Write R code that lasts

Scientific workflows: Tools and Tips 🛠️

2025-05-15

What is this lecture series?

Scientific workflows: Tools and Tips 🛠️

🗓️ Every 3rd Thursday 🕓 4-5 p.m. 📍 Webex

- One topic from the world of scientific workflows
- Material provided online
- If you don't want to miss a lecture
  - Subscribe to the mailing list

The problem

Artwork by Allison Horst, CC BY 4.0

The problem with messy projects

🔧 **Maintainability** Can I understand, update and fix my own code?

🔄 **Reproducibility** Can someone else reproduce my results?

🏋️ **Reliability** Will my code work in the future?

⚙️ **Reusability** Can someone else actually use my code?

Today

Learn how to keep the kitchen clean.

- Basic tips for **reproducible** and **maintainable** data analysis projects
  - Project organization
  - Coding
  - Dependency management

The reproducibility mindset

- Imagine how someone else will see the project for the first time.

Will they be able to understand and use your project?

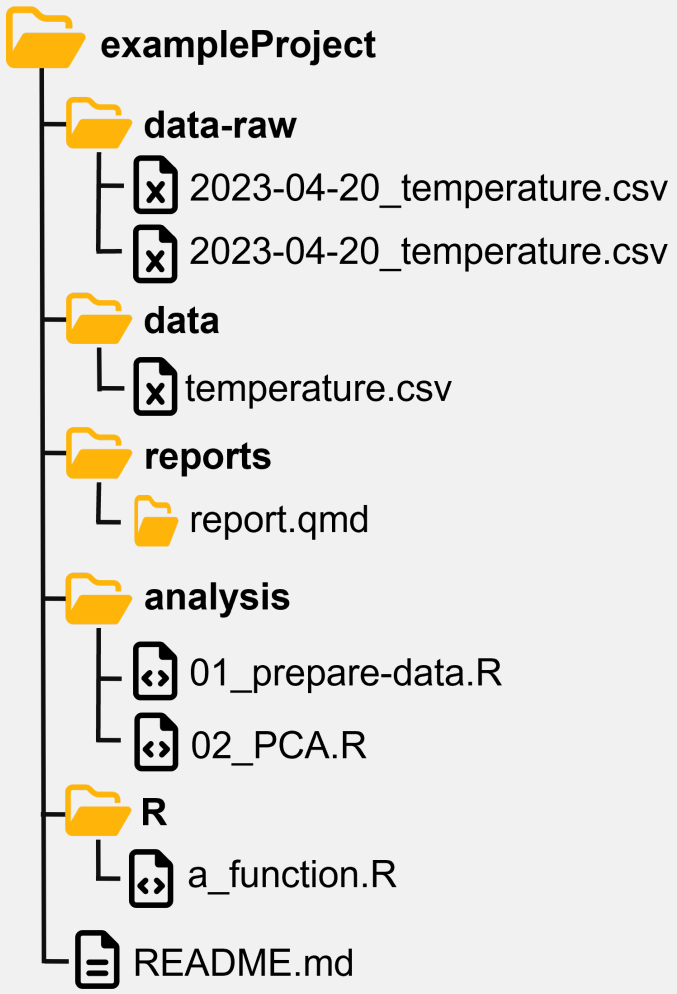
> **Practical reproducibility check**
>
> Send your project to a colleague and ask them **understand** and **run** the code your analysis.

First things first:

📁 Organizing projects

Have a solid project structure

- **Self-contained** projects: code, data, reports, etc. in one place
- Separate things into folders
- Always include a `README` file
- Use standard templates (see e.g. the `template` R package)



Name your files properly

Your collaborators and your future self will love you for this.

Principles [1]

File names should be

1. Machine readable
2. Human readable
3. Working with default file ordering

1. Machine readable file names

Names should allow for easy **searching**, **grouping** and **extracting information** from file names.

- No space & special characters
- Use consistent separators

Bad examples ❌

- `2023-04-20 temperature göttingen.csv`
- `2023-04-20 rainfall göttingen.csv`

Good examples ✔️

- `2023-04-20_temperature-goettingen.csv`
- `2023-04-20_rainfall-goettingen.csv`

2. Human readable file names

Names shoud be **informative** and reveal the **file content**.

- Use separators to make it readable

Bad examples ❌

- `01preparedataforanalysis.R`
- `01firstscript.R`

Good examples ✔️

- `01_prepare-data-for-analysis.R`
- `01_lm-temperature-trend.R`

3. Default ordering

If you order your files by name, the ordering should make sense:

- (Almost) always put something numeric first
  - Left-padded numbers (`01`, `02`, ...)
  - Dates in `YYYY-MM-DD` format

Chronological order


`2023-04-20_temperature-goettingen.csv`


`2023-04-21_temperature-goettingen.csv`

Logical order


`01_prepare-data.R`


`02_lm-temperature-trend.R`

Let’s start coding

Use save paths

To read and write files, you need to tell R where to find them.

Common workflow:

Set **working directory** with `setwd()`, then read files from there:

```
# set the project root as working directory
setwd("C:/Users/Selina/path/that/only/I/have")

# read my data using a path relative to the working directory
temperature <- read_csv("data/temperature.csv")
```

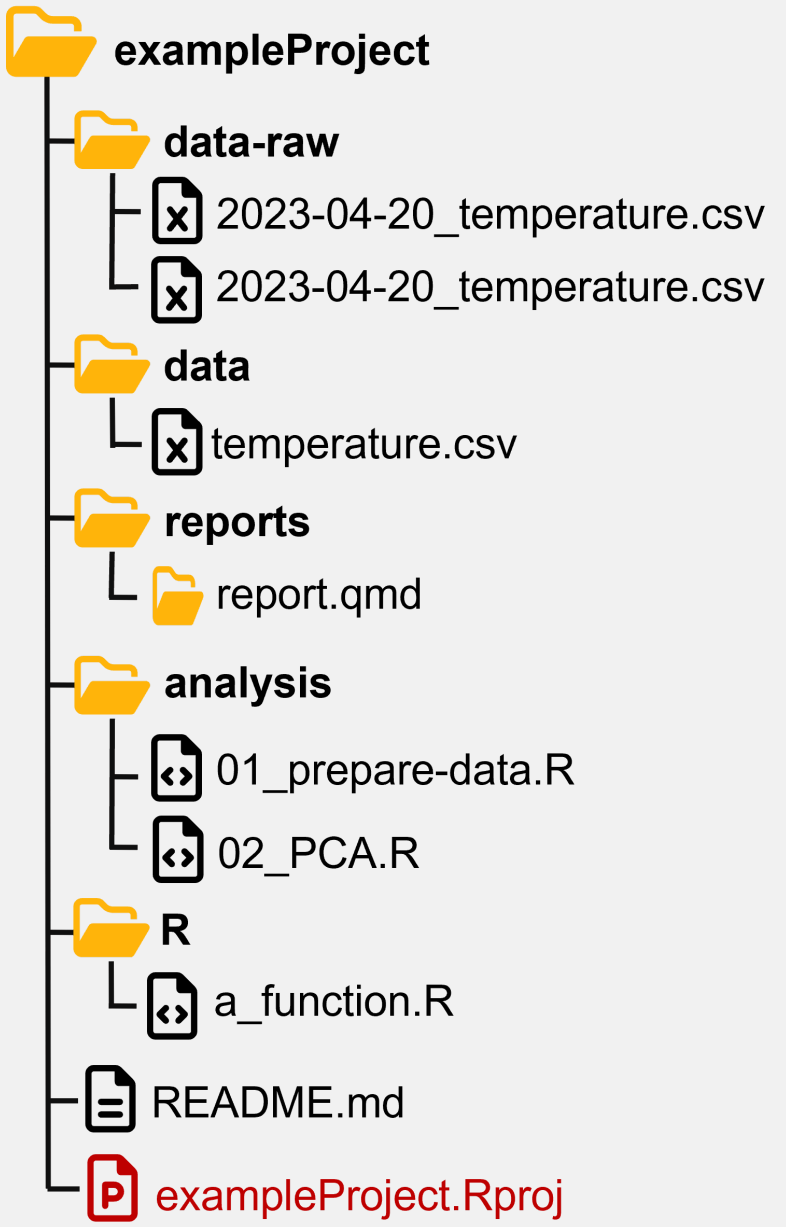
This is **not reproducible!** Your computer at exactly this time is the only one that has this working directory.

Use save paths

Option 1: Use **RStudio projects**

- Project root is automatically the working directory
- No need to use `setwd()` manually
- Read and write files with relative paths:

```
read_csv("data/temperature.csv")
```



Create an RStudio Project

From scratch:

1. `File -> New Project -> New Directory -> New Project`
2. Enter a directory name (this will be the name of your project)
3. Choose the directory where the project should be initiated
4. `Create Project`

Associate an existing folder with an RStudio Project:

1. `File -> New Project -> Existing Directory`
2. Choose your project folder
3. `Create Project`

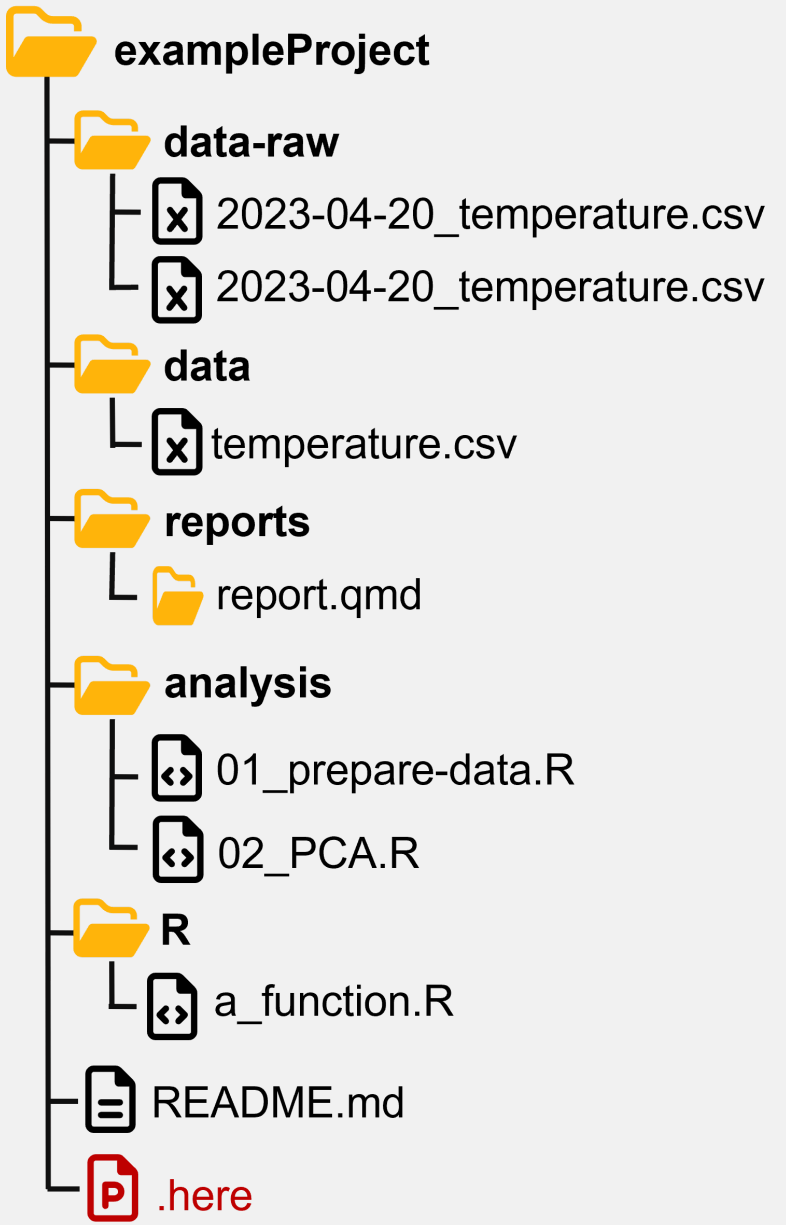
Use save paths

Option 2: Use the here package

- Build your paths using the `here` function

```
# use here::here
library(here)
read_csv(here("data", "temperature.csv"))
# or
read_csv(here("data/temperature.csv"))
```

- Reasonable heuristics to find project files
- Looks for `*.Rproj`, `.here`, `.git`, ...
- My recommendation: **always** use `here`
  - with `*.Rproj` OR with `.here`
  - works in all cases



Write well-structured and consistent code

Artwork by Allison Horst, CC BY 4.0

How to structure your scripts

- Use a standardized header
- Initialize at the top
  - `library()` calls
  - global options
  - source additional code
- Read all external data in one place
- Why?
  - fail fast
  - change code in one place

```
# Purpose: Create Figure 2 showing the relationship
# between body mass and bill length
# Authors: Selina Baldauf, Jane Doe, Jon Doe

# load libraries ---------------------------------
library(tidyverse)
library(vegan)

# Set global options -----------------------------
# Plot themes and colors
theme_set(theme_minimal())
custom_colors <- c("cyan4", "darkorange", "purple")

# Source additional code -------------------------
source("R/my_cool_function.R")

# Read data --------------------------------------
temperature <- read_csv("data/temperature.csv")
rainfall <- read_csv("data/rainfall.csv")
```

How to structure your scripts

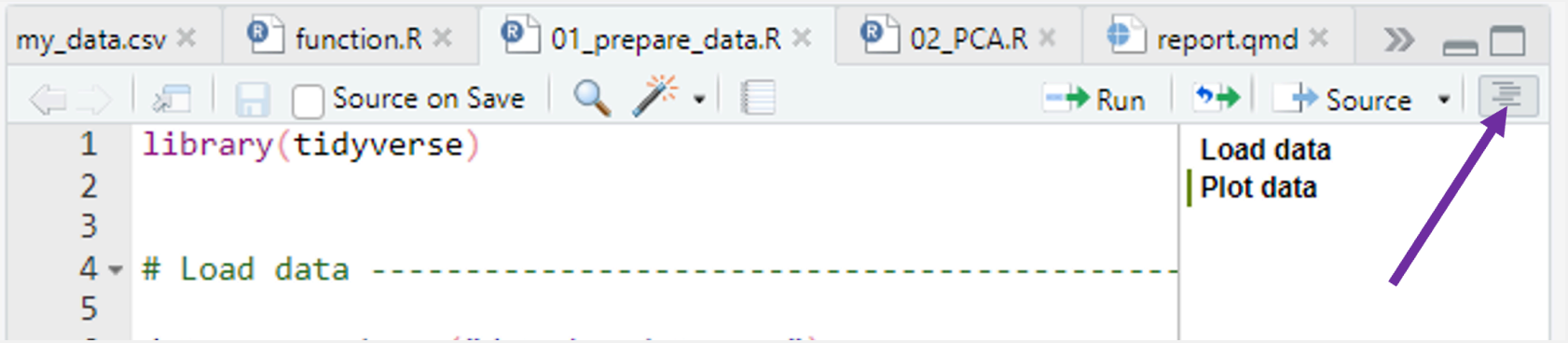
- Use headers to break up your file into sections

```
# Load data ------------------------------------------------------------

input_data <- read_csv(input_file)

# Plot data ------------------------------------------------------------

ggplot(input_data, aes(x = x, y = y)) +
  geom_point()
```

- Insert a section label with comment + at least 4 `####` or `----`
  - RStudio keyboard shortcut: `Ctrl/Cmd + Shift + R`

How to structure your scripts

- Navigate sections in the file outline



Use a consistent coding style - naming

Have a **naming convention** for variables and functions and stick to it

- Concise and descriptive (nouns for variables, verbs for functions)
- Consistent capitalization: Use `snake_case` for longer variable names
- Avoid conflicts with existing R functions and packages

```
# Good
day_one

clean_data()


# Bad
dayone
first_day_of_the_month
dm1

f1()
```

Use a consistent coding style - spacing

- Always put spaces after a comma

```
# Good
x[, 1]

# Bad
x[,1]
x[ , 1]
```

Use a consistent coding style - spacing

- Always put spaces after a comma
- No spaces around parentheses for normal function calls

```
# Good
mean(x, na.rm = TRUE)

# Bad
mean( x, na.rm = TRUE )
mean (x, na.rm = TRUE)
```

Use a consistent coding style - spacing

- Always put spaces after a comma
- No spaces around parentheses for normal function calls
- Spaces around most operators (<-, ==, +, etc.)

```
# Good
height <- (feet * 12) + inches
mean(x, na.rm = TRUE)

# Bad
height<-feet*12+inches
mean(x, na.rm=TRUE)
```

Use a consistent coding style - spacing

- Always put spaces after a comma
- No spaces around parentheses for normal function calls
- Spaces around most operators (<-, ==, +, etc.)
- Spaces before pipes (|>, %>%) and + in ggplot followed by new line

```
# Good
iris|>group_by(Species)|>summarize_if(is.numeric, mean)|>ungroup()

# Bad
iris |>
  group_by(Species) |>
  summarize_all(mean) |>
  ungroup()
```

Use a consistent coding style - spacing

- Always put spaces after a comma
- No spaces around parentheses for normal function calls
- Spaces around most operators (<-, ==, +, etc.)
- Spaces before pipes (|>, %>%) and + in ggplot followed by new line

```
# Good
ggplot(aes(x = Sepal.Width, y = Sepal.Length, color = Species))+geom_point()

# Bad
ggplot(aes(x = Sepal.Width, y = Sepal.Length, color = Species)) +
  geom_point()
```

Use a consistent coding style - width

- Always put spaces after a comma
- No spaces around parentheses for normal function calls
- Spaces around most operators (<-, ==, +, etc.)
- Spaces before pipes (|>, %>%) and + in ggplot followed by new line
- Limit your line width to 80 characters.

```
# Bad
iris |> group_by(Species) |> summarise(Sepal.Length = mean(Sepal.Length), Sepal.Width = mean(Sepal.Width), S

# Good
iris |>
  group_by(Species) |>
  summarise(
    Sepal.Length = mean(Sepal.Length),
    Sepal.Width = mean(Sepal.Width),
    Species = n_distinct(Species)
  )
```

Use a consistent coding style

Do I really have to remember all of this?

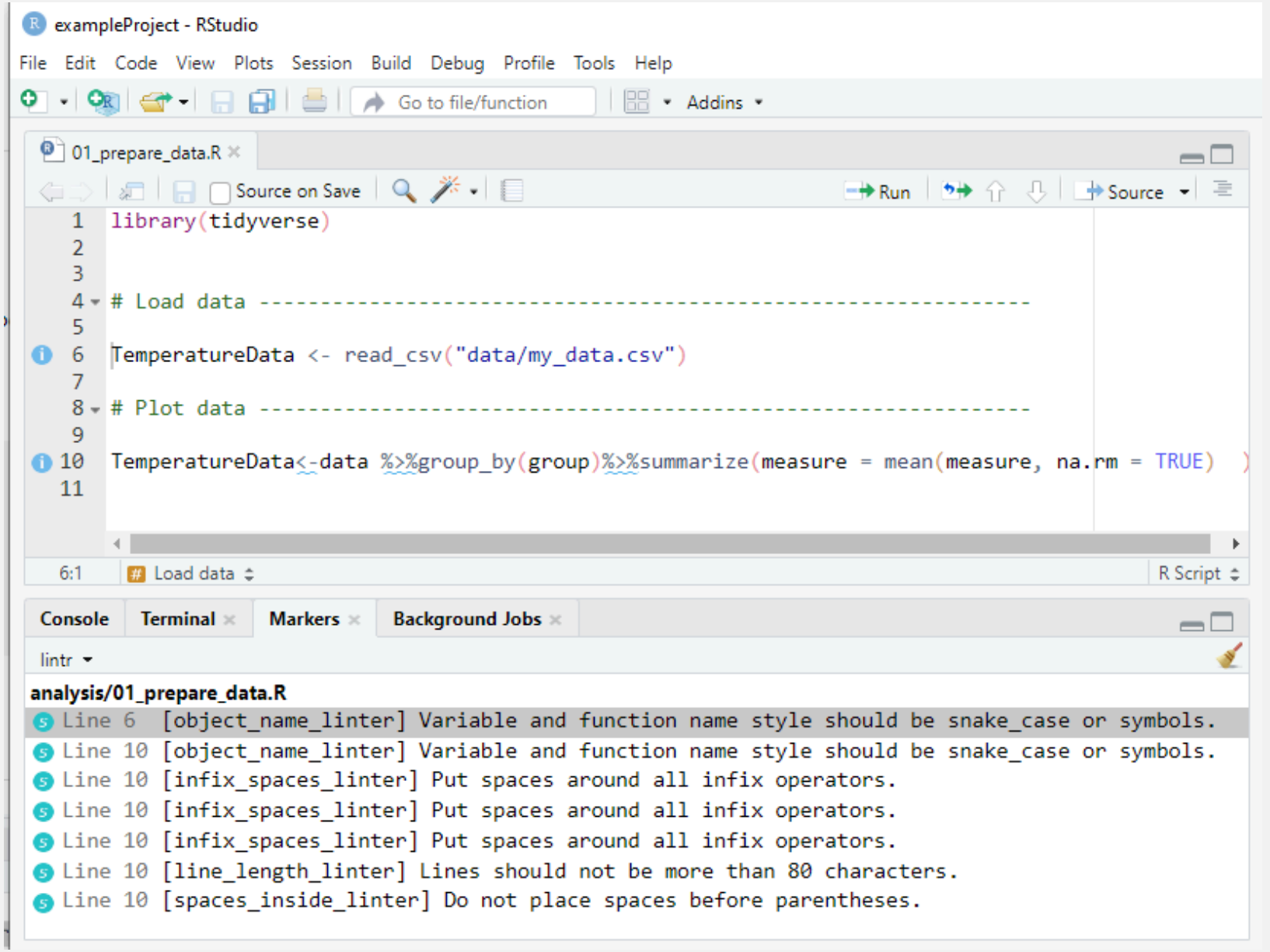
Luckily, no! R and RStudio provide some nice helpers

Coding style helpers - {lintr}

The lintr package analyses your code files or entire project and tells you what to fix.

```
# install the package before you can use it
install.packages("lintr")
# lint specific file
lintr::lint(filename = "analysis/01_prepare_data.R")
# lint a directory (by default the whole project)
lintr::lint_dir()
```

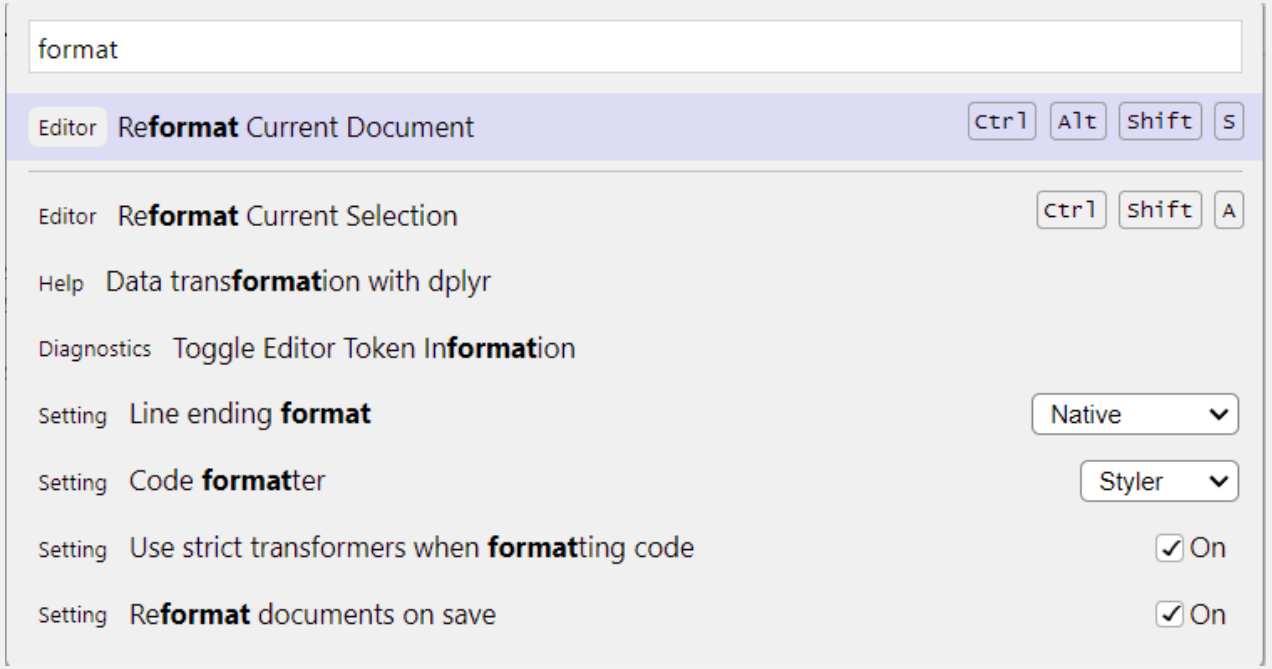
Coding style helpers - `lintr`



Coding style helpers - Auto-formatting

IDEs offer auto-formatting tools.

- Auto-format your scripts on save and let the IDE do the job
- RStudio: Open command palette (`Tools` -> `Show command palette`), search for "format" -> "Reformat documents on save"



Modularize long scripts

One huge script is **hard to maintain**

- Break down long scripts into logical units
- Write scripts that do one thing, e.g.
  - `01_prepare-data.R`: Read raw data and prepare it for analysis
  - `02_run-models.R`: Run statistical analysis
  - `03_make-figures.R`: Create manuscript figures
- Call these scripts sequentially
- Use `source()` to source R scripts in other scripts

```
# source data preparation in other scripts
source("01_prepare-data.R")
```

Modularize long scripts

Write a main **workflow script** that calls scripts in the right order.

- Often called `make.R`, `run.R` or `main.R`

```
# Prepare the data
source("01_prepare-data.R")
# Run all statistical models
source("02_run-models.R")
# Create manuscript figures
source("03_make-figures.R")
```

- Gives a nice overview of your workflow
- Very user-friendly: No need to run scripts separately and in the right order

Don't repeat yourself (DRY)

Example: Same data preparation code for multiple data sets

```
# Read the data
my_data1 <- readr::read_csv("data/data1.csv")
my_data2 <- readr::read_csv("data/data2.csv")
# Clean and summarize data
my_data1 <- my_data1 |>
  summarize(
    height = mean(height),
    biomass = mean(biomass),
    .by = c(country, species)
  )
my_data2 <- my_data2 |>
  summarize(
    height = mean(height),
    biomass = mean(biomass),
    .by = c(country, species)
  )
```

What's the **problem**?

Don't repeat yourself (DRY)

Solution: If you notice that you copy-paste code - write a **function**

Function in `R/prepare_data.R`:

```
prepare_data <- function(file_path) {
  # Read the data
  my_data <- read_csv(filepath)
  # Clean and summarize data
  my_data <- my_data |>
    summarize(
      height = mean(height),
      biomass = mean(biomass),
      .by = c(country, species)
    )
  # Return the cleaned data from the function
  return(my_data)
}
```

Source and use the `prepare_data` function in all scripts where it is needed

```
# Source the function
source("R/prepare_data.R")

# Use the function to read and prepare the data
my_data1 <- prepare_data("data/data1.csv")
my_data2 <- prepare_data("data/data2.csv")
```

Manage dependencies

Manage dependencies manually

Add the output of `devtools::session_info()` to your `README`

```
devtools::session_info()
```

```
─ Session info ─────────────────────────────────────────────────────────────
 setting  value
 version  R version 4.4.2 (2024-10-31 ucrt)
 os       Windows 11 x64 (build 26100)
 system   x86_64, mingw32
 ui       RTerm
 language (EN)
 collate  English_Germany.utf8
 ctype    English_Germany.utf8
 tz       Europe/Berlin
 date     2025-05-13
 pandoc   3.6.3 @ C:\\Users\\Selina\\AppData\\Local\\Programs\\Quarto\\bin\\tools/ (via rmarkdown)

─ Packages ─────────────────────────────────────────────────────────────────
 package     * version date (UTC) lib source
 cachem        1.1.0   2024-05-16 [1] CRAN (R 4.4.2)
 cli           3.6.3   2024-06-21 [1] CRAN (R 4.4.2)
 devtools      2.4.5   2022-10-11 [1] CRAN (R 4.4.2)
```

Manage dependencies with {renv}

Idea: Have a **project-local environment** with all packages needed by the project

- Keep log of the packages and versions you use
- Restore the local project library on other machines

Why this is useful?

- Code will still work even if packages upgrade
- Collaborators can recreate your local project library with one function
- Explicit dependency file states all dependencies

Check out the renv website for more information

Manage dependencies with {renv}

```
# Get started
install.packages("renv")
```

Very simple to use and integrate into your project workflow:

```
# Step 1: initialize a project level R library
renv::init()
# Step 2: save the current status of your library to a lock file
renv::snapshot()
# Step 3: restore state of your project from renv.lock
renv::restore()
```

- Others only need to install the `renv` package, then they can also call `renv::restore()`

Conclusion

Take aways

- So many tips and guidelines 😵
- Prioritize progress over perfection
  - Implement simple things first (e.g. format on save)
  - Other guidelines will become habits over time
- Plan ahead

Clean projects and workflows ...

... help you to write robust and reproducible code.

Artwork by Allison Horst, CC BY 4.0

Outlook

Of course there is much that I left out:

- Proper documentation
- Version control with Git
- Containerization (e.g. Docker) for full reproducibility
- Using R packages to build a research compendium
- ...

But this is for another time

Next lecture

Topic t.b.a.

📅 19th June 🕛 4-5 p.m. 📍 Webex

- For topic suggestions and/or feedback send me an email
- Subscribe to the mailing list

Thank you for your attention :)

Questions?

References

- What they forgot to teach you about R book by Jenny Bryan and Jim Hester
- Blogpost by Jenny Bryan on good project-oriented workflows
- R best practice blogpost by Krista L. DeStasio
- Book about coding style for R: The tidyverse style guide
- The Turing way book General concepts and things to think about regarding reproducible research
- renv package website
- lintr package website