# Good practice R coding

Day 1 - Introduction to Data Analysis with R

Selina Baldauf
Freie Universität Berlin - Theoretical Ecology

March 14, 2025

## Chaotic projects and workflows ...

... can make even small changes frustrating and difficult.

Artwork by Allison Horst, CC BY 4.0

## Background

Often, we want to **share** and **publish** our projects.

- Reproducibility 🔄
  - Can someone else reproduce my results?
- Reliability 🏋️
  - Will my code work in the future?
- Reusability ⚙️
  - Can someone else actually use my code?

## Set up your project properly

Consistent structure and filenames

## Have a clear project structure

- One directory with all files relevant for project
  - Scripts, data, plots, documents, …
- Choose a meaningful project structure [1]
- Add a readme file (usually `README.md`) in which you document the project structure

```
MyProject
|
|- data
|
|- docs
|    |
|    |- notes
|    |
|    |- reports
|
|- R
|    |
|    |- clean_data.R
|    |
|    |- statistics.R
|
|- MyProject.RProj
|
|- README.md
```

Example RStudio project structure

1. you can orient yourself at the R package structure

## Use RStudio projects

Always make your project an RStudio Project (if possible)!

✅ You already did that.

## Set up your project

R Studio offers a lot of settings and options.

So have a ☕ and check out **Tools -> Global Options** and all the other buttons.

- R Studio cheat sheet that explains all the buttons
- Update R Studio from time to time to get new settings (**Help -> Check for Updates**)

## 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. From this talk by J. Bryan

## 1. Machine readable file names

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

- No space & special characters

### 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

Which file names would you like to read at 4 a.m. in the morning?

- File names should reveal the file content
- Use separators to make it readable

### Bad examples ✗

```
01preparedata.R
01firstscript.R
```

### Good examples ✓

```
01_prepare-data.R
01_temperature-trend-analysis.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

> Good practice R coding

## Write beautiful code

- Try to write code that others (i.e. future you) can understand
- Follow standards for readable and maintainable code
    - For R: tidyverse style guide defines code organization, syntax standards, …

Artwork by Allison Horst, CC BY 4.0

# Standard code structure

1. General comment with purpose of the script, author, …
2. `library()` calls on top
3. Set default variables and global options
4. Source additional code
5. Write the actual code, starting with loading all data files

```r
# This code replicates figure 2 from the
# Baldauf et al. 2022 Journal of Ecology
# paper.
# Authors: Selina Baldauf and Jane Doe
# Copyright Selina Baldauf (2024)

library(tidyverse)
library(vegan)

# set defaults
input_file <- "data/results.csv"

# source files
source("R/my_cool_function.R")

# read input
input_data <- read_csv(input_file)
```

## Mark sections

- Use comments to break up your file into sections

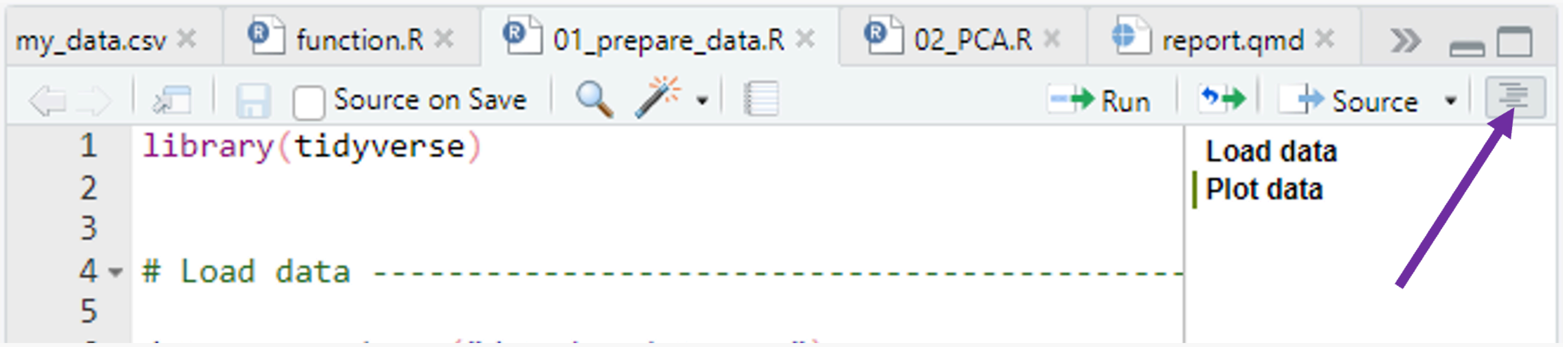
```
# Load data --------------------------------------------------------------

input_data <- read_csv(input_file)

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

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

- Insert a section label with `Ctrl/Cmd + Shift + R`
- Navigate sections in the file outline



```
library(tidyverse)


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

## Break down large scripts

- If your scripts become too big, split them
- You can use `source()` in R to load the content of another script
    - You can source the same script into multiple other scripts

`R/analyse_species.R`:

```r
# This code ....
# Authors: Selina Baldauf and Jane Doe
# Copyright Selina Baldauf (2024)

library(tidyverse)
library(vegan)

# source file for data reading and cleaning
# reads in the species data and cleans it for
# further analysis
# output is the tibble species_clean
source("R/prepare_data/read_and_clean_data.R")

# Analyse species data ----------------------

# code for the analysis
```

`R/prepare_data/read_and_clean_data.R`

```r
# Script to read in the raw species data and clean it

path_to_species <- "data/species.csv"

species_raw <- read_csv(path_to_species)

# further code for cleaning the data
```

## Coding style - Object names

- Variables should only have *lowercase letters*, *numbers*, and *_*
- Use `snake_case` for longer variable names
- Try to use concise but meaningful names

```
# Good
day_one
day_1

# Bad
DayOne
dayone
first_day_of_the_month
dm1
```

## Coding style - Spacing

- Always put spaces after a comma

```r
# Good
x[, 1]

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

## Coding style - Spacing

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

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

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

## Coding style - Spacing

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

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

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

## Coding style - Spacing

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

```r
# Good
iris |>
  summarize_if(is.numeric, mean, .by = Species)  |>
  arrange(desc(Sepal.Length))

# Bad
iris|>summarize_if(is.numeric, mean, .by = Species)|>arrange(desc(Sepal.Length))
```

## Coding style - Spacing

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

```r
# 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()
```

## Coding style - Line width

Try to limit your line width to 80 characters.

- You don't want to scroll to the right to read all code
- 80 characters can be displayed on most displays and programs
- Split your code into multiple lines if it is too long
  - See this grey vertical line in R Studio?

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

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

## Coding style

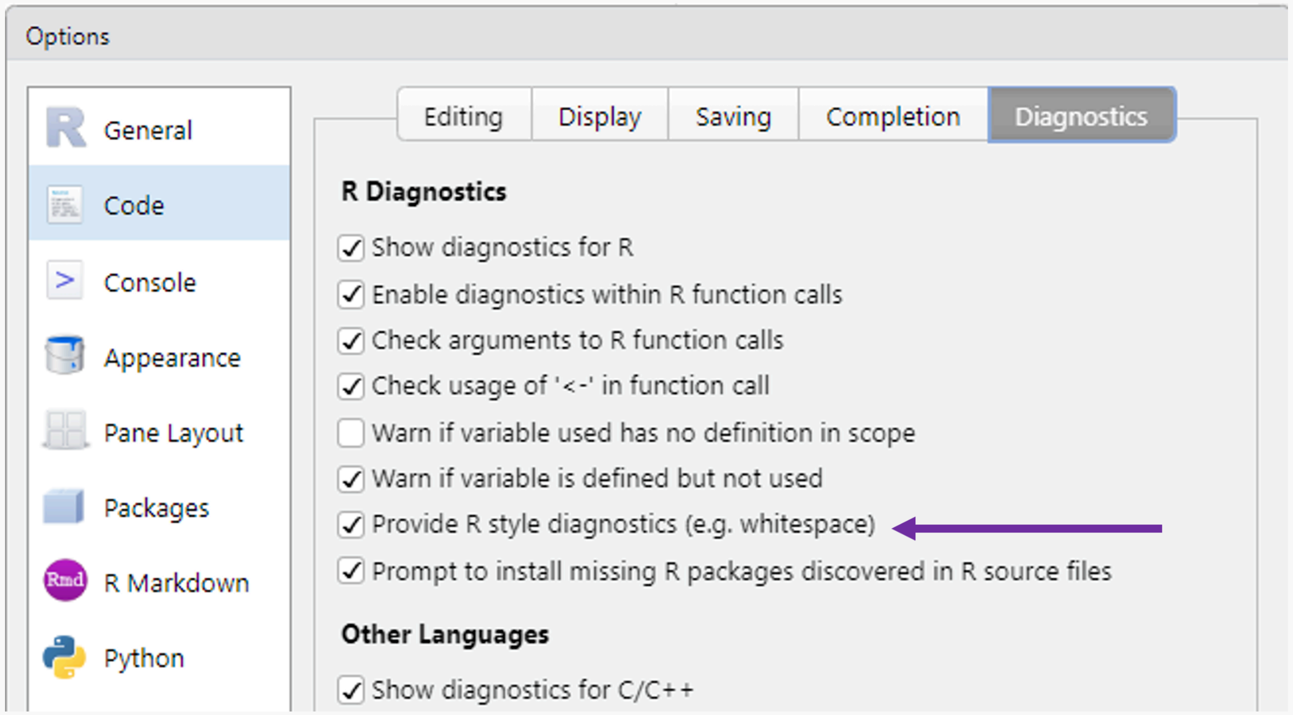
Do I really have to remember all of this?

Luckily, no! R and R Studio provide some nice helpers

# Coding style helpers - RStudio

RStudio has style diagnostics that tell you where something is wrong

Tools -> Gloabl Options -> Code -> Diagnostics

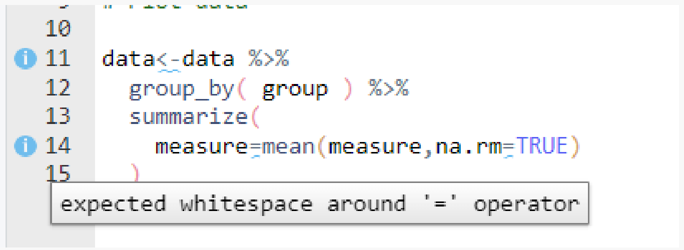


**R Diagnostics**

- [✓] Show diagnostics for R
- [✓] Enable diagnostics within R function calls
- [✓] Check arguments to R function calls
- [✓] Check usage of '<-' in function call
- [ ] Warn if variable used has no definition in scope
- [✓] Warn if variable is defined but not used
- [✓] Provide R style diagnostics (e.g. whitespace)
- [✓] Prompt to install missing R packages discovered in R source files

**Other Languages**

- [✓] Show diagnostics for C/C++

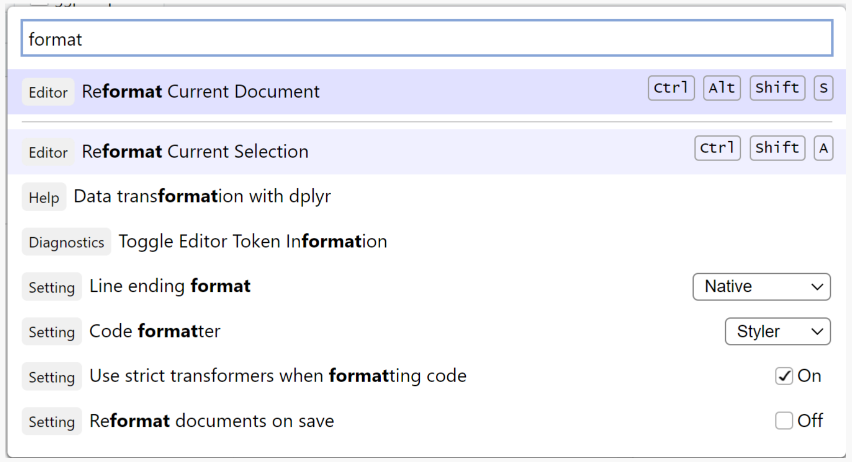


```
data<-data %>%
  group_by( group ) %>%
  summarize(
    measure=mean(measure,na.rm=TRUE)
  )
```

```
expected whitespace around '=' operator
```

## Coding style helpers - Auto-formatting

RStudio can automatically format your code!


format

Editor **Reformat** Current Document Ctrl Alt Shift S

Editor **Reformat** Current Selection Ctrl Shift A

Help Data trans**formation** with dplyr

Diagnostics Toggle Editor Token In**formation**

Setting Line ending **format** Native

Setting Code **formatter** Styler

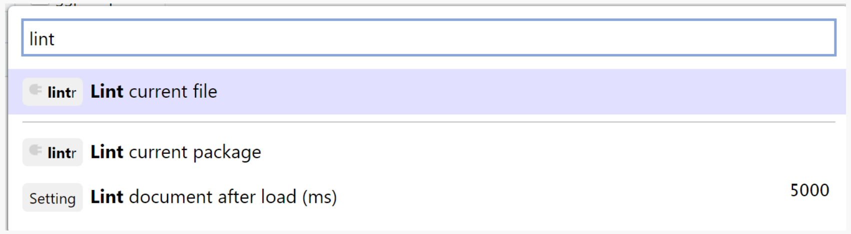
Setting Use strict transformers when **formatting** code ✓ On

Setting **Reformat** documents on save ☐ Off

- Open the command palette with `Ctrl/Cmd + Shift + P`
- Search for “format” to see all formatting options
  - “Reformat Current Document” to format the open script
  - You can toggle “Reformat documents on save” to format automatically on save
  - By default the “Code formatter” is the tidyverse style guide (Styler)

## Coding style helpers - Linting

- Linters are tools that analyze your code for potential errors
- The R package `lintr` is a linter for R code
  - Install it with `install.packages("lintr")`
- Open the command palette with `Ctrl/Cmd + Shift + P` and search for "lint"
  - "Lint current file" to lint the open script
  - This will print a list of potential improvements in the console area



lint

lintr **Lint** current file

lintr **Lint** current package

Setting **Lint** document after load (ms) 5000

## Let’s share

> Publish or share your analysis

## Make your code sharable

- Use relative paths!
- Make sure you include all necessary files
- Include a readme with instructions on how to run the code and an explanation of the project structure
- Comment your code to make it understandable

**Send your code to a colleague**

If you are unsure if your code is understandable and can run on another machine, send it to a colleague and ask them to run it and give you feedback

## Publishing your code

- Include a licence
  - Choose a licence (common for open source projects: MIT, GPL-3.0)
  - Use the `usethis` package to add a licence to your project
    - `install.packages("usethis")`
    - `usethis::use_mit_license("Your Name")` adds a licence file to your project
- Show which packages and which versions you used
  - this is important for reproducibility as packages change over time
  - `devtools::session_info()` prints a list of all packages and their versions (add this to your readme file)
  - `renv` is a package that helps you manage package versions
- Consider learning Git and publish on GitHub/Gitlab etc.

## Clean projects and workflows …

… allow you and others to work productively.

But don't get overwhelmed by all the advice. Just start with one thing.

Artwork by Allison Horst, CC BY 4.0