ires_avg <- rowMeans(raw_data[, c("ires1", "ires2", "ires3")], na.rm = TRUE)

(raw_data$ires1[1] + raw_data$ires2[1] + raw_data$ires3[1])/3

[1] 4.666667

raw_data$ires_avg[1]

[1] 4.666667
```

Approaches to Error-Resistant Coding

Sanity check: Merge dataframes

```
data_merged <- raw_data_a %>%
  full_join(raw_data_b) %>%
  full_join(raw_data_c)

# Calculate the row sum of the individual dataframes.
# If the values are the same, the sanity check passed.
nrow(data_merged)
nrow(raw_data_a) + nrow(raw_data_b) + nrow(raw_data_c)
```

Approaches to Error-Resistant Coding

Sanity check: Merge dataframes

```
data_merged <- raw_data_a %>%
  full_join(raw_data_b) %>%
  full_join(raw_data_c)

# Compare columns' names from the individual dataframes with the columns' names from data_merged
data_merged_columns <- names(data_merged)
all_columns <- unique(c(names(raw_data_uni_a),
                        names(raw_data_uni_b),
                        names(raw_data_uni_c))

# Columns missing in data_merged_columns. If empty, the sanity check passed.
setdiff(all_columns, data_merged_columns)

# Columns in excess in data_merged_columns. If empty, the sanity check passed.
setdiff(data_merged_columns, all_columns)
```

Publishing Reproducible and Portable Scripts

Save and publish your data in both CSV and RDA formats. CSV is a universal format, but when imported into RStudio it often requires transformations, which can introduce errors. RDA files preserve data properties without requiring transformations when reloading the data.

For example, a CSV file might use a comma as a decimal separator, while RStudio expects a period. As a result, RStudio may misinterpret the comma as a thousands separator, leading to an error that you might not notice.

CSV file:	RStudio:
`1,095`	`I am sure that is 1095.00` ❌

```
write.csv(data_cleaned, "data_cleaned.csv")

save(data_cleaned, file = "data_cleaned.rda")
```

Publishing Reproducible and Portable Scripts

ALWAYS keep the original ResponseID in all version of your data to avoid attribution errors and ensure you can trace the responses back to the survey platform.

Bad example:

```
StartDate            ID          consent  intro  happiness1  age
2025-02-23 12:21:09   1                1      1           6   34
2025-03-01 08:14:39   2                1      1           3   21
```

Good example:

```
StartDate            ResponseId          consent  intro  happiness1  age
2025-02-23 12:21:09  R_8xJmtpXpLsqMyfB         1      1           6   34
2025-03-01 08:14:39  R_3rBKeC9Sq8djSYf         1      1           3   21
```

Publishing Reproducible and Portable Scripts

Consider publishing both a **cleaned data version** and a **public data version**.

The public version should remain **as close to the raw data as possible** but is in compliance with ethical codes of conduct. For example, in an adults-only study, rows for underage participants are retained but anonymized, with all data except ResponseId and an "underage" identifier removed.

Public data:

```
StartDate            ResponseId        consent  intro  happiness1 age  identifier
2025-02-23 12:21:09  R_8xJmtpXpLsqMyfB       1      1           6  34
                     R_5oNJhFVJFoSdqKc                                  underage
2025-03-01 08:14:39  R_3rBKeC9Sq8djSYf       1      1           3  21
```

Publishing Reproducible and Portable Scripts

ALWAYS publish a codebook along your data. It should be regarding the data version you publish that is more close to the raw state. It acts as a user manual for your data, making it easier for both you and others to understand, use, and interpret the data correctly.

Codebook (usually presented in a table format):

Name	Variable	Description	Value	Type	Note
Response ID	ResponseId	A unique identifier assigned to each participant's response		categorical	
Political orientation	politic_left_right	In politics people sometimes talk of 'left' and 'right'. Where would you place yourself on this scale, in general?	1 (Left) to 10 (Right)	numerical	

Publishing Reproducible and Portable Scripts

Publish your files in an organized structure. Use numbers at the beginning of file names to define their order, rather than relying on alphabetical sorting. Name the files in a sequence that prioritizes the most important ones first.

Good Example:

```
├── 01_master_script.html
├── 01_master_script.qmd
├── 01_public_data.Rda
├── 01_public_data.csv
├── 02_cleaned_data.Rda
├── 02_cleaned_data.csv
├── 03_codebook.xlsx
├── 📂 materials/
```

Download the files you published as **if you were an outsider to the project**, and run the script to ensure you can execute it completely without needing to make any changes.

If you have **multiple scripts**, consider adding a **README.txt** file to specify which data files each script requires.

Hands-on Practice

1. Go to our repository https://github.com/sarahajones/KLIworkshop2025

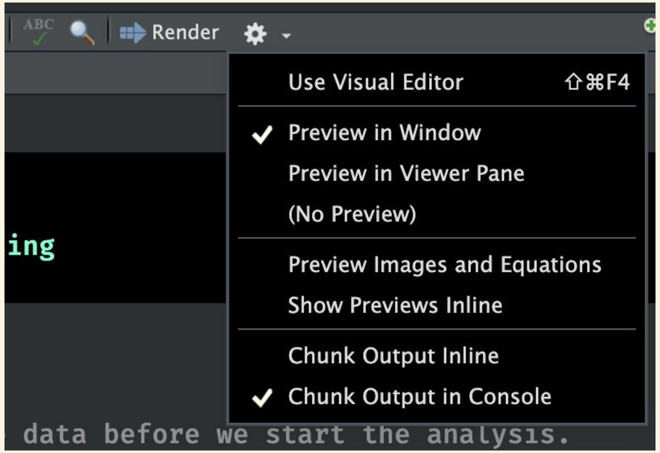
2. Place the files ***practice.qmd***, ***practice_data_us.csv*** and ***practice_data_uk.csv*** in the same folder.

3. Open the Quarto file .qmd

4. Restart the R session

5. Configure your settings

We suggest these settings:



Closing remarks

- Defensive coding isn't about perfection, it's about reducing errors and preventing them from going unnoticed.
- Have a **black-box script** to check the ethical compliance of the raw data. After removing underaged participants, hateful content, or identifiable information from text entries, save the data as the public data version. Then, **create a master script that starts with the public data** (that you will share) and proceed through preprocessing, analysis, etc.
- To make sure basic reproducible functions are in place, **ask a friend to run your script and data in their computer.**
- Consider **extracting data from Qualtrics directly into RStudio** and saving it as an .Rda file, rather than downloading it as a .csv file (see tutorial). You can find your Qualtrics API token in your Account Settings. If you don't have one, ask your lab manager.
- Consider using **Github Copilot** to save time (see tutorial).
- Consider using **Version Control with Git** from RStudio (see tutorial) to easily revert to a previous working state if needed.

Every year, Sarah seeks PhD students and Postdocs to develop their independent research projects and supervise small groups of BSc and MSc students in a cross-cultural setting.

If you are interested in gaining more experience as supervisor, follow us on LinkedIn, Instagram, or Bluesky.

Junior Researcher Programme

jrp.pscholars.org

Siena, Italy

University of Cambridge, UK