Advanced Data Analysis with R

👋 Welcome

- 🤓 Sebastian Hanß - Ecologist, PhD student for some years and now scientific programmer
- 🌿 Ecosystem Modelling Lab @ University of Göttingen
- Dynamic simulation models, spatial point patterns, theoretical ecology
- Contact: shanss@uni-goettingen.de

💡 We are always happy to supervise student projects and theses with a modeling and coding focus!

🎓 Classes that might interest you

- Agent-based modelling with NetLogo (700268, beginners at any level, SoSe, block in late summer)
- Introduction to Ecological Modelling (700549, MSc Forest and Ecosystem Sciences, WiSe)
- Ecological Simulation Modelling (700562 + 700563, MSc Forest and Ecosystem Sciences, SoSe)
- Spatial Statistics (700294, MSc Forest and Ecosystem Sciences, PhD, SoSe)
- Ecological Modelling with C++ (700300, MSc, WiSe)

Advanced Data Analysis with R

Goals

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

Today: Talk best practice rules to write clean, clear and maintainable code.

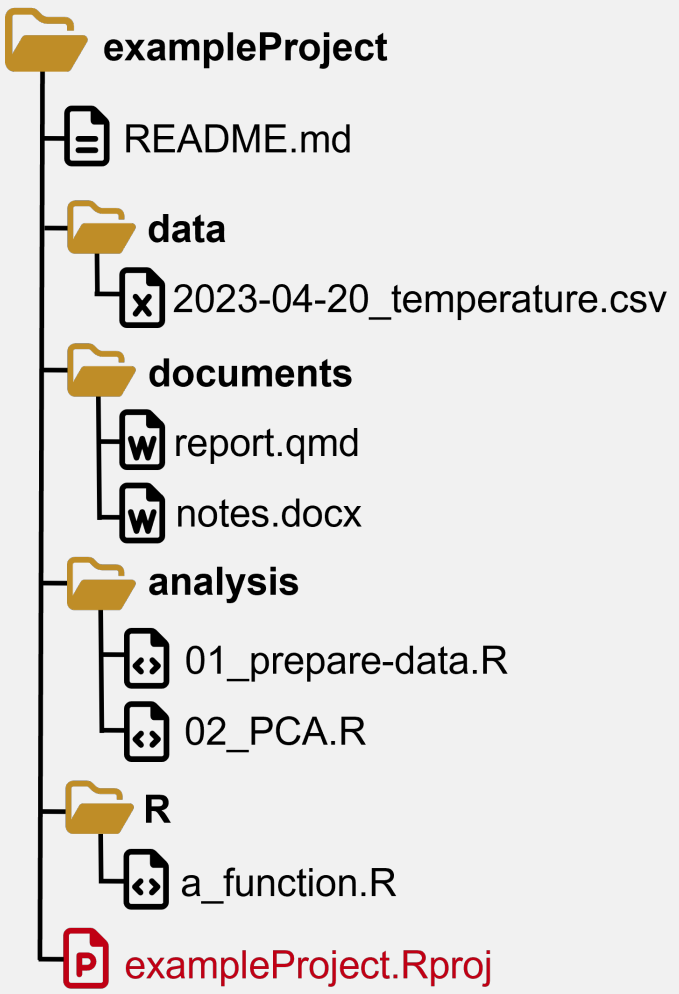
First things first

Project setup and structure

Use R Studio projects

Always make your project an R Studio Project (if possible)!

- Keep your files together
- An R Studio Project is just a normal directory with an `*.Rproj` file
  - double-click this file to open your project in R Studio
- Advantages:
  - Easy to navigate in R Studio
  - Project root is the working directory
  - Open multiple projects in separate R Studio instances



Create an R Studio 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`

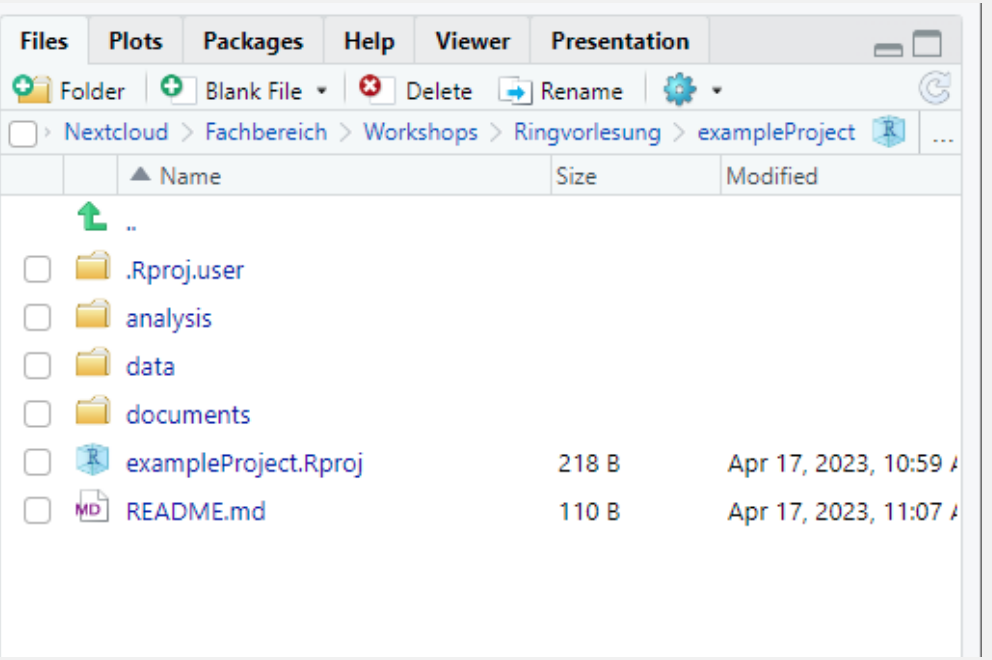
Associate an existing folder with an R Studio Project:

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

Navigate an R Studio Project

You can use the `Files` pane in R Studio to interact with your project folder:

- Navigate and open files
- Create files and folders
- Rename and delete
- …



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.

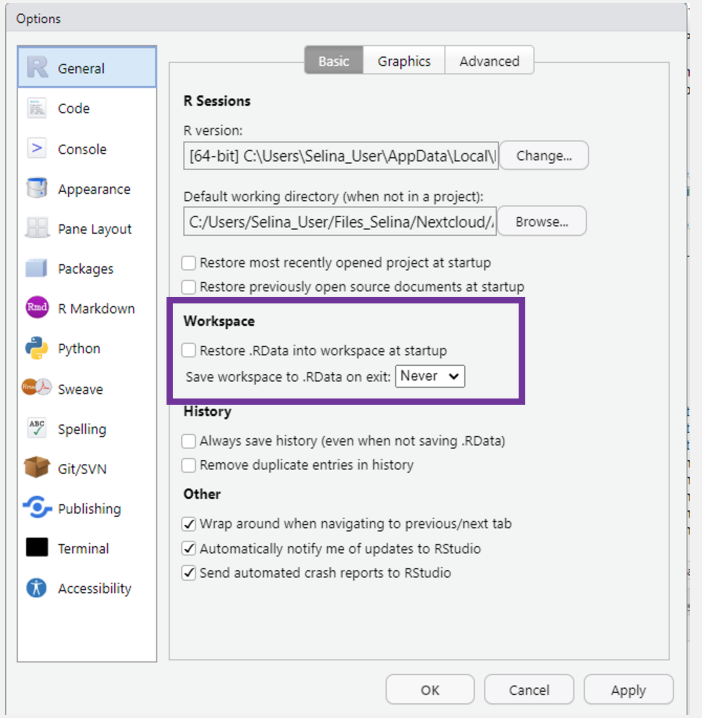
- R Studio cheat sheet that explains all the buttons
- Update R Studio from time to time to get new settings (`Help -> Check for Updates`)

Set up your project

R Studio offers a lot of settings and options.

Most important setting for reproducibility:

- Never save or restore your work space as `.Rdata` -> You always want to start working with a clean slate



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 ❌

📄 `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

Write beautiful code

Artwork by Allison Horst, CC BY 4.0

- 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, …

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

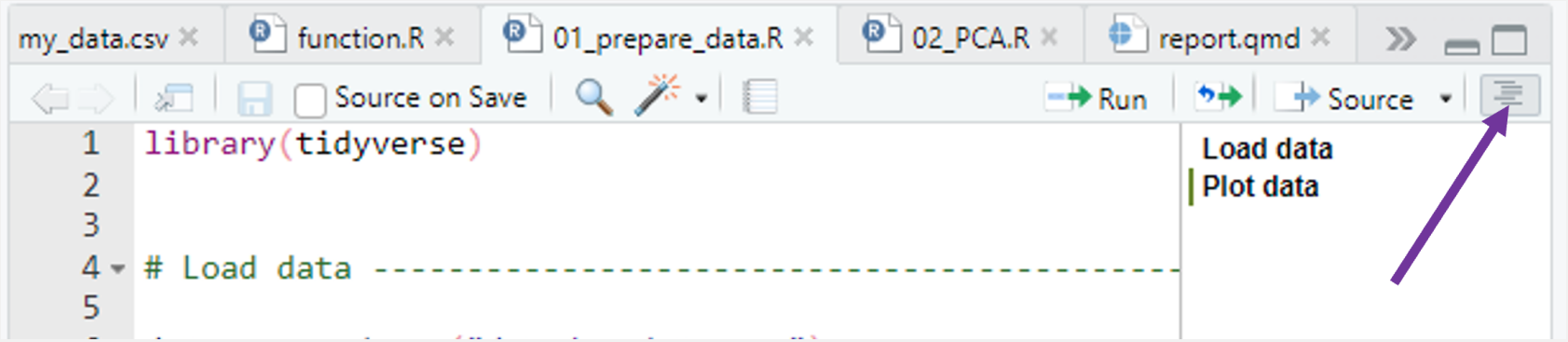
```
# This code replicates figure 2 from the
# Baldauf et al. 2022 Journal of Ecology paper.
# Authors: Selina Baldauf, Jane Doe, Jon Doe

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



Modularize your Code

- Don't put all your code into one long file (hard to maintain)
  - Write multiple files that can be called sequentially
    - E.g. `01_prepare-data.R`, `02_lm-temperature-trend.R`, `03_plot-temperture-trends.R`
  - Write functions that can be called in other scripts
    - Use the `source()` function to source these files
    - Have one main workflow script that calls these functions sequentially

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. But to this Jenny Bryan said:

> If the first line of your R script is
> `setwd("C:\Users\jenny\path\that\only\I\have")`
> I will come into your office and SET YOUR COMPUTER ON FIRE 🔥.

Why?

This is 100% not reproducible: Your computer at exactly this time is (probably) the only one in the world that has this working directory

. . .

Avoid `setwd()` if it is possible in any way!

Avoid `setwd()`

Use R Studio projects

- Project root is automatically the working directory
- Give your project to a friend at it will work on their machine as well

Instead of

```
# my unique path from hell with white space and special characters
setwd("C:/Users/Selina's PC/My Projects/Göttingen Temperatures/temperatures")

read_csv("data/2023-04-20_temperature_goettingen.csv")
```

You just need

```
read_csv("data/2023-04-20_temperature_goettingen.csv")
```

If you don't use R Studio Projects, have a look at the `{here}` package for reproducible paths

Coding style - Object names

- Variables and function names 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

```
# 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

```
# 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.)

```
# 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 pipes (%>%, |>) followed by new line

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

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

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 (%>%, |>) followed by new line
- Spaces before + 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()
```

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?

```
# Bad
iris %>%
    group_by(Species) %>%
    summarise(Sepal.Length = mean(Sepal.Length), Sepal.Width = mean(Sepal.Width), Species = n_distinct(Speci

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

Coding style

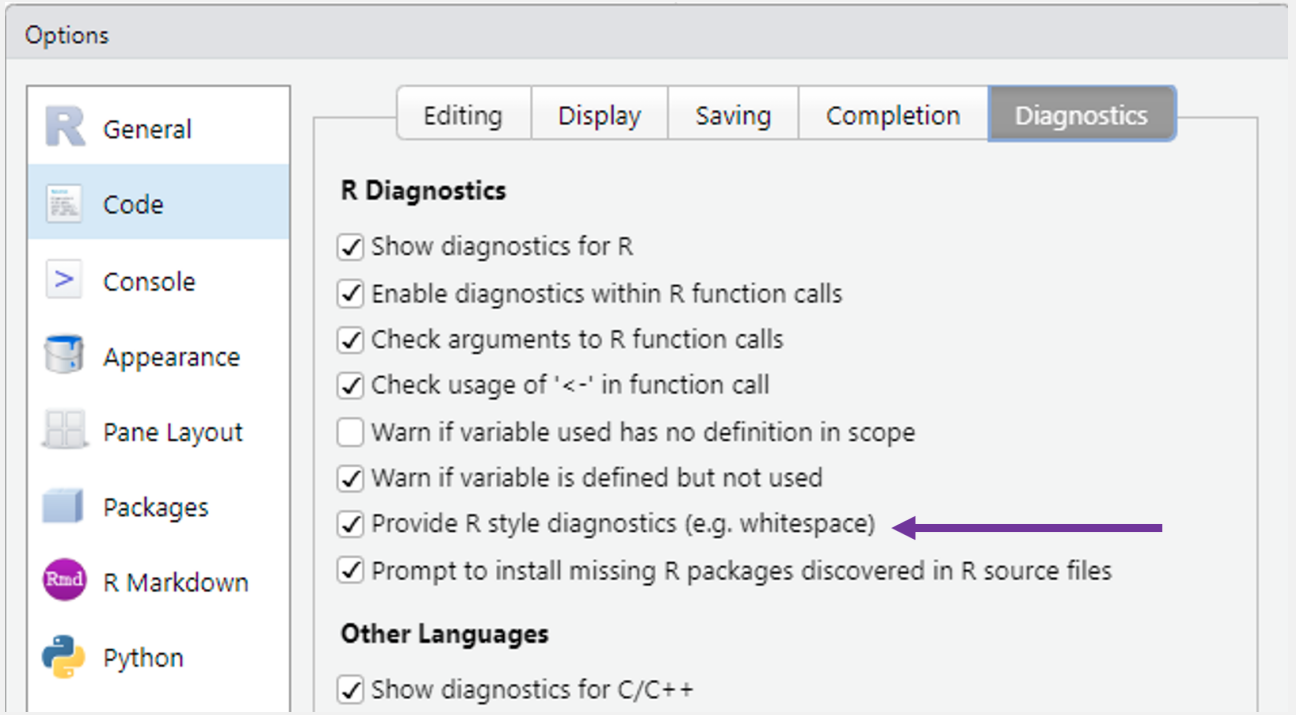
Do I really have to remember all of this?

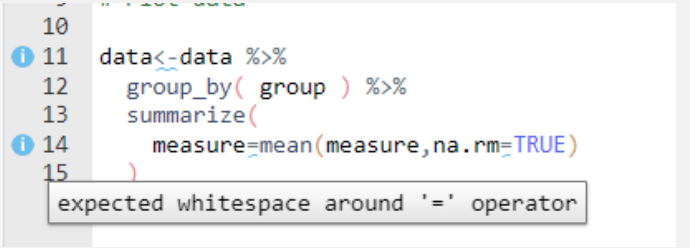
Luckily, no! R and R Studio provide some nice helpers

Coding style helpers - R Studio

R Studio has style diagnostics that tell you where something is wrong

- `Tools -> Gloabl Options -> Code -> Diagnostics`



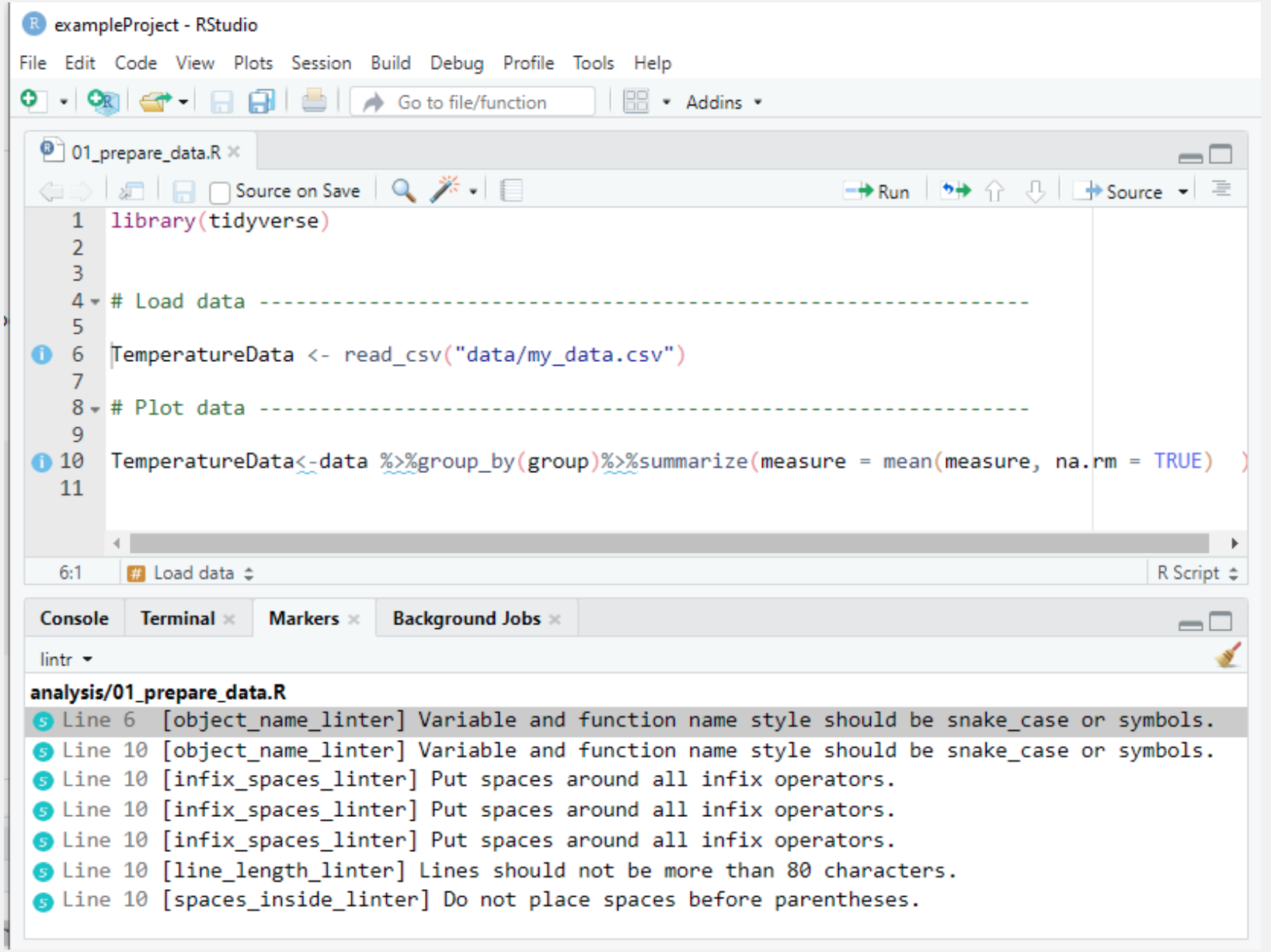


Coding style helpers - {lintr}

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}

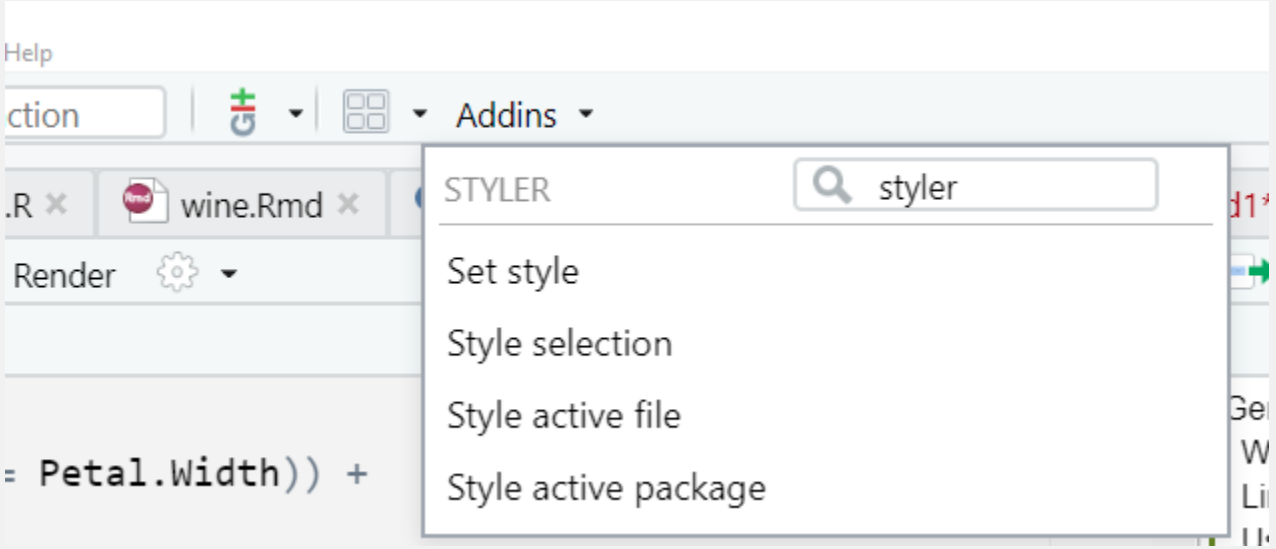


Coding style helpers - {styler}

The styler package package automatically styles your files and projects according to the tidyverse style guide.

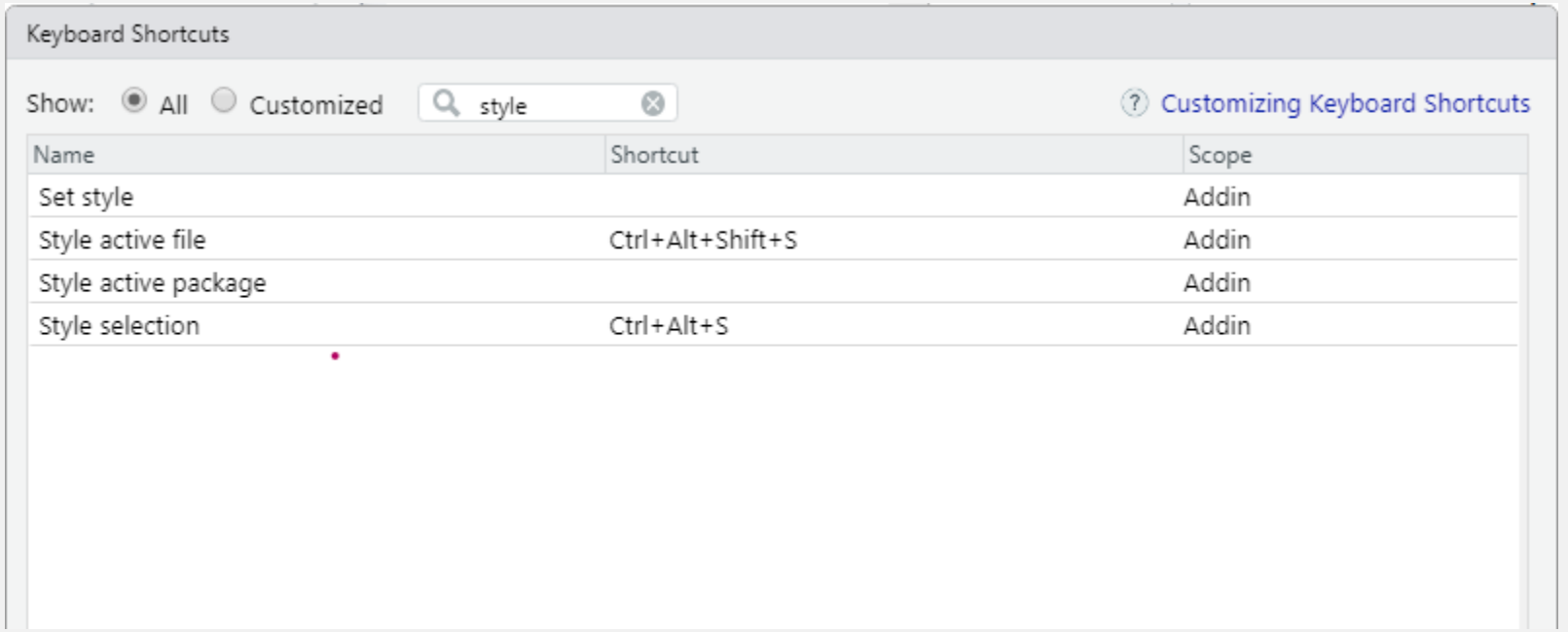
```
# install from CRAN
install.packages("styler")
```

- Use the R Studio `Addins` for styler:



Coding style helpers - `{styler}`

- Pro-Tip: Add a custom keyboard short cut to style your files
  - `Tools -> Modify Keyboard Shortcuts`



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()
```

- Your collaborators only need to install the `renv` package, then they can also call `renv::restore()`
- When you create an R Studio project there is a check mark to initialize with `renv`

Take aways Part 1

There are a lot of things that require minimal effort and that you can start to implement into your workflow NOW

1. Use R Studio projects -> Avoid `setwd()`!
2. Keep your projects clean

- Sort your files into folders
- Give your files meaningful names

3. Use `styler` to style your code automatically
4. Use `lintr` and let R analyse your project
5. Consider `renv` for project local environments

🛠️ Hands-on: Create your own R Project (~10-15 min)

- open `exercises_day_2.qmd` in RStudio and do the first exercise (*Create a Research Compendium with `usethis`*)
- play around with the `styler` and `lintr` packages

Further Reading

- Slides: Tools and tips lecture by Selina Baldauf (FU Berlin)
- 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
- styler package website
- lintr package website

💻 Recap Programming Basics

- Variables Vectors, Control flow
- 🔁 Loops and Vectorization
- 🧩 Functions

Variables Vectors, Control flow

```r
x <- 1:10
y <- x^2
z <- ifelse(x > 5, "greater than 5", "less than or equal to 5")
```

🔁 For-loops in R

```
for (i in 1:length(x)) {
  print(x[i])
}
```

🔁 For-loops in R

- For-loops are relatively slow and it is easy to make them even slower with bad design
- Often they are used when vectorized code would be better
- For loops can often be replaced, e.g. by
  - Functions from the apply family (e.g. `apply`, `lapply`, ...)
  - Vectorized functions (e.g. `sum`, `colMeans`, ...)
  - Vectorized functions from the `purrr` package (e.g. `map`)

But: For loops are not necessarily bad, sometimes they are the best solution and more readable than vectorized code.

Functions

- Functions are reusable blocks of code
- They take inputs (arguments) and return outputs
- Use functions to encapsulate logic and improve code readability

```
square <- function(x) {
  return(x^2)
}
square(5)
```

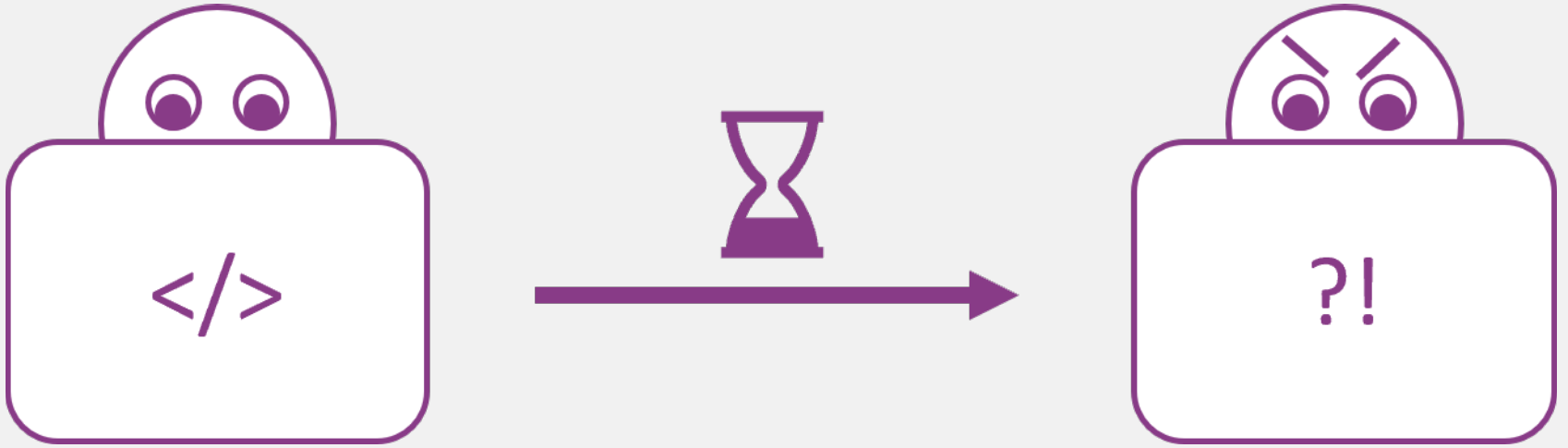
🛠️ Hands-on: Create your own functions (~ 10-15 min)

- open `exercises_day_2.qmd` in RStudio and do exercises 2, 3 and 4

Git & GitHub in RStudio

Motivation

Two examples in which proper version control can be a life/time saver



Requirements for good version control

- Complete and long-term history of every file in your project
- Safe (e.g. no accidental loss of versions)
- Easy to use
- Overview and documentation of all changes
- Collaboration should be possible

Version control with Git

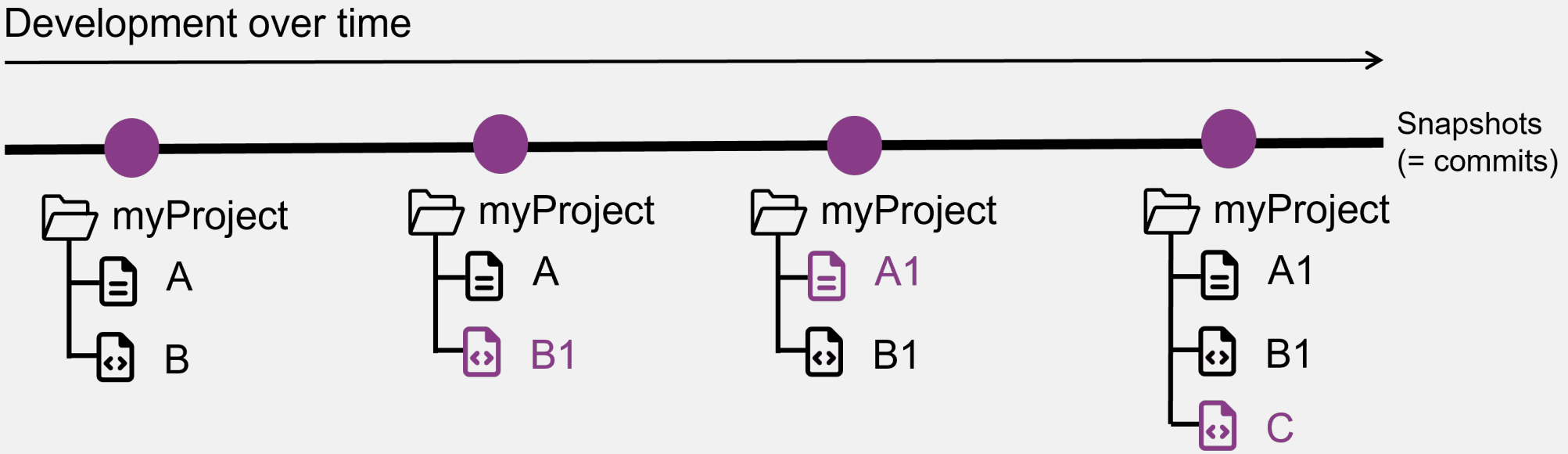
- Open source and free to use version control software
- Quasi standard for software development
- A whole universe of other software and services around it

Today

- Basic concepts of Git
- A simple workflow in theory and practice
- A small outlook on more advanced features

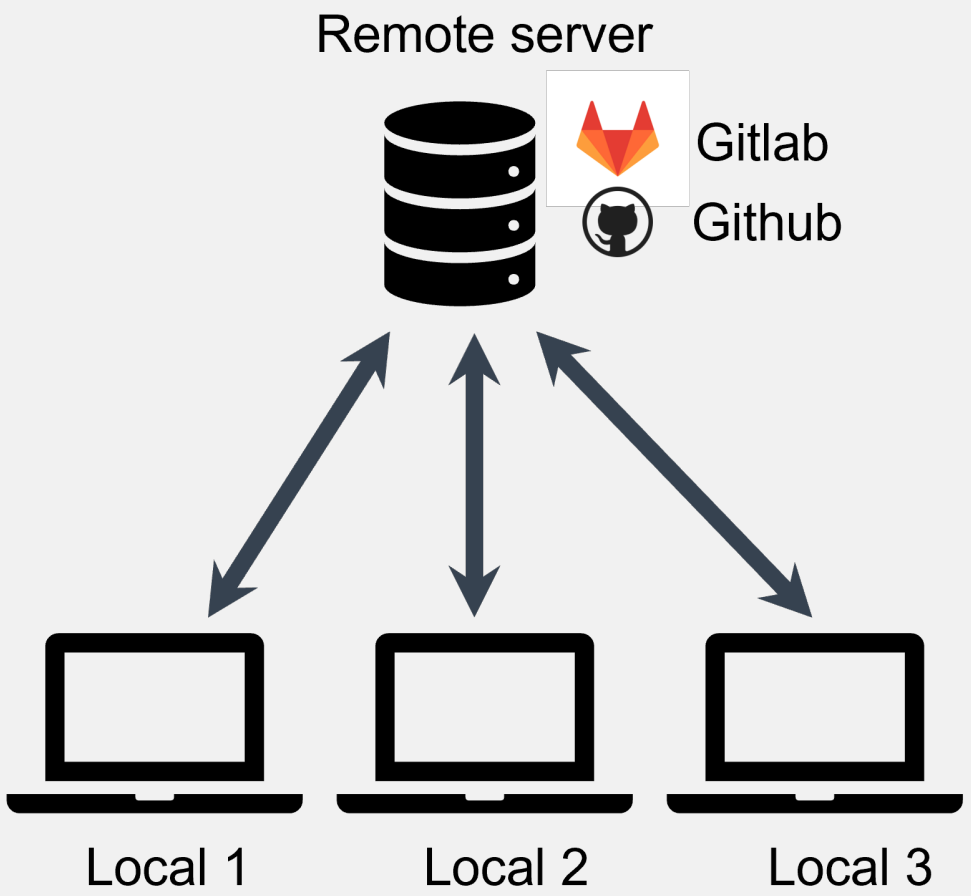
Version control with Git

- For projects with mainly text files (e.g. code, markdown files, …)
- Basic idea: Take snapshots of your project over time
  - Snapshots are called commits in Git
- A project that is version controlled with Git is called Git repository (or Git repo)



Version control with Git

Git is a distributed version control system



- Idea: many local repositories synced via one remote repo
- Every machine has full-fledged version of repository with entire history

How to use Git

After you installed it there are different ways to use the software for your projects

How to use Git - Terminal

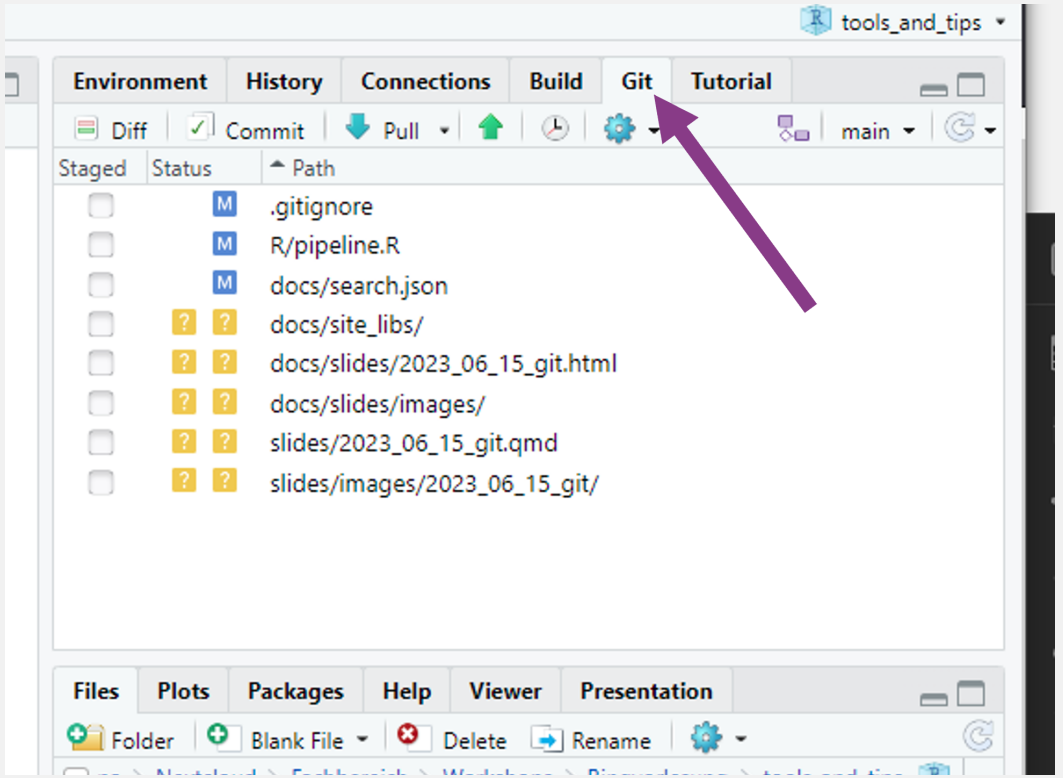
Using Git from the terminal

```
Selina_User@DESKTOP-G0RM7MS MINGW64 ~/Files_Selina
$ cd Repos/02_workshops/first_git_project/

Selina_User@DESKTOP-G0RM7MS MINGW64 ~/Files_Selina/Repos/02_workshops/first_git_
project
$ git init
Initialized empty Git repository in C:/Users/Selina_User/Files_Selina/Repos/02_w
orkshops/first_git_project/.git/

Selina_User@DESKTOP-G0RM7MS MINGW64 ~/Files_Selina/Repos/02_workshops/first_git_
project (master)
$
```

+ Gives you most control
+ You find a lot of help online
− You need to use the terminal

How to use Git - GUIs

A Git GUI is integrated in most (all?) IDEs, e.g. R Studio

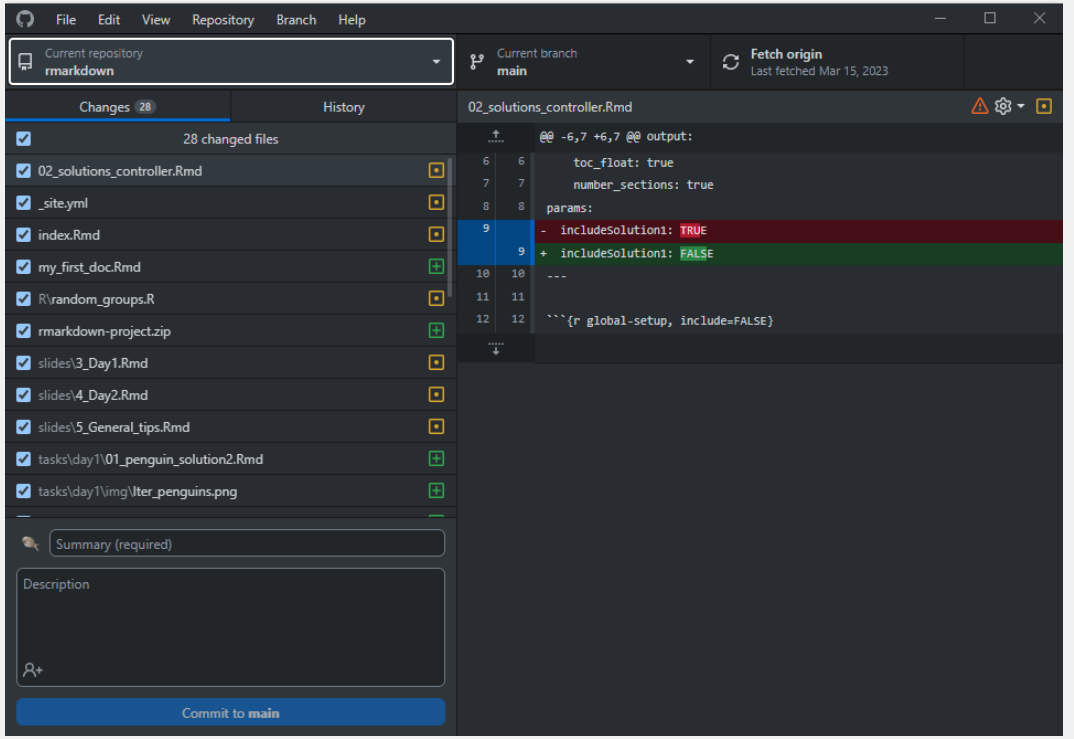


+ (Often) Easy and intuitive
+ Stay inside your IDE

− Not universal

How to use Git - GUIs

Standalone Git GUI software, e.g. Github Desktop



\+ Easy and intuitive

\+ Helps with initial setup of Git

\+ Nice integration with Github

— Switch program to use Git

How to use Git

Which one to choose?

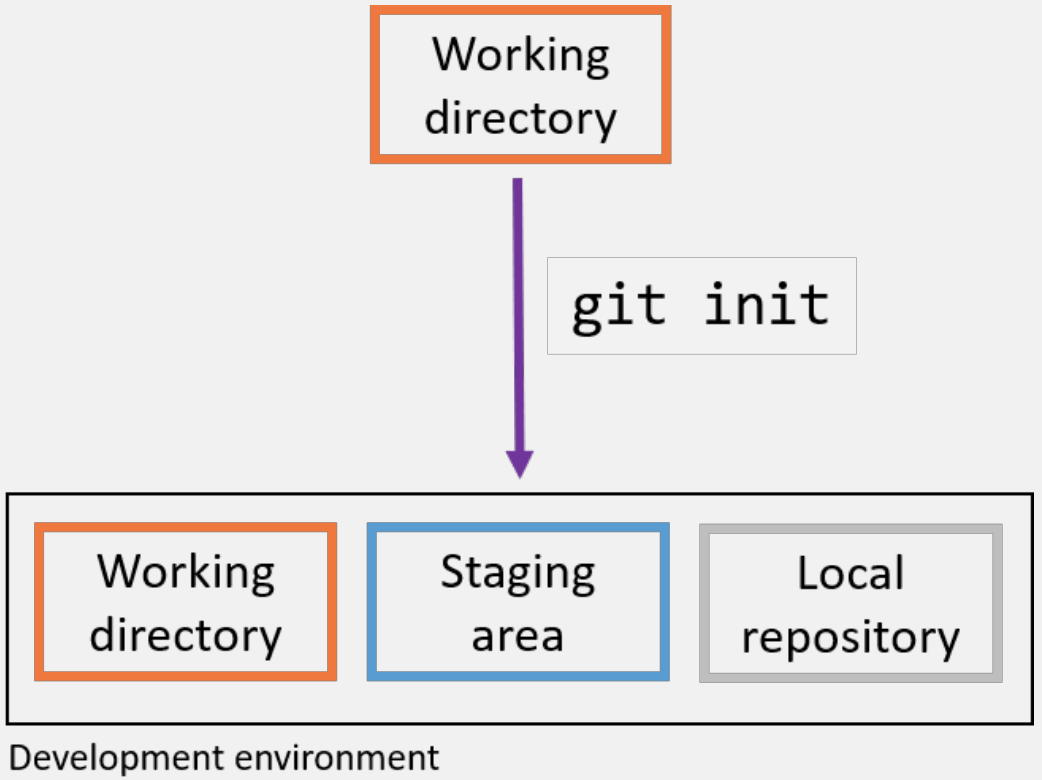
- Depends on your prior experience and taste
- If you never used the terminal before, I recommend to start with Github Desktop
  - But in the long run, it's definitely worth it looking into the terminal
- You can also mix methods and freely switch between them

The basic Git workflow

> `git init`, `git add`, `git commit`, `git push`

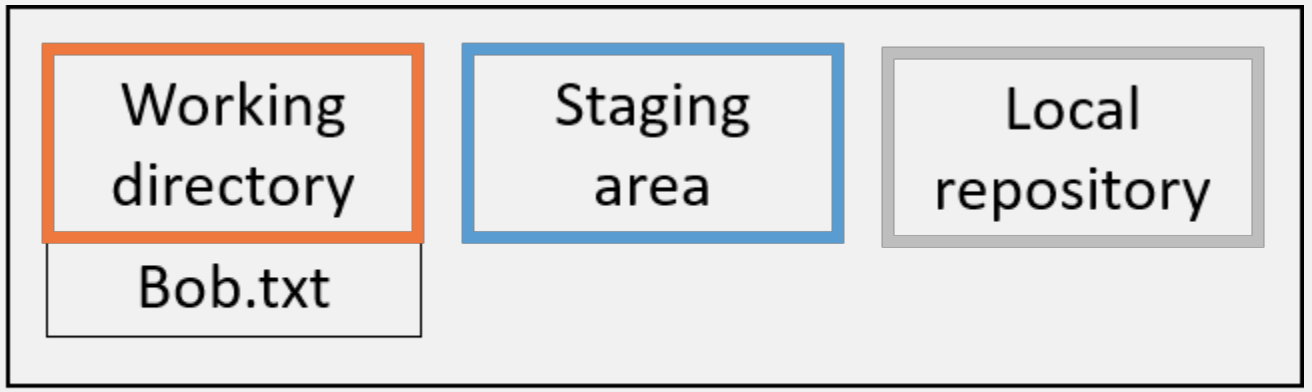
Step 1: Initialize a git repository

- Adds a (hidden) `.git` folder to your project that will contain the Git repository
- You don't have to touch anything that is in this folder



Step 2: Modify files and stage changes

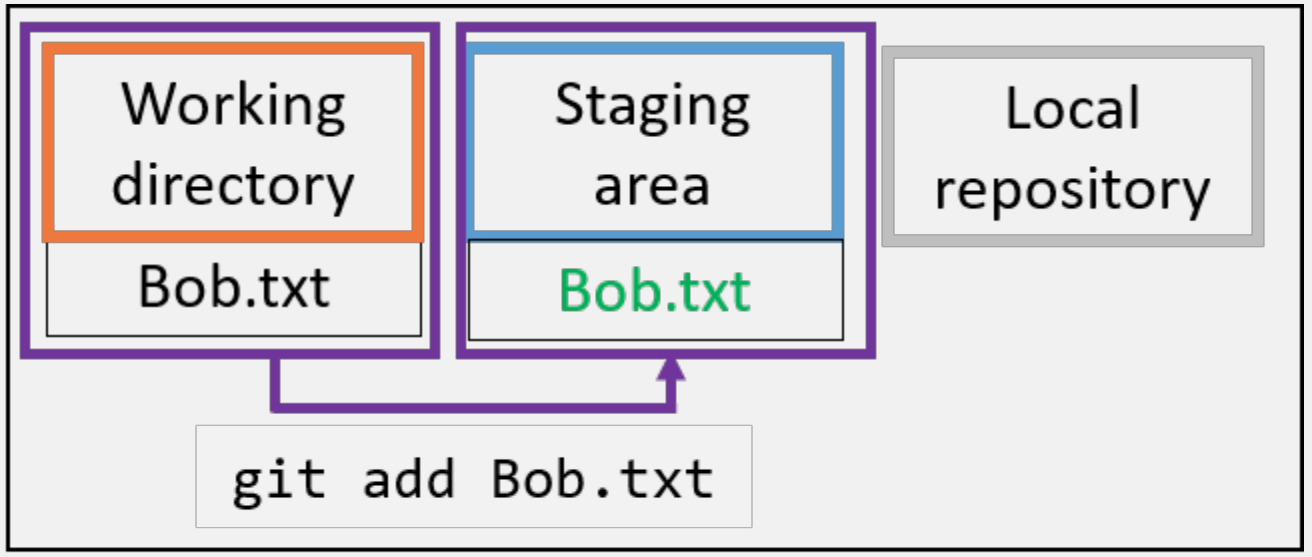
Git detects any changes in the working directory



Step 2: Modify files and stage changes

When you want a file to be part of the next commit (i.e. snapshot), you have to stage the file

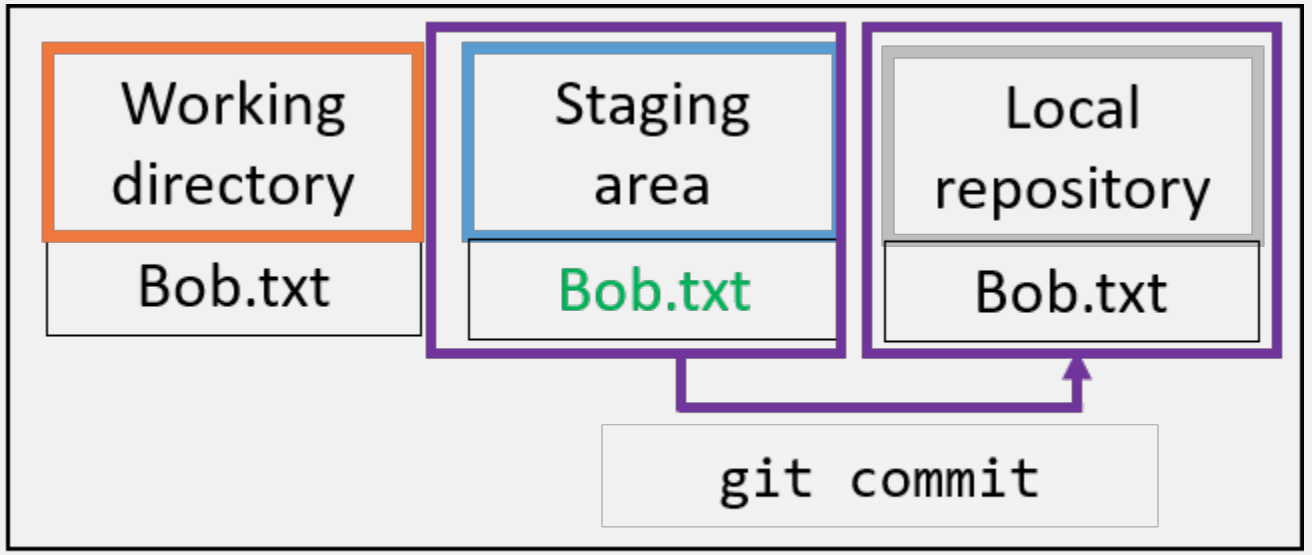
- In the terminal use `git add`
- Usually in Git GUIs this is just a check mark next to the file name



Development environment

Step 3: Commit changes

- Commits are the snapshots of your project states
- Commit work from staging area to local repository
  - Collect meaningful chunks of work in the staging area, then commit



- After a commit, your changes are part of your project's git history

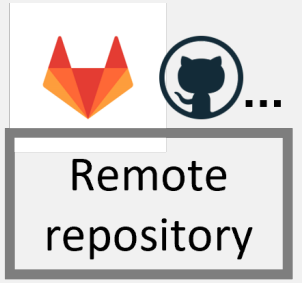
Step 3: Commit changes

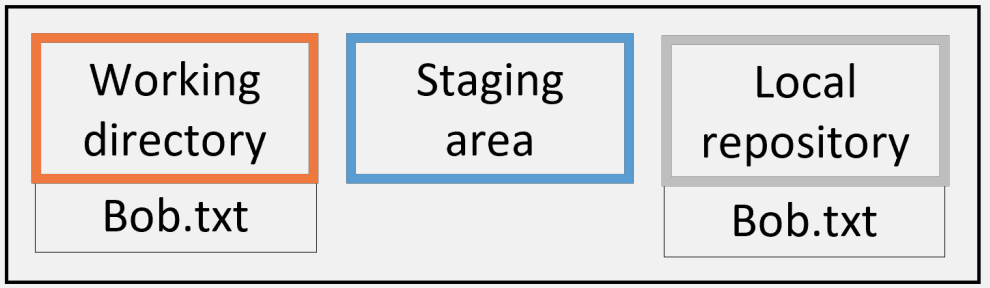
- Every commit has a unique identifier (so-called hash)
  - You can use this hash to come back to the version
- Every commit has a commit message that describes what the changes are about

Read here for a nice and detailed explanation on good commit messages how to write them.

Step 4: Create and connect a remote repo

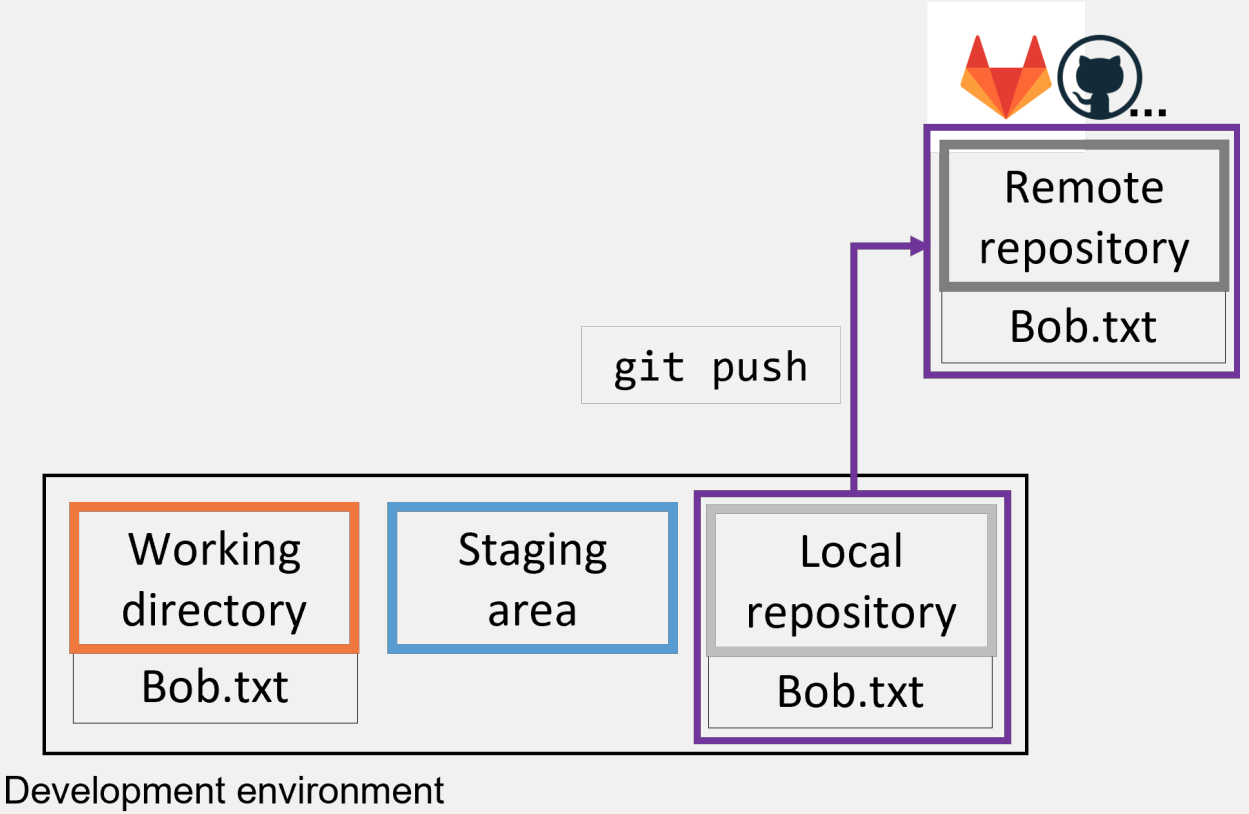
- Remote repositories are on a server and can be used to *synchronize*, *share* and *collaborate*
- Remote repositories can be private (only for you and selected collaborators) or public (visible to anyone online)





Step 5: Share your changes with the remote repo

- Push your local changes to the remote with `git push`

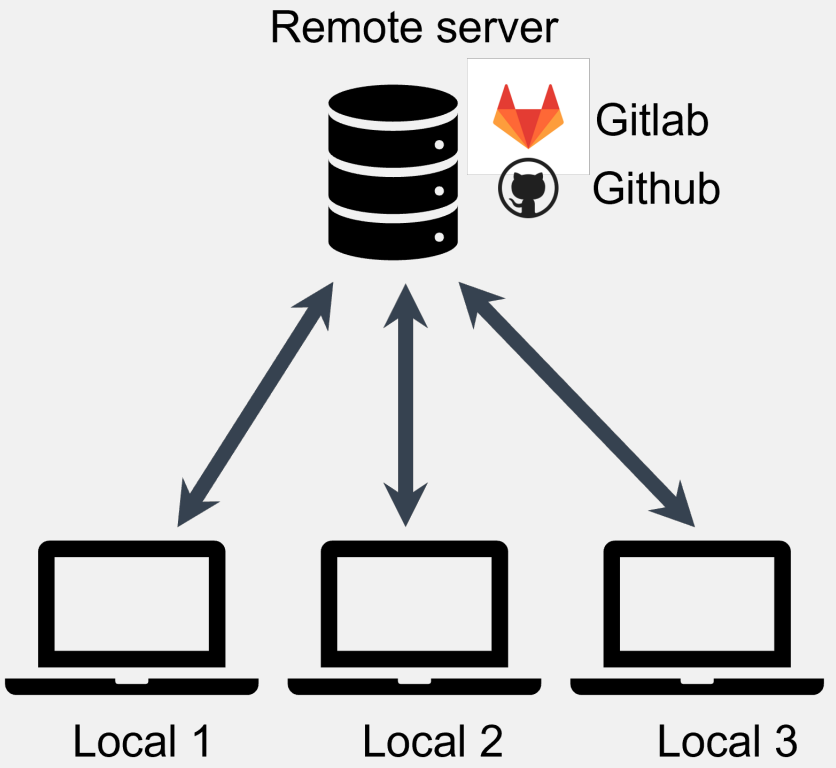


Summary of the basic steps

- `git init`: Initialize a git repository
  - adds a `.git` folder to your working directory
- `git add`: Add files to the staging area
  - This marks the files as being part of the next commit
- `git commit`: Take a snapshot of your current project version
  - Includes a timestamp, a meaningful commit message and information on the person who did the commit
- `git push`: Push your newest commits to the remote repository
  - Sync your local project version with the remote e.g. on Github

Synchronize, share and collaborate

Get a repo from a remote

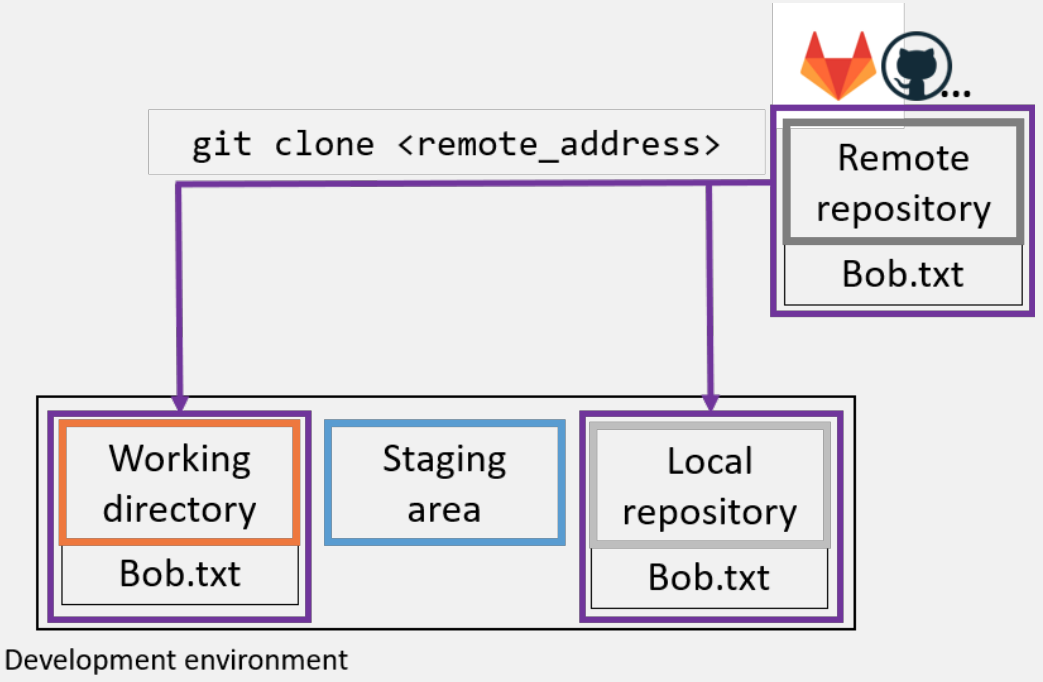


- In Git language, this is called cloning
- Get a copy of your own repository on a different machine
- Get the repository from somebody else

Get a repo from a remote

By cloning, you get a full copy of the repository and the working directory with all files on your machine.

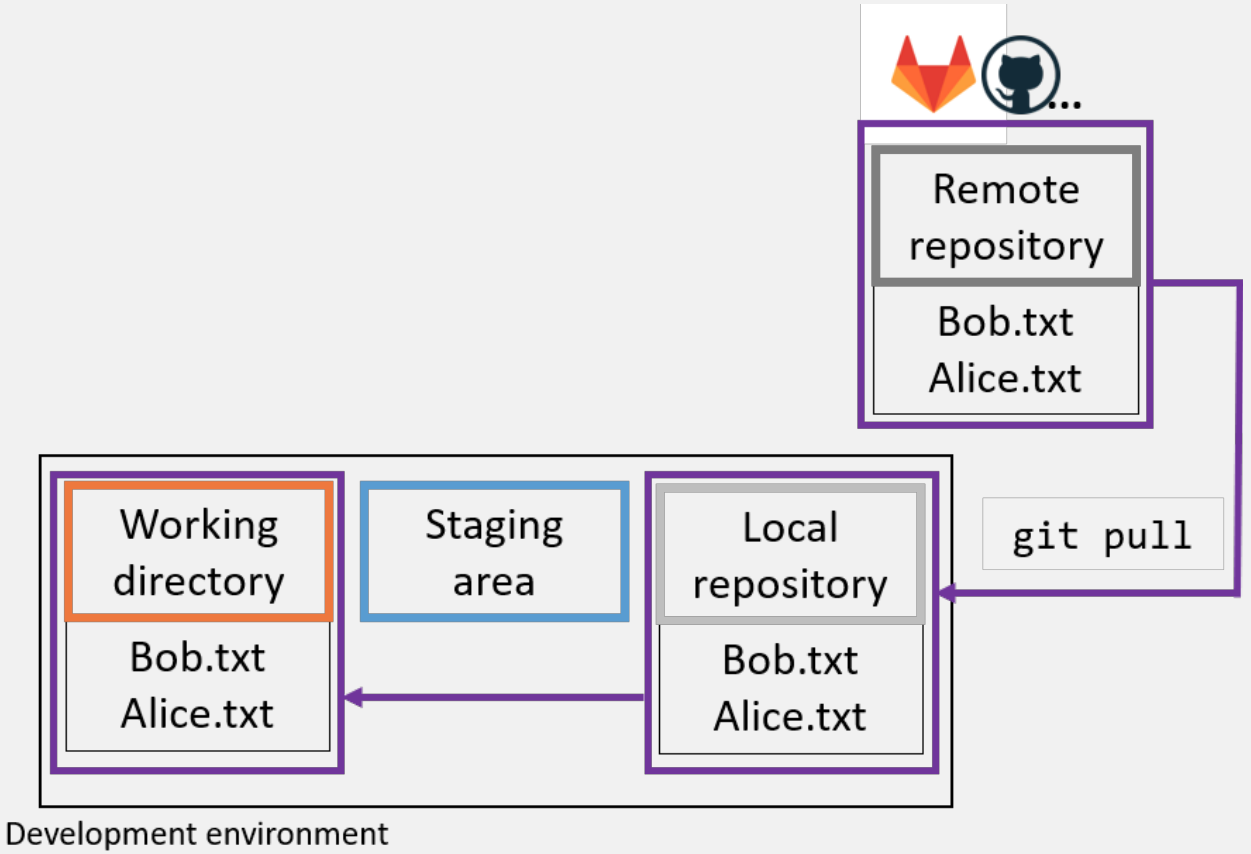
- Clone a remote repository with `git clone <remote_address>`



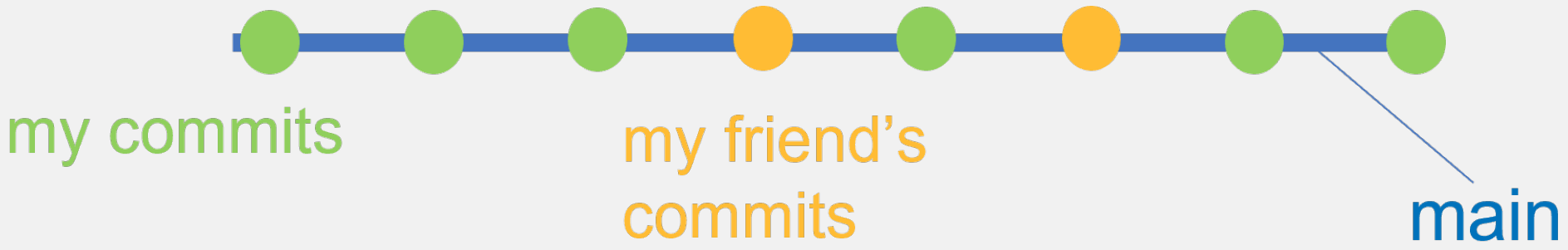
- If the clone is authorized it can also commit and push

Get changes from the remote

- Local changes, publish to remote: `git push`
- Remote changes, pull to local: `git pull`



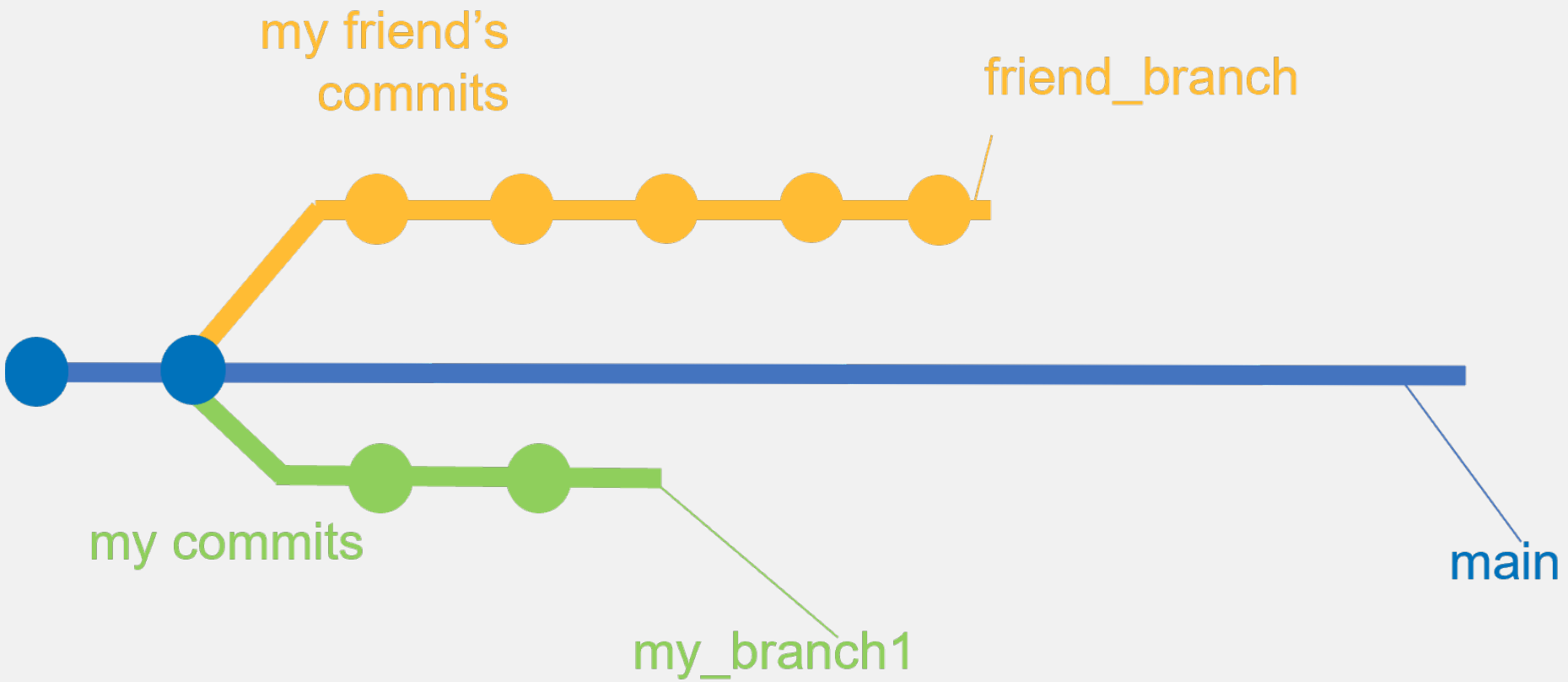
A simple collaboration workflow



- By default: Everything on one branch (main)
  - Branches are connections between specific commits
- Basic idea: Pull newest version before you start working, push new version after you are done

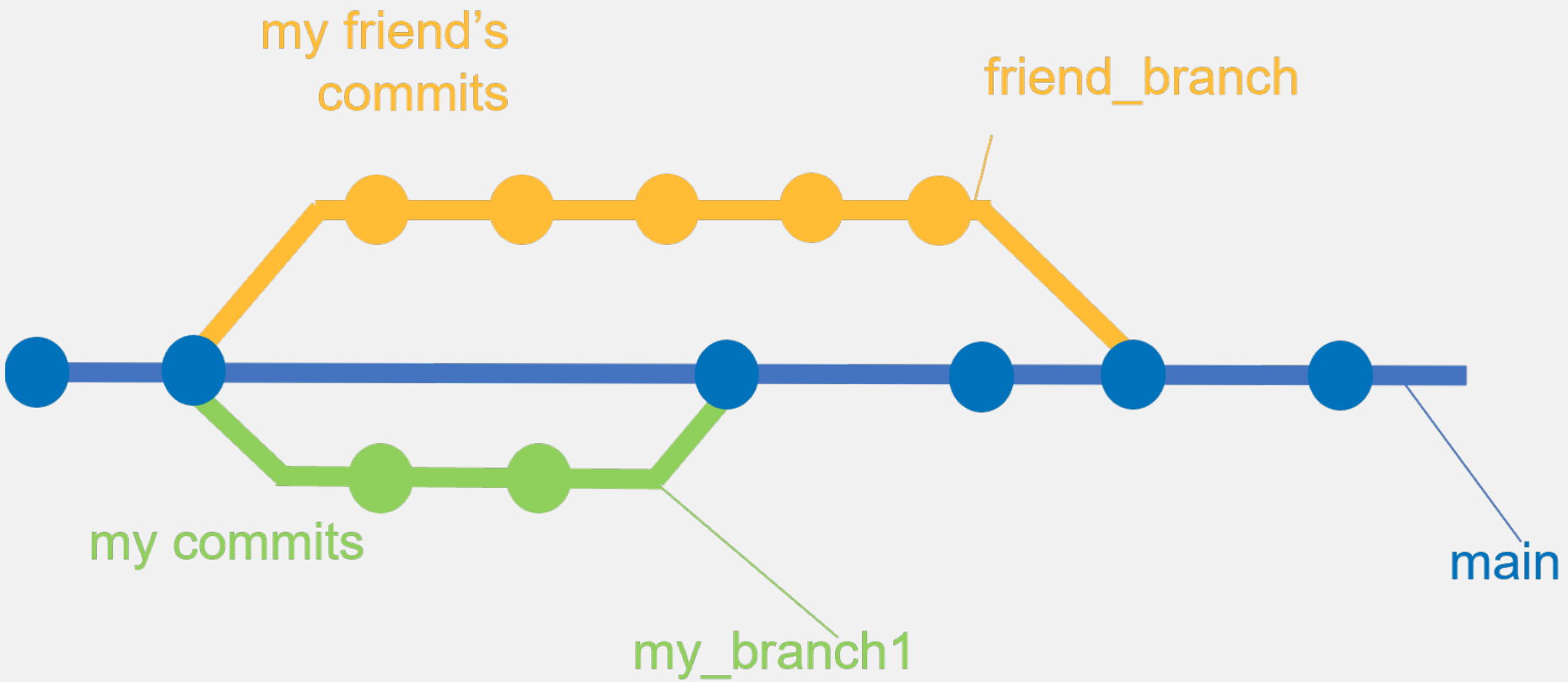
A more complex collaboration workflow

- You can also have multiple branches of the same project



A more complex collaboration workflow

- Branches can be merged using `git merge`



Remote repository platforms

The combination of Git and a remote repository platform unlocks a lot of possibilities!

- Advanced workflow features for collaboration and open-source development
  - Issues and pull requests
- Publishing and sharing of projects
  - Easily connect with Zenodo to get a DOI
  - Accepted by many journals
- Additional features
  - Project wikis
  - Project websites

Take home

- Git (+ Github) is very powerful for coding projects
  - Keep track of your changes and go back if you need to
  - Collaborate and share
- Can be confusing in the beginning, but Git GUIs make it intuitive
- Valuable addition to your toolbox that's also relevant outside academia

> Tips for getting started
>
> Start using it for small projects and discover features as you go along.
>
> Don't get frustrated by the complexity - it will get better.
>
> Use a GUI if you don't like the terminal.

Get started

Command line

Follow this Git training for learning the Git concepts in the command line.

. . .

R and R Studio

There is a whole book on using Git with R that explains the setup in detail but also goes into more advanced topics.

Follow this step by step guide to set up Git and a Github connection in R and R Studio

. . .

Github Desktop

There are detailed step by step guides on how to set up Github Desktop and how to work with in the Github Desktop Documentation

Further Reading

Slides: Tools and tips lecture by Selina Baldauf (FU Berlin)

Learn git concepts, not commands: Blogpost that explains really well the concepts of git, also more advanced ones like `rebase` or `cherry-pick`.

How to write good commit messages: Blogpost that explains why good commit messages are important and gives 7 rules for writing them.

Git cheat sheet: Always handy if you don't remember the basic commands

Book on how to use Git wih R