Contents

Some introductory words

Welcome to our Workshop on “Tools and Workflows for Reproducible Research in the Quantitative Social Sciences”!

This document utilizes a system called Jupyter notebooks, which makes it much easier to programmatically create, change, and delete files.

A1: A brief introduction to reproducible research (in the social sciences)

- Concerns about the replicability and reproducibility of scientific research [Goodman *et al.*, 2016]
- You might have heard about the “replication crisis”
- Example of Reinhart and Rogoff (see also Memo to Reinhart and Rogoff: I think it’s best to admit your errors and go on from there; for more examples see https://twitter.com/kirstie_j/status/1360172705933365248)
- Very hands-on example of an incorrect paper by T. K. Moon on “The Expectation-Maximization Algorithm” and an explanation what went wrong by Dennis Ogbe: https://ogbe.net/blog/sloppy_papers.html

Terminology

Terms used in the literature

- Reproducibility
- Replicability
- …
- Repeatability
- Readability
- …
- Reliability
- Robustness
- Generalizability
- …

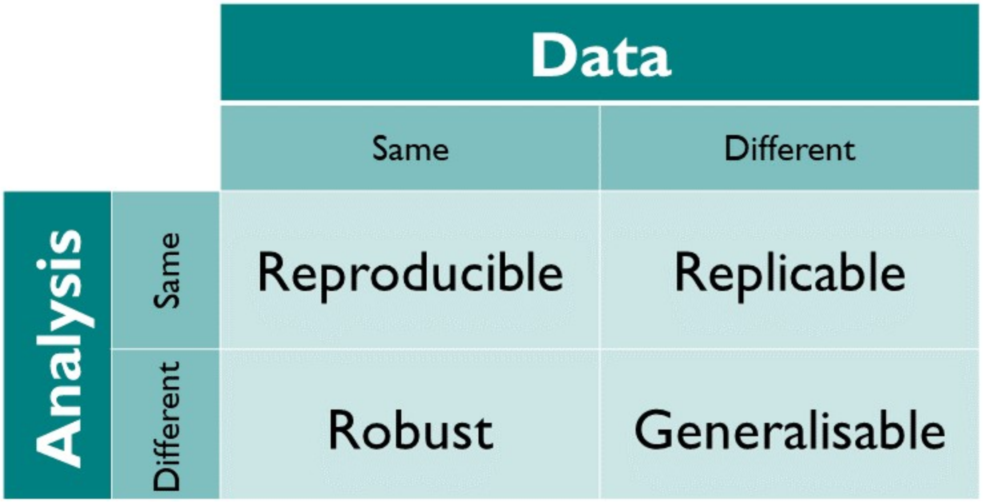
What terms do we use?

- Various papers among various disciplines provide a multitude of (sometimes conflicting) definitions [Freese and Peterson, 2017, Goodman *et al.*, 2016, Peng, 2011]
- Our approach is as follows: Reproducibility \(\neq\) Replicability [Barba, 2018]
- More precisely Stodden *et al.* [2014]:

> **Important**
>
> **Reproducibility** is the calculation of quantitative scientific results by independent scientists using the original datasets and methods

> **Important**
>
> **Replication** is the practice of independently implementing scientific experiments to validate specific findings

The Turing Way’s of defining reproducible research



		Data	
		Same	Different
Analysis	Same	Reproducible	Replicable
	Different	Robust	Generalisable

- "**Reproducible**: A result is reproducible when the same analysis steps performed on the same dataset consistently produces the same answer.
- **Replicable**: A result is replicable when the same analysis performed on different datasets produces qualitatively similar answers.
- **Robust**: A result is robust when the same dataset is subjected to different analysis workflows to answer the same research question (for example one pipeline written in R and another written in Python) and a qualitatively similar or identical answer is produced. Robust results show that the work is not dependent on the specificities of the programming language chosen to perform the analysis.
- **Generalisable**: Combining replicable and robust findings allow us to form generalisable results. Note that running an analysis on a different software implementation and with a different dataset does not provide generalised results. There will be many more steps to know how well the work applies to all the different aspects of the research question. Generalisation is an important step towards understanding that the result is not dependent on a particular dataset nor a particular version of the analysis pipeline."

(Source: https://the-turing-way.netlify.app/reproducible-research/overview/overview-definitions.html)

Reproducibility in more hands-on terms

> "'Reproducibility is just collaboration with people you don't know, including yourself next week' – @philipbstark #dsesummit #openscience"

(Source: https://twitter.com/jakevdp/status/519563939177197571, accessed on 2021-10-27)

Reproducible research is just research done right

> "Reproducible research is a by-product of careful attention to detail throughout the research process and allows researchers to ensure that they can repeat the same analysis multiple times with the same results, at any point in that process. Because of this, researchers who conduct reproducible research are the primary beneficiaries of this practice" [Alston and Rick, 2021].

Opportunities and obstacles of reproducible research

The Turing Way as well as Alston and Rick [2021] provide an overview of possible barriers to reproducibility:

- Limited incentives to give evidence against yourself
- Publication bias towards novel findings
- Not considered for promotion
- Big data and complex computational infrastructure
- Takes time
- Requires additional skills

- Intellectual property rights

Some of the technical aspects will be discussed in section A2: Computer literacy for reproducible research

Why does it matter?

> "An article about computational science in a scientific publication is not the scholarship itself, it is merely advertising of the scholarship. The actual scholarship is the complete software development environment and the complete set of instructions which generated the figures" [Buckheit and Donoho, 1995].

Leeper [2014] distinguishes external and internal reasons

External reasons

- Trust in scientific results is of immense importants, hence, make sure that your research results can be trusted
- Other scientists can build on your workflow/procedures, it helps to accumulate scientific knowledge
- Requirements of journals or funding agencies

https://the-turing-way.netlify.app/reproducible-research/overview/overview-benefit.html

Internal reasons

- You get questions about one of your earlier papers – and are unable to reproduce the analyses (or are even unable to share the data)
- Confidence in your own work
- Easier workflow
- Easier collaboration

Another list of reasons is provided by Alston and Rick [2021], here, the authors distinguish between "Reproducible research benefits those who do it" and "Reproducible research benefits the research community":

Reproducible research benefits those who do it

- It helps researchers remember how and why they performed specific analyses during the course of a project
- It enables researchers to quickly and simply modify analyses and figures
- Reproducible research enables quick reconfiguration of previously conducted research tasks so that new projects that require similar tasks become much simpler and easier
- Conducting reproducible research is a strong indicator to fellow researchers of rigor, trustworthiness, and transparency in scientific research
- Reproducible research increases paper citation rates

Reproducible research benefits the research community

- Reproducible research allows others to learn from your work
- Reproducible research allows others to protect themselves from your mistakes

Finally, The Turing Way also provides a list of reasons why reproducible research might be beneficial: https://the-turing-way.netlify.app/reproducible-research/overview/overview-benefit.html

References

[AR21](1,2,3)
Jesse M. Alston and Jessica A. Rick. A Beginner's Guide to Conducting Reproducible Research. *The Bulletin of the Ecological Society of America*, April 2021. doi:10.1002/bes2.1801.

[Bar18]
Lorena A. Barba. Terminologies for Reproducible Research. *arXiv:1802.03311 [cs]*, February 2018. arXiv:1802.03311.

[BD95]
Jonathan B. Buckheit and David L. Donoho. WaveLab and Reproducible Research. In Anestis Antoniadis, Georges Oppenheim, P. Bickel, P. Diggle, S. Fienberg, K. Krickeberg, I. Olkin, N. Wermuth, and S. Zeger, editors, *Wavelets and Statistics*, volume 103 of Lecture Notes in Statistics, pages 55–81. Springer New York, New York, NY, 1995. doi:10.1007/978-1-4612-2544-7.

[Div18]
Patrick Divine. String Case Styles: camel, Pascal, Snake, and Kebab Case. https://betterprogramming.pub/string-case-styles-camel-pascal-snake-and-kebab-case-981407998841, 2018.

[Fit12]
Michael Fitzgerald. *Introducing Regular Expressions*. O'Reilly, Beijing, Köln, 1. ed edition, 2012. ISBN 978-1-4493-9268-0.

[For17]
Tiago Forte. The PARA Method: a Universal System for Organizing Digital Information. https://fortelabs.co/blog/para/, February 2017.

[FP17]
Jeremy Freese and David Peterson. Replication in Social Science. *Annual Review of Sociology*, 43(1):147–165, July 2017. doi:10.1146/annurev-soc-060116-053450.

[GFI16](1,2)
Steven N. Goodman, Daniele Fanelli, and John P. A. Ioannidis. What does research reproducibility mean? *Science Translational Medicine*, 8(341):341ps12–341ps12, June 2016. doi:10.1126/scitranslmed.aaf5027.

[Hea19]
Kieran Healy. The Plain Person's Guide to Plain Text Social Science. https://plain-text.co/, October 2019.

[Lee14]
Thomas J. Leeper. Reproducible Research:what, Why, and How? 2014.

[Narkebski16]
Jakub Narębski. *Mastering Git: Attain Expert-Level Proficiency with Git for Enhanced Productivity and Efficient Collaboration by Mastering Advanced Distributed Version Control Features*. Packt Publishing, Birmingham, 2016. ISBN 978-1-78355-375-4.

[Pen11]
Roger D. Peng. Reproducible Research in Computational Science. *Science*, 334(6060):1226–1227, February 2011. doi:10.1126/science.1213847.

[PF16]
Stephen R. Piccolo and Michael B. Frampton. Tools and techniques for computational reproducibility. *GigaScience*, 5(1):30, December 2016. doi:10.1186/s13742-016-0135-4.

[SWeiss20]
Ines Schaurer and Bernd Weiß. Investigating selection bias of online surveys on coronavirus-related behavioral outcomes: an example utilizing the GESIS Panel Special Survey on the Coronavirus SARS-CoV-2 Outbreak in Germany. *Survey Research Methods*, 2(14):103–108, June 2020. doi:10.18148/SRM/2020.V14I2.7751.

[SLP14]
Victoria Stodden, Friedrich Leisch, and Roger D. Peng. *Implementing Reproducible Research*. CRC Press, Boca Raton, 2014.

A2: Computer literacy for reproducible research

Computer literacy live demo

(Icons made by Freepik from www.flaticon.com)

Overview

Computer literacy as a barrier to reproducibility

- There exists multiple barriers to reproducibility
- In "The Turing Way", some of these Barriers to reproducibility are introduced
- Among them:
    - Big data and complex computational infrastructure
    - Takes time
    - Requires additional skills

Overall objective

- Introduce concepts and software tools
- Provide necessary skills for subsequent modules
- Topics that will be covered:
    - Be lazy, but trustworthy!
    - Organization of a research project on a computer (file/folder structure, naming conventions, etc.)
    - Working with files and folders (file system, absolute and relative paths, working directory)
    - Using a command-line interface (CLI)
    - The beauty of plain text

Disclaimer

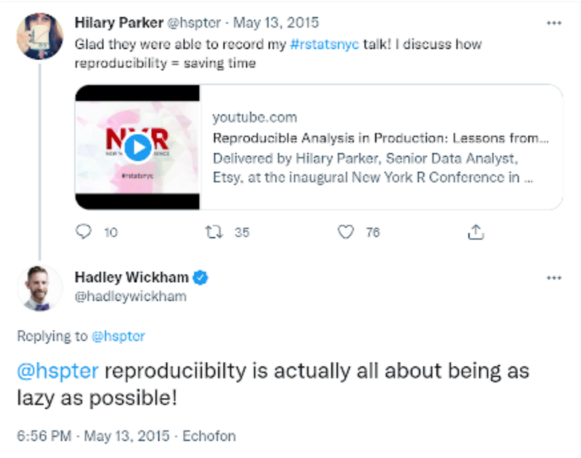
- Targets MS Windows user
- Your mileage may vary…

Two guiding principles of reproducibility

> Be trustworthy!

> By lazy!

Be lazy!



Hilary Parker @hspter · May 13, 2015

Glad they were able to record my #rstatsnyc talk! I discuss how reproducibility = saving time

youtube.com
Reproducible Analysis in Production: Lessons from…
Delivered by Hilary Parker, Senior Data Analyst, Etsy, at the inaugural New York R Conference in …

Hadley Wickham
@hadleywickham

Replying to @hspter

@hspter reproduciibilty is actually all about being as lazy as possible!

6:56 PM · May 13, 2015 · Echofon

(Source: https://twitter.com/hadleywickham/status/598532170160873472)

Some derived principles of being a trustworthy researcher

> Be trustworthy!

1. Document everything and be transparent in your research!
2. Share all materials (if possible)!
3. Ideally, also document and share workflows (→ automation)
4. ….

Some derived principles of being lazy

> "Be lazy!"

1. Be consistent!
    1. Create your own guidelines
    2. Reduce cognitive burden
    3. Reduce decision fatigue
    4. …
2. Automate as much as possible!
3. …

Organization of a research project

Research projects on your computer

- What is a "project"?

> "A series of tasks linked to a goal, with a deadline" [Forte, 2017]

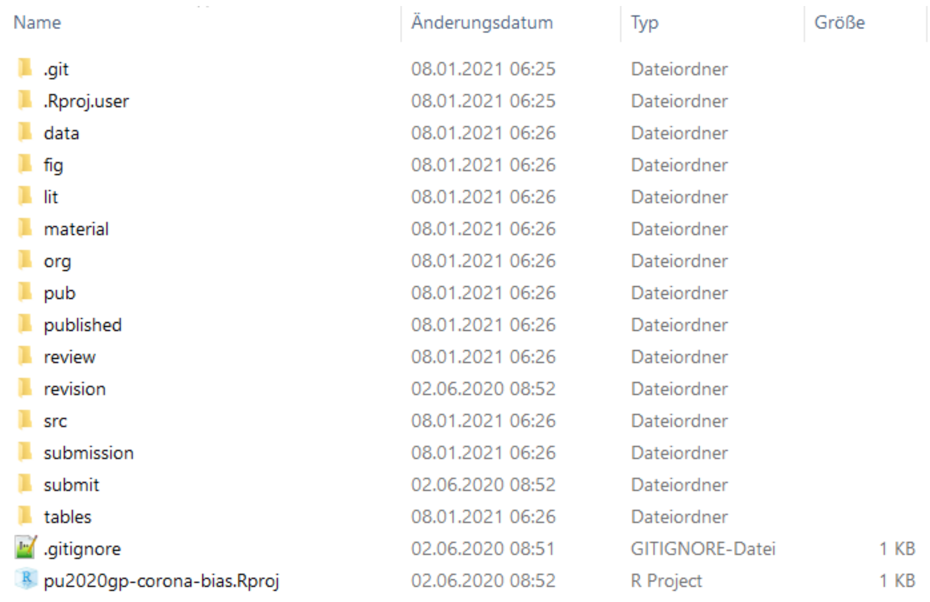
- A research project can be writing a paper, giving a conference presentation or a lecture, …
- Ideally, all information, materials, data, …, should be located in a distinct project folder
- That is, on your computer, everything you need to complete the project should be stored inside *one* folder (which, of course, can/should have multiple subfolders) – "a project should be self-sufficient"

A project's folder structure

- Be consistent!
- A uniform folder structure helps to find items in older projects
- Choose a structure and naming convention that fit your working style

- Generally, folder and file names should consist of the letters **a-z**, the number **0-9**, underscores, **and should never include a blank space**
- I also use hyphens "`-`" sometimes, e.g., `mail_weiss-bernd.pdf`, then, the underscore is used to separate semantically different parts (e.g., `ps2021-03_gesis-panel-meet-the-expert`, `ps2020-11_utrecht_bigsurv20`)
- Furthermore, it is advisable to never use capital letters, as this will increase the likelihood of being inconsistent
- Folder and file names should be as short as possible, but as long as necessary to bring across the gist of its function/content

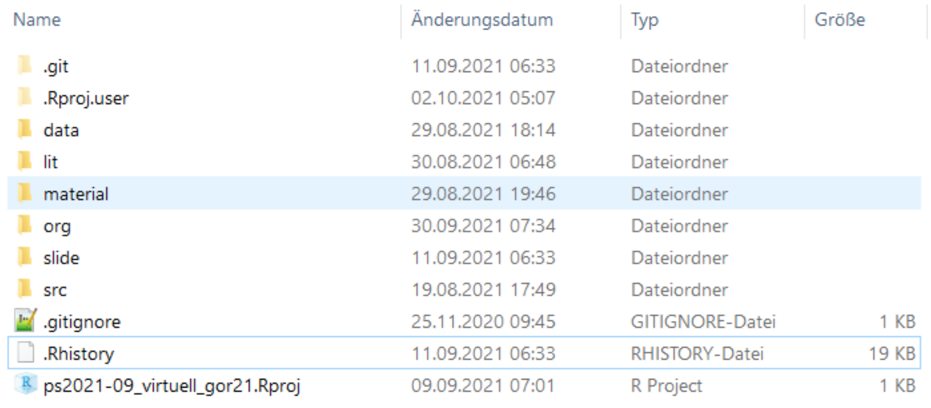
Folder structure of a paper (not perfect)



Name	Änderungsdatum	Typ	Größe
.git	08.01.2021 06:25	Dateiordner	
.Rproj.user	08.01.2021 06:25	Dateiordner	
data	08.01.2021 06:26	Dateiordner	
fig	08.01.2021 06:26	Dateiordner	
lit	08.01.2021 06:26	Dateiordner	
material	08.01.2021 06:26	Dateiordner	
org	08.01.2021 06:26	Dateiordner	
pub	08.01.2021 06:26	Dateiordner	
published	08.01.2021 06:26	Dateiordner	
review	08.01.2021 06:26	Dateiordner	
revision	02.06.2020 08:52	Dateiordner	
src	08.01.2021 06:26	Dateiordner	
submission	08.01.2021 06:26	Dateiordner	
submit	02.06.2020 08:52	Dateiordner	
tables	08.01.2021 06:26	Dateiordner	
.gitignore	02.06.2020 08:51	GITIGNORE-Datei	1 KB
pu2020gp-corona-bias.Rproj	02.06.2020 08:52	R Project	1 KB

This is related to [Schaurer and Weiß, 2020]

Folder structure of a conference presentation (not perfect)



Name	Änderungsdatum	Typ	Größe
.git	11.09.2021 06:33	Dateiordner	
.Rproj.user	02.10.2021 05:07	Dateiordner	
data	29.08.2021 18:14	Dateiordner	
lit	30.08.2021 06:48	Dateiordner	
material	29.08.2021 19:46	Dateiordner	
org	30.09.2021 07:34	Dateiordner	
slide	11.09.2021 06:33	Dateiordner	
src	19.08.2021 17:49	Dateiordner	
.gitignore	25.11.2020 09:45	GITIGNORE-Datei	1 KB
.Rhistory	11.09.2021 06:33	RHISTORY-Datei	19 KB
ps2021-09_virtuell_gor21.Rproj	09.09.2021 07:01	R Project	1 KB

This is related to a GOR presentation on "Investigating self-selection bias of online surveys on COVID-19 pandemic-related outcomes and health characteristics"

Data Carpentry suggests the following project layout

The following guidelines are copyed from the Data Carpentry website on "Introduction to automation" and aim at specific workflow utilizing R, Rmarkdown, and knitr:

- `data`: the original raw data, you shouldn't edit or otherwise alter any of the files in this folder. DATA ARE READ ONLY. If they are encoded in a supported file format, they'll automatically be loaded when you call load.project().
- `cache`: Here is where you will store any data sets that (a) are generated during a preprocessing step and (b) don't need to be regenerated every single time you analyze your data. You can use the cache() function to store data to this directory automatically. Any data set found in both the cache and data directories will be drawn from cache instead of data based on ProjectTemplate's priority rules. We write them to .csv files (comma eperated values) so they are machine readable and can be easily shared.
- `graphs`: the folder where we can store the figures used in the project In our example, the figures are generated directly during the rendering of the RMardown file for the manuscript, but having the figures as standalone files may facilitate getting feedback from your collaborators, or save time if you just work on tweaking its appearance without having to recompile the full manuscript.
- `src`: our R code (the functions that will generate the intermediate datasets, the analyses, and the figures), it's often easier to keep the prose separated from the code. If you have a lot of code (and/or if the manuscript is long), it's easier to navigate.
- `tests`: the code to test that our functions are behaving properly and that all our data is included in the analysis. There are other directories that you may not need as a newcomer, but these will come on handy as you increase your knowledge and prowess with R.

(Source: https://datacarpentry.org/rr-automation/01-automation/index.html)

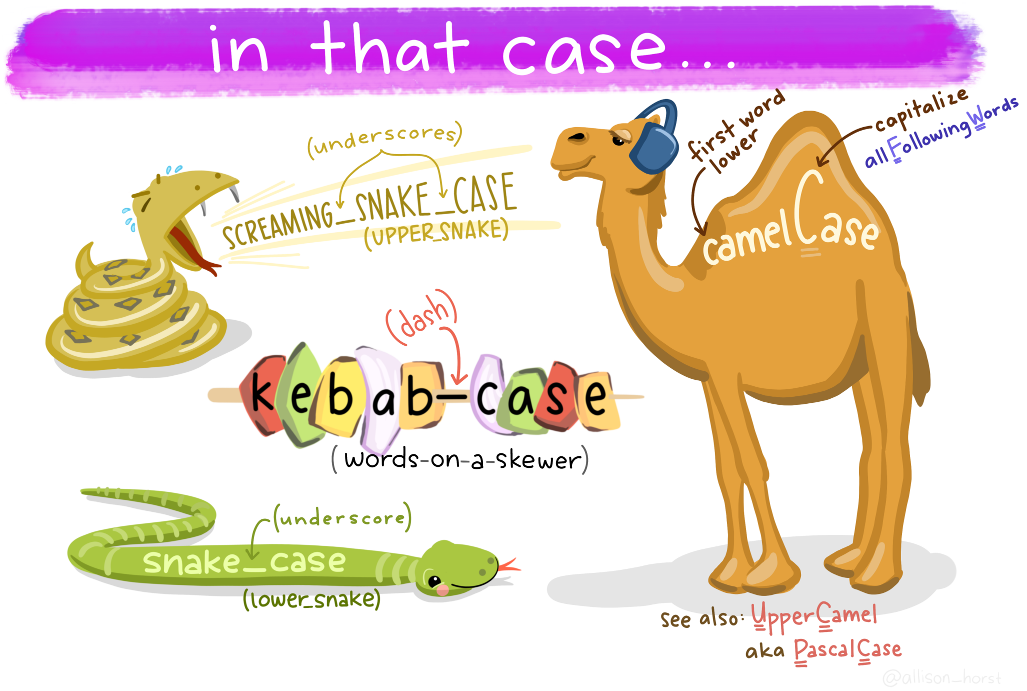
More suggestions to structure a data analysis project

- Structure Your Data Science Projects
- Managing a statistical analysis project – guidelines and best practices
- Creating a research compendium w/ rrtools
- Management of R project
- The Basic Reproducible Workflow Template
- ...

Existing naming conventions (for files/folders, variable names etc.)

- Avoide any whitespace characters
- Some of the following examples are syntactically invalid in some programming languages, e.g., "kebab-case" does not work in R
- Some of the most popular ways to combine words into a single string [Divine, 2018]:
  - CamelCase
  - camelCase
  - **pothole_case/snake_case**
  - kebab-case
  - this.case
  - likethis
  - ...

> Choose one naming convention and stick with it!

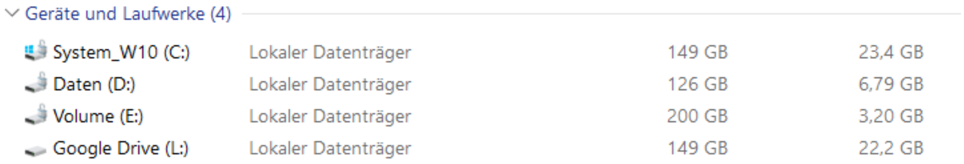


(Source: Artwork by @allison_horst)

Where to store your data?

(the following applies to MS Windows only)

- Quite often, only the `C:` drive is initialized
- Most Windows software suggests to save your files in your "home directory", e.g. `C:\Users\Weiss`
- If, for some reasons, you cannot start your computer and need to reinstall everything, all your data is gone!
- My suggestion: setup at least a second drive (`D:`), which is for your data only.
- Below is my setup, which I have been applying for the last 20 years (the Google Drive is new, though):



Geräte und Laufwerke (4)

System_W10 (C:)	Lokaler Datenträger	149 GB	23,4 GB
Daten (D:)	Lokaler Datenträger	126 GB	6,79 GB
Volume (E:)	Lokaler Datenträger	200 GB	3,20 GB
Google Drive (L:)	Lokaler Datenträger	149 GB	22,2 GB

Backup your data

> Just do it!

Working with files and folders

Why bother?

- When you write code, you often need to read or save data
- For your RMarkdown document, you will store data, figures, tables etc. in different folders, and you need to access this information/material
- So, you need to tell the software where your data can be found
- How do you ensure that your code or your RMarkdown document can be processed on your colleague's computer, your next computer, a remote server?

Examples of a folder/file path

- MS Windows: Folder path to my workshop materials `E:\syncwork\projects\confer\ps2021-10-ws-repro-research`
- Linux (MSYS2): Folder path to my workshop materials `/E/syncwork/projects/confer/ps2021-10-ws-repro-research`
- Full file path specified in Stata to load a dataset (MS Windows): `use "J:\Work\PARI\PARI-F\data\pari-f_data_v0-1.dta"`
- Full file path specified in R to load a dataset (MS Windows): `read.dta("J:/Work/PARI/PARI-F/data/pari-f_data_v0-1.dta")` (confusing? see below…)

(Note, these are all examples of *absolute path* specifications)

What is a path?

> “A path to an entity (in this case, a file, folder, or web page) describes the entity’s unique location within a hierarchical directory or website structure” (https://www.educative.io/edpresso/absolute-vs-relative-path).

> “A path is a string of characters used to uniquely identify a location in a directory structure. It is composed by following the directory tree hierarchy in which components, separated by a delimiting character, represent each directory. The **delimiting character** is most commonly the **slash (“/”)**, the **backslash character (“”)**, or colon (“:”), though some operating systems may use a different delimiter. Paths are used extensively in computer science to represent the directory/file relationships common in modern operating systems, […]. Resources can be represented by either **absolute or relative paths**” (emphasis mine; https://en.wikipedia.org/wiki/Path_(computing))

Delimiting characters

THIS IS IMPORTANT !!$!!!”§!111111

- Delimiting characters `/` or `\` vary by operating system
- MS Windows:
  - Use a colon `:` to specify the drive name (e.g., `c:`, `d:`, `e:`)
  - Folders and files are seperated by a backslash character (`\`)
  - Example: `J:\Work\PARI\PARI-F\data\pari-f_data_v0-1.dta`
- Linux/macOS:
  - No colon (`:`)
  - Use only the slash (`/`) character
  - Example: `/E/syncwork/projects/confer/ps2021-10-ws-repro-research`
  - Another (dynamically created) example showing the (full) path to my Git slides (using the command `readlink`):

```
!readlink -f a3_introduction-to-git.ipynb
```

```
/e/syncwork/projects/confer/ps2021-10-ws-repro-research/src/bw/a3_introduction-to-git.ipynb
```

Delimiting characters in R, \(\LaTeX\), Python, … under MS Windows

- Software, such as R, \(\LaTeX\), Python, with a Linux/UNIX background but can be used in MS Windows as well but behave differently when it comes to specifying file/folder paths
- That is, when specifiying a path (e.g., in R or \(\LaTeX\)) in MS Windows, these programs do not like the backslash character (`\`) (the backslash is used for “escaping” other characters)
- Two solutions:

    - Use the slash character `/`, e.g.:

```
library(haven)
read.dta("J:/Work/PARI/PARI-F/data/pari-f_data_v0-1.dta")
```

    - Escape the backslash character via `\\`, e.g.:

```
library(haven)
read.dta("J:\\Work\\PARI\\PARI-F\\data\\pari-f_data_v0-1.dta")
```

- For more information see, e.g., https://www.dummies.com/programming/r/how-to-work-with-files-and-folders-in-r/
- Note: Many programming languages/statistical packages (R, Python, …) can dynamically create a full path that follows the rules of the respective operating system:
    - R:

```
file.path("J:", "Work", "PARI", "PARI-F", "data", "pari-f_data_v0-1.dta")
[1] "J:/Work/PARI/PARI-F/data/pari-f_data_v0-1.dta"
```

    - Python:

```
import os
os.path.join("e:", "folder1", "folder2", "file")
```

  returns: `'e:folder1\\folder2\\file'` (in Windows)
    - Stata, actually, does not care…
        - `use "J:\Work\PARI\PARI-F\data\pari-f_data_v0-1.dta"`
        - `use "J:/Work/PARI/PARI-F/data/pari-f_data_v0-1.dta"`
- Johannes suggests a handy tool when working on both operating system: Path Copy Copy – Copy file paths from Windows explorer's contextual menu

Absolute and relative paths

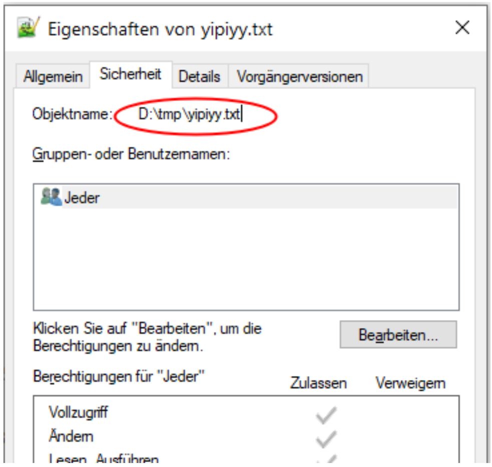
- There exists two types of paths: absolute or relative paths
- Example of an absolute path (in MS Windows): `J:\Work\PARI\PARI-F\data\pari-f_data_v0-1.dta`
- Example of a relative path (in MS Windows): `..\data\pari-f_data_v0-1.dta`

(Source: https://www.earthdatascience.org/courses/intro-to-earth-data-science/python-code-fundamentals/work-with-files-directories-paths-in-python/)

Absolute paths (or full path)

- Contains the entire path to the file or directory that you need to access
- It will begin at you computer's root directory (or respective Window's drive) and will end with the file or directory that you wish to access

- E.g., in MS Windows: `J:\Work\PARI\PARI-F\data\pari-f_data_v0-1.dta` or `D:\tmp\yipiyy.txt`



Objektname: D:\tmp\yipiyy.txt

Gruppen- oder Benutzernamen:

Jeder

Klicken Sie auf "Bearbeiten", um die Berechtigungen zu ändern.

Berechtigungen für "Jeder"

Vollzugriff ✓

Ändern ✓

Lesen, Ausführen ✓

- In Linux or macOS: `/home/weiss/important/yipiyy.txt`
- **Avoid using absolute paths as much as possible since it reduces the portability of your project (your colleague’s computer, someone who wants to reproduce your analyses)**

Working directory and relative paths

- A relative path is relative to a “fixed location” on your computer
- Often, this “fixed location” is the so-called “working directory”
- So, when starting your data analysis, you first define your “working directory” as absolute path – **this is done just once!**
- All other file- or folder-related operations are defined relative to this working directory
- **The huge benefit: when you share your project with a colleague or move it to a new computer, you only have to define the working directory once, everything else should work flawlessly**
- How to define a working directory?
  - R: `setwd("full-path-to-working-directory")` -
  - Stata: `cd "full-path-to-working-directory"` (`cd` = change directory)
  - Python: `import os; os.chdir("full-path-to-working-directory")`
- How to get information about the current working directory?
  - R: `getwd()`

```
> getwd()
[1] "D:/Eigene Dateien/Dokumente"
```

  - Stata: `cd`

```
.cd
C:\Program Files\Stata15
```

  - Python: `import os; os.getcwd()` (cwd = current working directory); see below for an example

```
import os
os.getcwd()
```

```
'E:\\syncwork\\projects\\confer\\ps2021-10-ws-repro-research\\src\\bw'
```

- So, let’s assume the project “PARI-F” is located on drive `J:`, the full absolute path is `J:\Work\PARI\PARI-F`
- The content of the project’s folder `PARI-F` is:

```
.
|-- analysis
|-- data
|-- doc
|-- pari-f.stpr
`-- report
```

- Loading a dataset located in folder `data`, can be accomplished as follows:
    - In R:

```
library(haven); library(ggplot2)
setwd("J:/Work/PARI/PARI-F")          # Define working directory just once.
read.dta("data/pari-f_data_v0-5.dta") # Absolute path: J:/Work/PARI/PARI-F/data/pari-f_data_v0-5.dta
...
ggplot(...)
ggsave("doc/my-fancy-plot.pdf")       # Will be saved in J:/Work/PARI/PARI-F/doc/my-fancy-plot.pdf
```

Note: In R, I would actually suggest to use the package here which avoids manually defining a working directory at all

- In Stata:

```
cd "J:/Work/PARI/PARI-F" // Define working directory just once.
use "data/pari-f_data_v0-5.dta", clear
...
scatter x y
graph export "doc/my-fancy-plot.pdf"
```

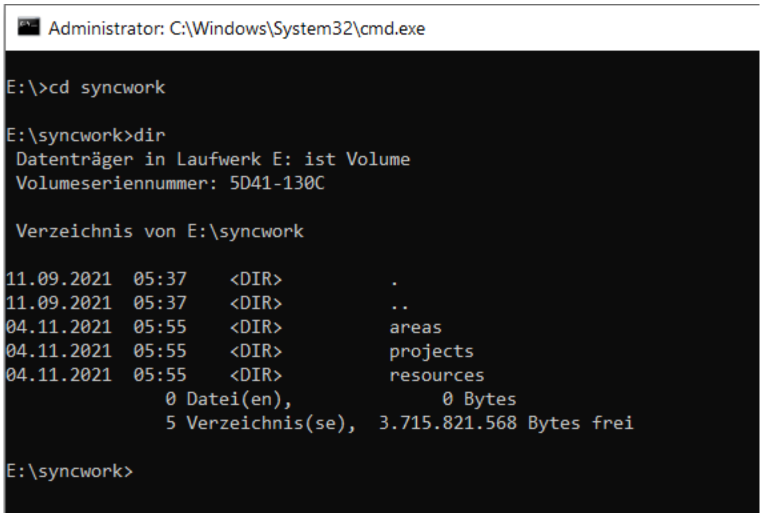
- Shortcuts (`.` and `..`) and the parent directory:
    - The dot `.` denotes the current working directory
    - The dot dot `..` denotes the parent directory, i.e., it points upwards in the folder hierarchy
    - Example in R

```
library(haven); library(ggplot2)
setwd("J:/Work/PARI/PARI-F/data")     # Define working directory just once, now we are in /data.
read.dta("pari-f_data_v0-5.dta")      # Absolute path: J:/Work/PARI/PARI-F/data/pari-f_data_v0-5.dta
...
ggplot(...)
ggsave("../doc/my-fancy-plot.pdf")    # Read:
                                     # - We are in J:/Work/PARI/PARI-F/data
                                     # - The plot should be saved to J:/Work/PARI/PARI-F/doc
                                     # - ".." = go to parent folder of data, i.e., J:/Work/PARI/PARI-F
                                     # - Then, go to folder /doc and store the plot in this folder
```

- Finally, the tilde symbol ~ will bring you back to your home directory, e.g. `cd ~`
- See also https://en.wikipedia.org/wiki/Path_(computing)

Using a comand-line interface (CLI)

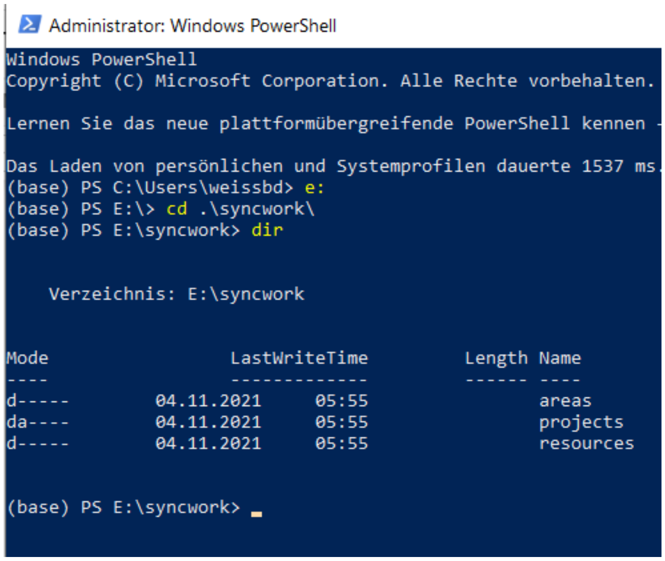
MS Window's default comand-line interpreter (`cmd.exe`)



Administrator: C:\Windows\System32\cmd.exe

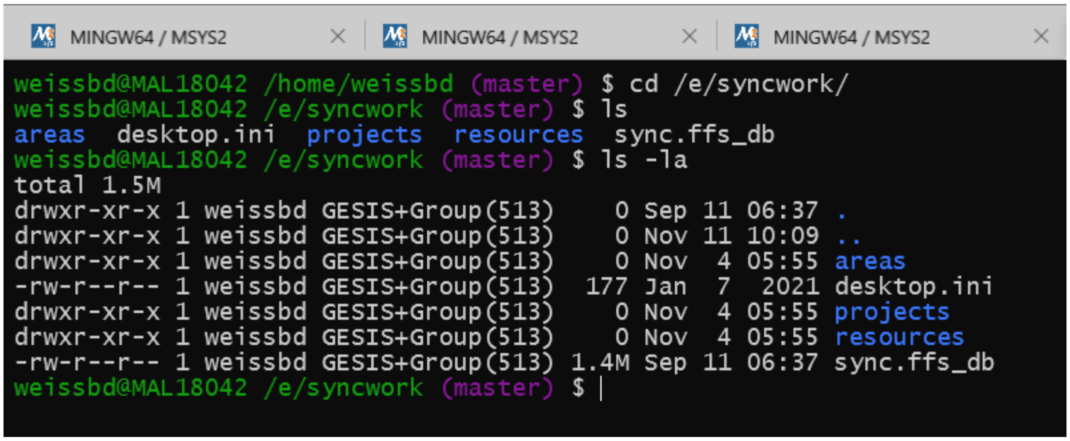
```
E:\>cd syncwork

E:\syncwork>dir
 Datenträger in Laufwerk E: ist Volume
 Volumeseriennummer: 5D41-130C

 Verzeichnis von E:\syncwork

11.09.2021  05:37    <DIR>          .
11.09.2021  05:37    <DIR>          ..
04.11.2021  05:55    <DIR>          areas
04.11.2021  05:55    <DIR>          projects
04.11.2021  05:55    <DIR>          resources
               0 Datei(en),              0 Bytes
               5 Verzeichnis(se),  3.715.821.568 Bytes frei

E:\syncwork>
```

Windows PowerShell



```
Windows PowerShell
Copyright (C) Microsoft Corporation. Alle Rechte vorbehalten.

Lernen Sie das neue plattformübergreifende PowerShell kennen -

Das Laden von persönlichen und Systemprofilen dauerte 1537 ms.
(base) PS C:\Users\weissbd> e:
(base) PS E:\> cd .\syncwork\
(base) PS E:\syncwork> dir


    Verzeichnis: E:\syncwork


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        04.11.2021     05:55                areas
da----        04.11.2021     05:55                projects
d-----        04.11.2021     05:55                resources


(base) PS E:\syncwork> _
```

The Bash shell (Linux, macOS, MS Windows, …)



```
weissbd@MAL18042 /home/weissbd (master) $ cd /e/syncwork/
weissbd@MAL18042 /e/syncwork (master) $ ls
areas  desktop.ini  projects  resources  sync.ffs_db
weissbd@MAL18042 /e/syncwork (master) $ ls -la
total 1.5M
drwxr-xr-x 1 weissbd GESIS+Group(513)    0 Sep 11 06:37 .
drwxr-xr-x 1 weissbd GESIS+Group(513)    0 Nov 11 10:09 ..
drwxr-xr-x 1 weissbd GESIS+Group(513)    0 Nov  4 05:55 areas
-rw-r--r-- 1 weissbd GESIS+Group(513)  177 Jan  7  2021 desktop.ini
drwxr-xr-x 1 weissbd GESIS+Group(513)    0 Nov  4 05:55 projects
drwxr-xr-x 1 weissbd GESIS+Group(513)    0 Nov  4 05:55 resources
-rw-r--r-- 1 weissbd GESIS+Group(513) 1.4M Sep 11 06:37 sync.ffs_db
weissbd@MAL18042 /e/syncwork (master) $ |
```

Don’t always use a sledgehammer for the job

> “When I fire up my Terminal app people that don’t know me well often look on in disgust, no doubt asking themselves what century I was born in. Surely they think, this is 2020! We have touch screens! Beautiful web apps! Amazing software with graphical interfaces! Yup. They’re fun. **But, if your goal is to conduct a reproducible analysis, then you need to step away from those tools and dig into the command line.** Another reaction is, don’t you know about this great R package to do that? I love R! But it’s not always the right tool for the job. Sometimes using R is like using a sledgehammer to pound in a nail or worse, sometimes it’s like using a sledgehammer to pound in a screw. I could, but why? Doing the same thing at the command line with bash commands would be so much easier.” (emphasis mine; https://riffomonas.org/code_club/2020-08-17-command-line-life)

What is a command-line interface (CLI)?

> “A command-line interface (CLI) **processes commands** to a computer program in the form of **lines of text**. The program which handles the interface is called a command-line interpreter or command-line processor. Operating systems implement a command-line interface in a shell for interactive access to operating system functions or services. Such access was primarily provided to users by computer terminals starting in the mid-1960s, and continued to be used throughout the 1970s and 1980s on VAX/VMS, Unix systems and personal computer systems including DOS, CP/M and Apple DOS.”

(emphasis mine; https://en.wikipedia.org/wiki/Command-line_interface)

And what is a shell?

> “In computing, a shell is a computer program which exposes an operating system’s services to a human user or other program. In general, operating system shells use either a command-line interface (CLI) or graphical user interface (GUI), depending on a computer’s role and particular operation. It is named a shell because it is the outermost layer around the operating system.”

(https://en.wikipedia.org/wiki/Shell_(computing))

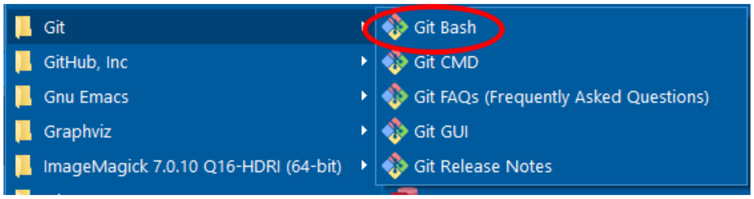
Okay, but why?!

- Fast and effcient way to interact with your computer
- Important part of your automation toolbox to create a reproducible data analysis pipeline
- “1-Click Reproducibility” or better (but less catchy): “1-Command-on-a-CLI Reproducibility”
- Many functions that RStudio provides are actually based on command-line tools (Git, pandoc, \(\LaTeX\), …) – it might be helpful to understand what is happening under the hood
- Accessing a remote server almost always requires some sort of command line skills
- …

[Piccolo and Frampton, 2016]

I also recommend “Top ten reasons to learn to use the command line: Expanding your reproducibility tools”, see https://riffomonas.org/code_club/2020-08-17-command-line-life

The Git Bash


Git
GitHub, Inc
Gnu Emacs
Graphviz
ImageMagick 7.0.10 Q16-HDRI (64-bit)
Git Bash
Git CMD
Git FAQs (Frequently Asked Questions)
Git GUI
Git Release Notes

Important commands

- Note, some of these commands depend on the respective operating system
- Almost all of these command can have several arguments, e.g., `ls -la`
- Linux cheatsheet: https://files.fosswire.com/2007/08/fwunixref.pdf
- MS Windows cheatsheet: http://www.cs.columbia.edu/~sedwards/classes/2015/1102-fall/Command Prompt Cheatsheet.pdf

Description	Win	Linux, macOS
Copy files, folders	`copy`	`cp`
Move files, folders	`move`	`mv`
List folder content	`dir`	`ls`
Create new folder	`mkdir`	`mkdir`
Change current folder	`cd`	`cd`
Show current path	`echo %cd%`	`pwd`
Danger zone No undo!		
Delete file(s)	`del`	`rm`
Delete folder(s)	`rmdir`	`rm`

A small script to create a project folder

The following small bash/shell (on Windows: use the Git bash) script was introduced in the very beginning of this module. It initializes a new project folder. It even detects when you forget to submit a folder name for your project:

```
#!/bin/sh

# Yes, I had to google that...
# https://stackoverflow.com/questions/6482377/check-existence-of-input-argument-in-a-bash-shell-script
if [ $# -eq 0 ]
  then
    echo "No arguments supplied! You stupid!!!!!!"
    exit 1
fi

mkdir $1
cd $1
mkdir data
mkdir src
mkdir output
touch .gitignore
touch README
```

Exercise

- Start the Git Bash
- List the content of the current folder via `ls -la`
- Determine where you are via `pwd`
- Go to the parent folder via `cd <your input>`
- Copy the content from my small script (see above, section A small script to create a project folder, use the little copy-icon in the top-right corner) and past it into a text file, save the text file as `create-project.sh`
- Execute the script `./create-project.sh` (the `./` is important, for an explanation see here) without a folder name
- Execute the script `./create-project.sh <your-new-project>` with a folder name as argument

The power of plain text

Again, it is 2021, why would I write in plain text?

- Version control (e.g., Git) works well (efficient, differences between versions are easy to understand) with plain text documents, `.docx` et al. not so much

- Future proof
- Can easily manipulated with external tools
- ...

Canonical reference: The Plain Person’s Guide to Plain Text Social Science by Kieran Healy

Encoding

- Puh, what is (character) encoding?
- See also https://www.w3.org/International/questions/qa-what-is-encoding.en
- My only rule: Use utf-8
- If you run into trouble, use a search engine your trust... which will most likely lead to a Stackoverflow page
- Example: How to keep umlaut in R?
- Notepad++ is my my tool of choice...
- On the command line (Linux/Bash), you can use the command `file`, see below for the encoding of my references file:

```
!file references.bib
```

```
references.bib: Unicode text, UTF-8 text, with very long lines (1869)
```

The power of regular expressions

- “Regular expressions are specially encoded text strings used as patterns for matching sets of strings” [Fitzgerald, 2012]
- Regex = Regular expressions
- Regex can be really handy when it comes to:
  - Searching for highly standardized strings, e.g., Emails, URLs etc.
  - Search and replace in an automated fashion
  - Extract text
  - ...
- Regex are implemented in almost all programming languages, including Python, R, Stata

The following example utilizes Python for a brief demonstration of Regex. It uses the following sample text:

```
lalaal@gesis.org ysdfvysdfc sdcfasefasdf hhatz@gesis.org,
aselkjcf nöasjfasö djfcapsodfak http://www.gesis.org/welt
acsd1132 33.45 sklfdgssfd ef caois 12.1 willlst@gesis.org
http://www.gesis.org/helloworld
```

```
import re

sample = """lalaal@gesis.org ysdfvysdfc sdcfasefasdf hhatz@gesis.org,
aselkjcf nöasjfasö djfcapsodfak http://www.gesis.org/welt
acsd1132 33.45 sklfdgssfd ef caois 12.1 willlst@gesis.org
http://www.gesis.org/helloworld
"""
```

```
print(sample)
```

```
lalaal@gesis.org ysdfvysdfc sdcfasefasdf hhatz@gesis.org,
aselkjcf nöasjfasö djfcapsodfak http://www.gesis.org/welt
acsd1132 33.45 sklfdgssfd ef caois 12.1 willlst@gesis.org
http://www.gesis.org/helloworld
```

Extract all emails:

```
m = re.findall(r'[\w.+-]+@[\w-]+\.[\w.-]+', sample)
m
```

```
['lalaal@gesis.org', 'hhatz@gesis.org', 'willlst@gesis.org']
```

Extract all numbers:

```
m = re.findall(r'\d+', sample)
m
```

```
['1132', '33', '45', '12', '1']
```

Automation and productivity

- There is a trade-off between automation and productivity
- Automation, learning the required skills, setting up your tool chain will take a lot of time – but can also safe a lot of time
- Find the sweet spot where you invest some time now to save – in the long run – more time and increase your productivity
- “Find the sweet spot” – well, easier said than done

Zhuowei Zhang
@zhuowei

Never spend 6 minutes doing something by hand when you can spend 6 hours failing to automate it

6:28 AM · Apr 26, 2020

References

[AR21]
Jesse M. Alston and Jessica A. Rick. A Beginner's Guide to Conducting Reproducible Research. *The Bulletin of the Ecological Society of America*, April 2021. doi:10.1002/bes2.1801.

[Bar18]
Lorena A. Barba. Terminologies for Reproducible Research. *arXiv:1802.03311 [cs]*, February 2018. arXiv:1802.03311.

[BD95]
Jonathan B. Buckheit and David L. Donoho. WaveLab and Reproducible Research. In Anestis Antoniadis, Georges Oppenheim, P. Bickel, P. Diggle, S. Fienberg, K. Krickeberg, I. Olkin, N. Wermuth, and S. Zeger, editors, *Wavelets and Statistics*, volume 103 of Lecture Notes in Statistics, pages 55–81. Springer New York, New York, NY, 1995. doi:10.1007/978-1-4612-2544-7.

[Div18]
Patrick Divine. String Case Styles: camel, Pascal, Snake, and Kebab Case. https://betterprogramming.pub/string-case-styles-camel-pascal-snake-and-kebab-case-981407998841, 2018.

[Fit12]
Michael Fitzgerald. *Introducing Regular Expressions*. O'Reilly, Beijing, Köln, 1. ed edition, 2012. ISBN 978-1-4493-9268-0.

[For17]
Tiago Forte. The PARA Method: a Universal System for Organizing Digital Information. https://fortelabs.co/blog/para/, February 2017.

[FP17]
Jeremy Freese and David Peterson. Replication in Social Science. *Annual Review of Sociology*, 43(1):147–165, July 2017. doi:10.1146/annurev-soc-060116-053450.

[GFI16]
Steven N. Goodman, Daniele Fanelli, and John P. A. Ioannidis. What does research reproducibility mean? *Science Translational Medicine*, 8(341):341ps12–341ps12, June 2016. doi:10.1126/scitranslmed.aaf5027.

[Hea19]
Kieran Healy. The Plain Person's Guide to Plain Text Social Science. https://plain-text.co/, October 2019.

[Lee14]
Thomas J. Leeper. Reproducible Research:what, Why, and How? 2014.

[Narkebski16]
Jakub Narębski. *Mastering Git: Attain Expert-Level Proficiency with Git for Enhanced Productivity and Efficient Collaboration by Mastering Advanced Distributed Version Control Features*. Packt Publishing, Birmingham, 2016. ISBN 978-1-78355-375-4.

[Pen11]
Roger D. Peng. Reproducible Research in Computational Science. *Science*, 334(6060):1226–1227, February 2011. doi:10.1126/science.1213847.

[PF16]
Stephen R. Piccolo and Michael B. Frampton. Tools and techniques for computational reproducibility. *GigaScience*, 5(1):30, December 2016. doi:10.1186/s13742-016-0135-4.

[SWeiss20]
Ines Schaurer and Bernd Weiß. Investigating selection bias of online surveys on coronavirus-related behavioral outcomes: an example utilizing the GESIS Panel Special Survey on the Coronavirus SARS-CoV-2 Outbreak in Germany. *Survey Research Methods*, 2(14):103–108, June 2020. doi:10.18148/SRM/2020.V14I2.7751.

[SLP14]
Victoria Stodden, Friedrich Leisch, and Roger D. Peng. *Implementing Reproducible Research*. CRC Press, Boca Raton, 2014.

A3: An introduction to Git

Bernd Weiß

THIS IS GIT. IT TRACKS COLLABORATIVE WORK ON PROJECTS THROUGH A BEAUTIFUL DISTRIBUTED GRAPH THEORY TREE MODEL.
COOL. HOW DO WE USE IT?
NO IDEA. JUST MEMORIZE THESE SHELL COMMANDS AND TYPE THEM TO SYNC UP. IF YOU GET ERRORS, SAVE YOUR WORK ELSEWHERE, DELETE THE PROJECT, AND DOWNLOAD A FRESH COPY.

(Source: xkcd, https://xkcd.com/1597/, accessed on 2017-12-23)

Git live demo

(Icons made by Freepik from www.flaticon.com)

Preface

A few words on notation

All interactive Git commands in this document start with an exclamation mark like so: `!git add` In your terminal, though, you have to type in `git add` without exclamation mark

Initializing folders and files

Since we'd like to have a fully reproducible example, everytime this notebook is started everything is created from scratch. First, a test folder for Git is being created.

Second, change into git's test directory and create a new file with a total of four lines of text.

So, let's print the content of the test file:

```
f = open("test.txt", "r")
print(f.read())
f.close()
```

```
l1: Branch: master
l2: Author: BW
l3: // Always start with a dumb comment:
l4: use my_fancy_data.dta, clear
l5: gen v1 = 1
l6: gen v2 = v1 + 1
l7: mean v1
```

I also will create a couple of handy python functions that will make my life much easier:

The Concept of Version Control

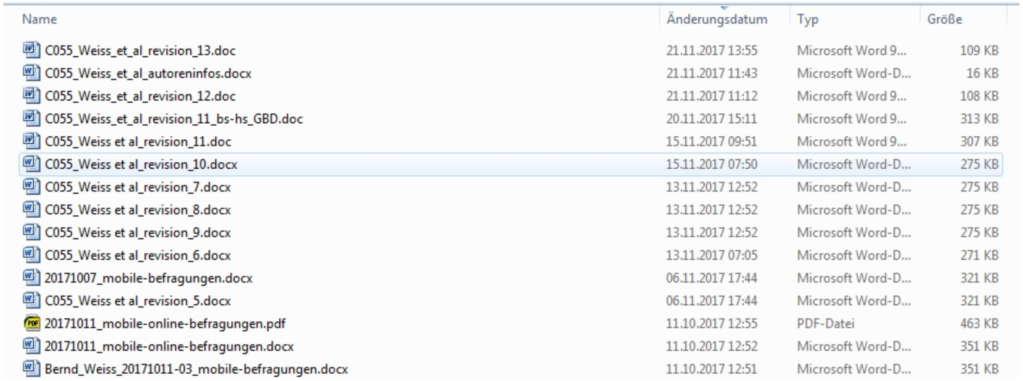
Prerequisits

- Git and other tools have been developed in the context of software development (in the Linux community, to be more precise)
- Even though there exits graphical user interfaces (GUIs) for working with Git, it is highly recommended that you have a basic knowledge of how to use the CLI (see section on Using a comand-line interface (CLI))
- Once you mastered using Git at the command line, using a GUI is a peace of cake

Since most of you are working on a MS Windows PC, it is also a good idea to know the meaning of the `PATH` variable (what it is used for, how to access it and how to modify it).

Why use a Version Control System (VCS)?

- For backup
- For collaborative work (including running into conflicting versions of a file) und syncing
    - There is always a (the) "most recent" version of a file
    - Given that there are conflicting versions of (text) files, Git is able to clearly identify these conflicts by displaying the differences of conflicting files (given they are in plain text)
- Keeping track of changes (aka, time travel; all changes are tracked and it is quite easy to go back in time). So, even if you invented Skynet (popcultural reference) and mankind is about to being terminated for good, you always can go back in time
- And, avoiding the horror of `final_rev2_update12_after-computer-crashed.docx` (see http://phdcomics.com/comics.php?f=1531)



Name	Änderungsdatum	Typ	Größe
C055_Weiss_et_al_revision_13.doc	21.11.2017 13:55	Microsoft Word 9...	109 KB
C055_Weiss_et_al_autoreninfos.docx	21.11.2017 11:43	Microsoft Word-D...	16 KB
C055_Weiss_et_al_revision_12.doc	21.11.2017 11:12	Microsoft Word 9...	108 KB
C055_Weiss_et_al_revision_11_bs-hs_GBD.doc	20.11.2017 15:11	Microsoft Word 9...	313 KB
C055_Weiss et al_revision_11.doc	15.11.2017 09:51	Microsoft Word 9...	307 KB
C055_Weiss et al_revision_10.docx	15.11.2017 07:50	Microsoft Word-D...	275 KB
C055_Weiss et al_revision_7.docx	13.11.2017 12:52	Microsoft Word-D...	275 KB
C055_Weiss et al_revision_8.docx	13.11.2017 12:52	Microsoft Word-D...	275 KB
C055_Weiss et al_revision_9.docx	13.11.2017 12:52	Microsoft Word-D...	275 KB
C055_Weiss et al_revision_6.docx	13.11.2017 07:05	Microsoft Word-D...	271 KB
20171007_mobile-befragungen.docx	06.11.2017 17:44	Microsoft Word-D...	321 KB
C055_Weiss et al_revision_5.docx	06.11.2017 17:44	Microsoft Word-D...	321 KB
20171011_mobile-online-befragungen.pdf	11.10.2017 12:55	PDF-Datei	463 KB
20171011_mobile-online-befragungen.docx	11.10.2017 12:52	Microsoft Word-D...	351 KB
Bernd_Weiss_20171011-03_mobile-befragungen.docx	11.10.2017 12:51	Microsoft Word-D...	351 KB

- For having the possibility to test new code/functionalities in a "sandbox" (aka a new branch) (following-up my Skynet reference: create a parallel universe (the branch), do your evil thing and then go back to your reality if you don't like it, i.e., delete the parallel universe/branch)
- While creating a history of changes, you are supposed to provide proper and meaningful messages that describe what has changed. If you do this thouroghly, you have a nice log file of all changes (like a lab notebook)
- Authorship attribution
- Modern web interfaces such as GitHub also allow for social interaction (strangers can send you pull requests to improve your code/document/…)

For what type of files is a VCS useful?

- Most useful for text files (Stata do files, SPSS syntax files, R skripts etc.). Text files can stored very efficient since only changes between version are tracked
- Binary files (Blob = binary large object) (images, word files, stata data files, etc.) can be stored in a VCS but less efficient than text files since every time the entire file is saved

Terminology and concepts

- There is Git.
- And, then there are GitHub, GitLab etc., which are (web-based GUI) frontends to make working with Git easier, especially when it comes to collaborative work.
- GitHub and GitLab provide the opportunity to setup a remote Git repository.
- So, in most cases you will need a local installation of Git.

- In addition to being a frontend to Git, GitHub and GitLab also provide project management features and allow to create so-called issues that can be considered file-related todo items. You also can use milestone etc.

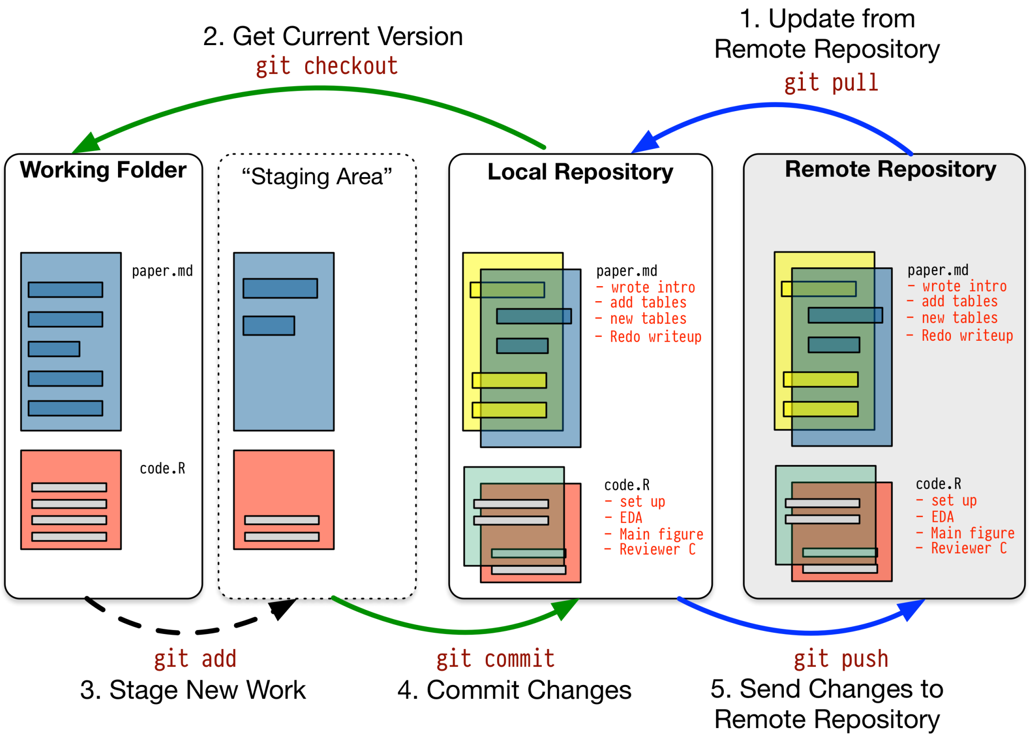
- One more thing, "Git" is the name of the software, and the actual command-line tool is `git` (e.g., in Windows it is `git.exe`).

Git: A 30,000 foot view

- Git is a version control system (VCS). As mentioned above, a VCS allows you to track the history and attribution of your project files over time in a repository [Narębski, 2016]
- It is, if you will, a (very, very) powerful undo function (well, kind of…)
- To be more precise, Git is a distributed VCS (DCVS) and hence a tool for collaborartive work
- If you want to utilize Git for collaborative work, one approach of using Git in this context assumes that there exists a central and remote repository. Most fomous are GitHub, at GESIS we use GitLab

- Workflow in Git (given that a Git repository has already be initialized):

1. Work locally (i.e. on your computer) on your files until a certain feature is completed (a function is completed, a paragraph written etc).
2. *Commit* your file and write a commit message, i.e. inform git that a certain file (or more) have changed and inform your future self (or someone else) about the nature of your changes (aka write a commit message). This has to be done manually.
3. Commit early, commit often!
4. When a remote repository exists: send ("push") you changes to the remote repository.



Source: [Healy, 2019], see https://plain-text.co/keep-a-record.html

How I use Git

- Git is a very powerful tool, in my own work, I utilize a rather limited set of its capabilities
- This is what I mostly do with Git:
  - Initialize a new Git repository or clone an existing respository
  - Backup my work on a remote server
  - Track changes
  - Use branches to implement experimental features
  - Search (and undo) previous changes (most of the time using the interface provided bv GitHub or GitLab)
- "Google" (or whatever your prefered seach engine is) a lot …

Installing Git and setup

Download and installation

Git (for Windows) can be downloaded from: https://git-scm.com/download/win.

Here are a few questions that you will be asked during the installation:

- Default editor (use Notepad++ if you have it on your computed, vim also works)
- Adjusting your PATH environment (you might want to go with the second optionn "Use Git from the Windows command Prompt")
- Choosing HTTPS… (go with default: OpenSSL)

In case you will be working with others, you also will need a remote repository (be able to acces a remote respository). For convenience reasons it is recommended that you also install/set up SSH (see next slides).

For various reasons, I no longer use a standalone version of Git but use a version of Git that can be installed via MSYS2. "MSYS2 is a collection of tools and libraries providing you with an easy-to-use environment for building, installing and running native Windows software" (https://www.msys2.org/).

Prepare to access remote repositories

Well, not so distributed at all…

- Even though Git is called "distributed", most of the time, there is just one central server (e.,g., GitHub, GitLab, …)
- A Git project is stored in a repository, which can be local or remote
- When using Git to access a remote repository (for backup or collaborative work) on a remote server, you need to authenticate yourself to the server
- There are two ways of authentication: HTTPS or SSH
- Despite its technical details, I always choose SSH but Johannes, for instance, prefers HTTPS
- More information can be found on these websites:
  - https://docs.github.com/en/get-started/getting-started-with-git/about-remote-repositories
  - https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/about-authentication-to-github#authenticating-with-the-command-line

Setting up SSH

To work with git on you local computer, you do not need SSH (= Secure Shell). SSH is a network protocol that comes in handy, when you work with remote repositories and when you do not want to type-in your passwort every time you pull (fetch) or push (send) from a remote repository. You still need to authenticate yourself, though.

Authentication in SSH (which is also the name of the program) works by using a private and a public key (usually the public key has the file extension `.pub`, e.g. my public key is `id_rsa.pub`). When you start working with SSH for the very first time, you have to create both keys.

The private key remains on your local computer and you have to make sure that it is safe – it is a simple text file and it is your password now, and everyone who has your private key can access your files. Again, everyone who has your private key has your password!

This is what my *public key* looks like:

```
ssh-rsa
AAAAB3NzaC1yc2EAAAABIwAAAQEAyOQ9RT6TkfgkdO2NspzdVJE5CZ03yYAhVwLGo
CrI3E9/Ix0MAySunXExjhsQi2XkhPBjLOEahYuuLaAWHuBc7apUPRNSBy+mdUHnH3
0BdTQijQ6vj3RL99HO4yrZnipIlkS5ufw/+hpbXXOzSOqTvyGtL9ygm3eA2HDSQtz
2ptFq8anODJDKrgTbNLb/YZ9KDIcpdO/Sfk4LtvaGF3tIFlyE+pogNmN4eWiYg9Xv
25BhVVxWMHadRFLeDastWO4SedriEHzQYaNgxVNTufqolJ0nbg4R//fVDxjR2SbzV
AHLZ+eVPUx+vzcPVMP9wYPcnii9YLiSRy+hlUAOR/kXeQ== berndweiss
```

> **Important**
>
> The *public key* (not the private key!) has to be stored at the GitHub/GitLab/… website. Now, everyone who has your public key can encrypt files (that are sent to you via the internet) but only you (or anyone else who has your private key) can decrypt the files. And, for that reasons you do not have to login everytime you push/pull files from the remote repository.

- How to setup SSH on you computer is explained on this website: https://docs.gitlab.com/ce/ssh/README.html ("Generate an SSH key pair")
- The most important point is that `ssh` is able to find your key pair, i.e., it needs to be located in your HOME folder

If everything works well, you should receive the following friendly welcome message after typing in the command `ssh -T git@git.gesis.org`:

> Welcome to GitLab, [your username]

Basic workflow

Setting up a Git repository

Usually, there are two ways to set up/obtain a Git repository:

1. You create a new Git repository, push it to GitHub/GitLab and start collaborating with your colleagues (or only yourself)
2. You "clone" an existing repository from a remote Git server such as GitHub/GitLab (for more details see section Working with remote repositories)

Creating a local Git repository

The first step is to create a Git repository. After the repository has been created, we need to tell `git` which files will be subject to version control. So, the following git commands will be utilized:

- `git init`: Creates a new folder `.git`, which contains configuration files and the repository. As of now, `git` does not know anything about our file(s), e.g. `test.txt`
- From now on, all examples will refer to a demo repository called `git_test_folder`

The content of `git_test_folder` is

```
!ls -la
```

```
total 9
drwxr-xr-x 1 weissbd DomÃ¤nen-Benutzer   0 Nov 17 05:21 .
drwxr-xr-x 1 weissbd DomÃ¤nen-Benutzer   0 Nov 17 05:21 ..
-rw-r--r-- 1 weissbd DomÃ¤nen-Benutzer 160 Nov 17 05:21 test.txt
```

And, `test.txt` contains the following content:

```
!cat test.txt
```

```
l1: Branch: master
l2: Author: BW
l3: // Always start with a dumb comment:
l4: use my_fancy_data.dta, clear
l5: gen v1 = 1
l6: gen v2 = v1 + 1
l7: mean v1
```

Now, let's initialize the Git repository using the `git init` command.

```
!git init
```

```
Initialized empty Git repository in /e/syncwork/projects/confer/ps2021-10-ws-repro-
research/src/bw/git_test_folder/.git/
```

The status of a repository (`git status`)

Before we start doing anything, let's check the status of our newly created repository using the command `git status`. It shows the status of the currect working tree, the main one is called `master`. As of now, `git` is not aware of any files yet, so it informs us about the existance of 'Untracked files: ...'.

```
!git status
```

```
On branch main

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        test.txt

nothing added to commit but untracked files present (use "git add" to track)
```

Adding files to a git repository: `git add` and `git commit`

Now it is time for some file action by adding (a) file(s) to our repository.

In the previous section on `git status` it was recommended that `git add` is used to add files to the git repository:

```
(use "git add <file>..." to include in what will be committed)
```

That is what we are going to do now: To actually 'save' (check-in or track) files in the repository, a *two-step* procedure needs to be performed.

The first step is to call `git add`, the second step is to commit the file(s) using `git commit`. For now, it might be hard to see the benefit of this two-step procedure, see http://gitolite.com/uses-of-index.html for a thorough description.

- `git add -A` : Adds (here ´-A´ means "all files") files to the *index* (or staging area).
- To add a particilar files to the index, use `git add my_special_file.do`.

```
!git add -A
```

```
!git status
```

```
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
	new file:   test.txt
```

The second step is to run the command `git commit -m "your text, verbs in imperative form"` (see below), e.g. `git commit -m "add function to compute tau^2"`. Since this is my first commit, I always apply the following commit message: `git commit -m "initial commit`.

According to https://git.kernel.org/pub/scm/git/git.git/tree/Documentation/SubmittingPatches?id=HEAD#n133 commit messages should follow the “imperative-style”:

> “Describe your changes in imperative mood, e.g. “make xyzzy do frotz” instead of “[This patch] makes xyzzy do frotz” or “[I] changed xyzzy to do frotz”, as if you are giving orders to the codebase to change its behavior. Try to make sure your explanation can be understood without external resources. Instead of giving a URL to a mailing list archive, summarize the relevant points of the discussion.”

```
!git commit -m "Initial commit"
```

```
[main (root-commit) 12b9483] Initial commit
 1 file changed, 7 insertions(+)
 create mode 100644 test.txt
```

Again, let’s see what `git status` reports:

```
!git status
```

```
On branch main
nothing to commit, working tree clean
```

So, there are no untracked files, that is, “nothing to commit, working directory clean”.

Viewing Git’s commit history: `git log`

There is another useful command `git log` that informs about `git`’s history, i.e. commited files and folders:

```
!git log
```

```
commit 12b9483953c67438aa087586c82b7ba17d02975d
Author: Bernd Weiss <spam@metaanalyse.de>
Date:   Wed Nov 17 05:21:12 2021 +0100

    Initial commit
```

Right now, the history only contains one entry. The very first line `commit ...` shows the SHA1 hash. The ‘Secure Hash Algorithm 1’ is used to calculate this long, hexadecimal number for a file. Files with identical content are representet by an identical SHA1 hash, files with different content do not share an identical SHA1 hash. Using these SHA1 numbers, `git` can identify changes in a file.

Let’s start changing the content of text.txt. For instance, remove line 3 (`// Always start with a dumb comment:`). First, let’s print the original file content again:

```
print_file("test.txt")
```

```
l1: Branch: master
l2: Author: BW
l3: // Always start with a dumb comment:
l4: use my_fancy_data.dta, clear
l5: gen v1 = 1
l6: gen v2 = v1 + 1
l7: mean v1
```

Here is some Python code that removes the comment line.

```
remove_line("test.txt", 3)
```

Print out the new code file (remember, the comment line has been removed).

```
f = open("test.txt", "r")
print(f.read())
f.close()
```

```
l1: Branch: master
l2: Author: BW
l4: use my_fancy_data.dta, clear
l5: gen v1 = 1
l6: gen v2 = v1 + 1
l7: mean v1
```

Again, let's check `git status` to see what our repository is doing and what has changed...

```
!git status
```

```
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   test.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

Exercise

- Open your Git Bash
- Go to your Home directory via `cd ~` (or, actually, go wherever you want)
- Create a new folder (either via `mkdir` or `create-project.sh`)
- Change into the newly created directory via `<your input here>`
- Initialize your new Git project via `git init`
- Copy a few files (PDF files etc. – does not really matter, but no sensitive material!) in your new project folder
- What comes next? Hint: `git add` and then `git commit <your input>`
- Check the status and the history of your Git repository
- **PLEASE do not delete the repository, we will need it for a later exercise!**

Undo changes / Going back in Git's history

- Undoing changes can be done utilizing three different approaches (`git checkout`, `git revert`, `git reset`)
- Depends on the state of your working directory (clean or uncommitted changes)
- A pragmatic approach is to utilize the search funtionalities of a web platform such as GitHub or GitLab
- Here, only some basics will be introduced, further information is provided by https://www.atlassian.com/git/tutorials/undoing-changes or https://git-scm.com/book/en/v2/Git-Basics-Undoing-Things

git checkout

- In order to undo (a) uncommitted changes or (b) going back to an earlier commit, respectively, the command `git checkout` can be utilized

- You have multiple possibilities to undo changes. You can undo changes regarding a particular file or you can go back to an earlier commit, which may contain multiple changes (not a good practice, though)
- `git checkout -- myfile` will discard all changes with respect to `myfile`
- `git checkout -- .` (or use `git restore .`) will discard all changes in your working directory, which can include multiple files (remember the dot `.`, see section Working directory and relative paths)

- Now, let's discard all (uncommitted) changes that we made to file `text.txt` and recover the lost line 3:

```
!git status
```

```
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   test.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

```
# Check again the content of test.txt; line 3 should be missing.
!cat test.txt
```

```
l1: Branch: master
l2: Author: BW
l4: use my_fancy_data.dta, clear
l5: gen v1 = 1
l6: gen v2 = v1 + 1
l7: mean v1
```

```
!git checkout -- test.txt
```

Voilà, our beloved comment (line 3) has been risen from the dead...

```
print_file("test.txt")
```

```
l1: Branch: master
l2: Author: BW
l3: // Always start with a dumb comment:
l4: use my_fancy_data.dta, clear
l5: gen v1 = 1
l6: gen v2 = v1 + 1
l7: mean v1
```

For more information see https://www.atlassian.com/git/tutorials/using-branches/git-checkout

git revert

Put simply: `git revert` can undo a certain commit

For more information see https://www.atlassian.com/git/tutorials/undoing-changes/git-revert

git reset

Put simply: `git reset` goes back to a certain commit and discards all later commits

For more information see https://www.atlassian.com/git/tutorials/undoing-changes/git-reset

What has changed at the file level? `git show` and `git diff`

In this chapter we will learn about `git show` and `git diff`, which show differences at the file level. However, for those of you who do not feel comfortable using the command line I highly recommend meld (http://meldmerge.org/).

So far, we have only a few commits. `git log` shows all commits, the SHA1 hash and the respective commit message.

```
!git log
```

```
commit 421c480c05cf1d5d233c3728f60b1672303164f1
Author: Bernd Weiss <spam@metaanalyse.de>
Date:   Wed Nov 17 05:21:14 2021 +0100

    add new comment (M2)

commit ea3dd7685c8f1de5cab96fddaba4e003f1db29d0
Author: Bernd Weiss <spam@metaanalyse.de>
Date:   Wed Nov 17 05:21:13 2021 +0100

    add new line (M1)

commit 12b9483953c67438aa087586c82b7ba17d02975d
Author: Bernd Weiss <spam@metaanalyse.de>
Date:   Wed Nov 17 05:21:12 2021 +0100

    Initial commit
```

A brief intro to the unified diff format

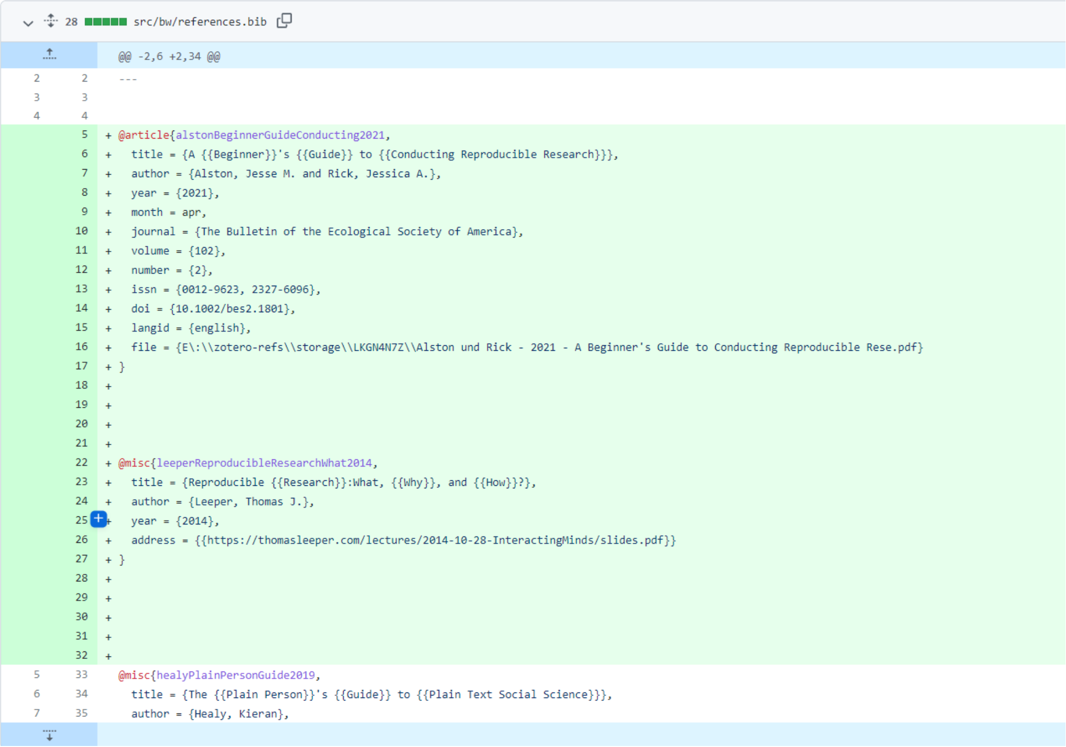
Using `git show` without any additional arguments shows the differences between the last commit and HEAD. The output follows the so called "unified diff format" (UDF). A good introduction of UDF ist provided by https://www.gnu.org/software/diffutils/manual/html_node/Detailed-Unified.html#Detailed-Unified. The following is mostly copy-and-paste from the aforementioned source. It is also imported to note that UDF utilizes so-called (c)hunks to describe changes. A hunk is a paragraph separated by an empty line.

`git show`

```
!git show
```

```
commit 421c480c05cf1d5d233c3728f60b1672303164f1
Author: Bernd Weiss <spam@metaanalyse.de>
Date:   Wed Nov 17 05:21:14 2021 +0100

    add new comment (M2)

diff --git a/test.txt b/test.txt
index ed73d1e..32af706 100644
--- a/test.txt
+++ b/test.txt
@@ -5,4 +5,5 @@ l3: // Always start with a dumb comment:
 l4: use my_fancy_data.dta, clear
 l5: gen v1 = 1
 l6: gen v2 = v1 + 1
-l7: mean v1
\ No newline at end of file
+l7: // M2: A new comment.
+mean v1
\ No newline at end of file
```

Frankly, I go to GitHub or GitLab and check the respective differences between files...



```
28 src/bw/references.bib
@@ -2,6 +2,34 @@
  ---


+ @article{alstonBeginnerGuideConducting2021,
+   title = {A {{Beginner}}'s {{Guide}} to {{Conducting Reproducible Research}}},
+   author = {Alston, Jesse M. and Rick, Jessica A.},
+   year = {2021},
+   month = apr,
+   journal = {The Bulletin of the Ecological Society of America},
+   volume = {102},
+   number = {2},
+   issn = {0012-9623, 2327-6096},
+   doi = {10.1002/bes2.1801},
+   langid = {english},
+   file = {E\:\\zotero-refs\\storage\\LKGN4N7Z\\Alston und Rick - 2021 - A Beginner's Guide to Conducting Reproducible Rese.pdf}
+ }
+
+
+
+
+ @misc{leeperReproducibleResearchWhat2014,
+   title = {Reproducible {{Research}}:What, {{Why}}, and {{How}}?},
+   author = {Leeper, Thomas J.},
+   year = {2014},
+   address = {{https://thomasleeper.com/lectures/2014-10-28-InteractingMinds/slides.pdf}}
+ }
+
+
+
+
+
  @misc{healyPlainPersonGuide2019,
    title = {The {{Plain Person}}'s {{Guide}} to {{Plain Text Social Science}}},
    author = {Healy, Kieran},
```

Branching

- In addition to being a powerful undo function, git also allows you to "toy around" with different "versions" of your text or code
- Let's assume that you wrote a first draft of a Stata program (macro). Everything works as expected. From a programming perspective, though, the program is just ugly and it is therefore quite hard to add additional functionalities.
- What I used to do was: save my original file as `my-great-program.do` and start working on a new version of the program using a file called `my-great-program_new.do`
- This is not necessary with `git branch`

Let's start with a list of files that are currently in my project folder:

```
!ls -la
```

```
total 13
```

```
drwxr-xr-x 1 weissbd DomÃ¤nen-Benutzer   0 Nov 17 05:21 .
drwxr-xr-x 1 weissbd DomÃ¤nen-Benutzer   0 Nov 17 05:21 ..
drwxr-xr-x 1 weissbd DomÃ¤nen-Benutzer   0 Nov 17 05:21 .git
-rw-r--r-- 1 weissbd DomÃ¤nen-Benutzer 203 Nov 17 05:21 test.txt
```

```
!git status
```

```
On branch main
nothing to commit, working tree clean
```

What branches are available? Once we have more than one branch, the `*` shows which branch is active (or: in wich branch we are in)

```
!git branch
```

```
* main
```

Create a new branch called `testing`

```
!git branch testing
```

```
!git branch
```

```
* main
  testing
```

How do we get into the `testing` branch? Use `git checkout testing`

```
!git checkout testing
!git branch
```

```
Switched to branch 'testing'
```

```
  main
* testing
```

Create a new file `testingfile`

```
!touch testingfile
!echo "in testing" > testingfile
!ls -la
```

```
total 14
```

```
drwxr-xr-x 1 weissbd DomÃ¤nen-Benutzer   0 Nov 17 05:21 .
drwxr-xr-x 1 weissbd DomÃ¤nen-Benutzer   0 Nov 17 05:21 ..
drwxr-xr-x 1 weissbd DomÃ¤nen-Benutzer   0 Nov 17 05:21 .git
-rw-r--r-- 1 weissbd DomÃ¤nen-Benutzer 203 Nov 17 05:21 test.txt
-rw-r--r-- 1 weissbd DomÃ¤nen-Benutzer  15 Nov 17 05:21 testingfile
```

```
!cat testingfile
```

```
"in testing"
```

```
!git add testingfile
!git commit -m "new branch testing"
```

```
[testing d21b859] new branch testing
 1 file changed, 1 insertion(+)
 create mode 100644 testingfile
```

Switch back to branch `main` (and `cat testingfile` should result in an error message, since there is no `testingfile` in branch `main`)

```
!git checkout main
!cat testingfile
```

```
Switched to branch 'main'
```

```
cat: testingfile: No such file or directory
```

Now, we can use `merge` to combine `main` and `testing`

```
!git merge testing
```

```
Updating 421c480..d21b859
Fast-forward
 testingfile | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 testingfile
```

```
!ls -la
```

```
total 14
drwxr-xr-x 1 weissbd DomÃ¤nen-Benutzer   0 Nov 17 05:21 .
drwxr-xr-x 1 weissbd DomÃ¤nen-Benutzer   0 Nov 17 05:21 ..
drwxr-xr-x 1 weissbd DomÃ¤nen-Benutzer   0 Nov 17 05:21 .git
-rw-r--r-- 1 weissbd DomÃ¤nen-Benutzer 203 Nov 17 05:21 test.txt
-rw-r--r-- 1 weissbd DomÃ¤nen-Benutzer  15 Nov 17 05:21 testingfile
```

```
!cat testingfile
```

```
"in testing"
```

Working with remote repositories

As mentioned in the introduction, git is especially powerful when it comes to collaborative work. In order to work with others, you need some sort of connetion to these other person(s). The one I am discussing here is having a central repository C. Let us assume that you (x) have two other collaborators y and z. Then x (that's you), as well as y and z need to synchronise with the same repository C. There also exists another model which is based on a decentralized approach, where you could indivdually sync with x-y, x-z, y-z etc.

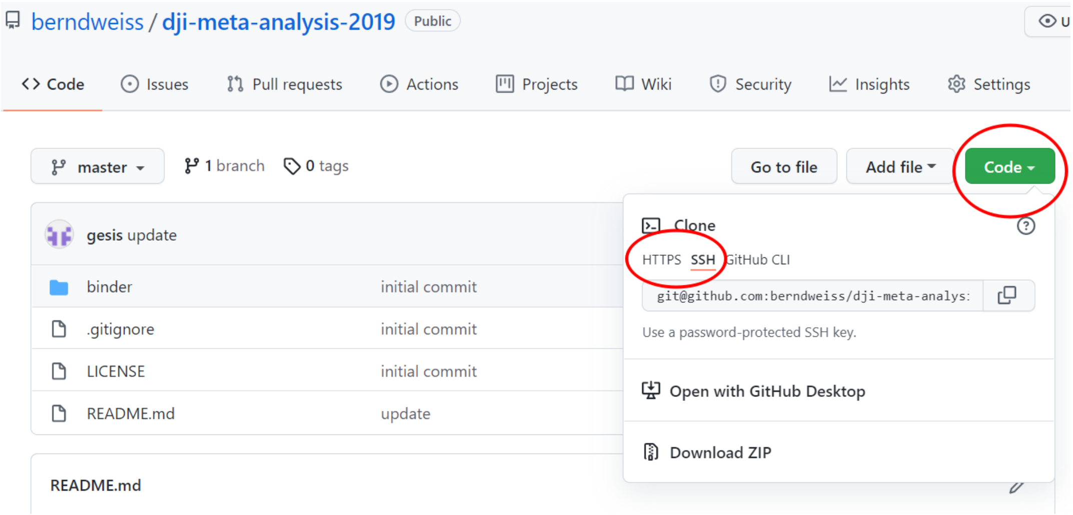
Establishing a connection to a remote repository

There are two ways to establish a connection to a remote repository:

1. Clone a remote repository via `git clone ....`
2. Setting up a new remote repository via `git remote add <name> <url>`.

Cloning a remote repository

- Cloning a remote repository via GitHub/GitLab/… is quite easy
- Visit the website, on GitHub look for the green "Code" button, see also screenshot below
- Decide wether you would like to use the HTTPS or SSH protocoll
- Copy the link and execute `git clone`
- Here is an example using my workshop on "Meta-Analysis in Social Research", see https://github.com/berndweiss/dji-meta-analysis-2019
- Open a CLI and execute `git clone git@github.com:berndweiss/dji-meta-analysis-2019.git`



gesis update

binder	initial commit
.gitignore	initial commit
LICENSE	initial commit
README.md	update

git@github.com:berndweiss/dji-meta-analys:

Use a password-protected SSH key.

```
!cd
```

```
E:\syncwork\projects\confer\ps2021-10-ws-repro-research\src\bw\git_test_folder
```

Adding a remote repository via `git remote add ...`

Using the git command `git remote add <name> <url>`. The usual name for `<name>` is `origin`, however, feel free to choose another name. The `<url>` for this repository looks like `git@git.gesis.org:weissbd/ps2017-xx-intro2git.git`; another example is this one: `git@github.com:berndweiss/ps2017-11_porto-campbell-ma-workshop.git`. The url can be found in the respective github/gitlab repository.

The most convenient way in working with remove respsitories is using SSH. In order to utilize SSH, the remote url has to be start with `git@git...`.

It is also possible to use the HTTPS protocol. In these cases the urls look like so `https://example.com/path/to/repo.git`.

Delete remote branch

`git push <remote_name> --delete <branch_name>`, e.g. `git push origin --delete my_branch`

Dowloading a remote branch that is not on your computer (yet)

Just run a simple `git pull` (see https://stackoverflow.com/a/2294385). Then, on your local repository checkout to that remote branch, e.g. `git checkput indepday`:

```
Switched to a new branch 'indepday'
Branch 'indepday' set up to track remote branch 'indepday' from 'origin'.
```

After checkout to `indepday`, git automatically starts tracking the new branch.

Exercise

- Start the Git Bash
- Clone the respository of my workshop https://github.com/berndweiss/dji-meta-analysis-2019
- Change into the newly created directory
- List the Git history via `<your input --oneline>` (the `--oneline` is very handy) and determine the first 7 SHA1 digits

Setting up a Git repository locally and remotly

Exercise

- In a previous exercise (see [Exercise](#)), you have created your own repository (let's call it `your-new-repo`)
- Now, go to your GitHub account and create a new repository on GitHub
    - Startpage -> tab "Repositories" -> green button "New"
    - Enter a new "Repository name"
    - Make it "Private" (unless you have something important to share)
    - Do **not** check any of the "Initialize this repository with" boxes
    - Hit the green "Create repository" button
    - Choose SSH or HTTPS as protocol ("Quick setup — if you've done this kind of thing before")
    - Look for "…or push an existing repository from the command line"
    - SSH:
        - Copy the line `git remote add origin git@github.com:berndweiss/your-repo-name.git`
        - Execute the command `git remote add origin git@github.com:berndweiss/your-repo-name.git` in the Git Bash in your local repository (`your-new-repo`)
    - HTTPS:
        - Copy the line `git remote add origin https://github.com/berndweiss/your-repo-name.git`
        - Execute the command `git remote add origin https://github.com/berndweiss/your-repo-name.git` in the Git Bash in your local repository (`your-new-repo`)

- Make sure that `git status` shows a clean repository
- Now you can run your first `git push origin main`
- Reload the GitHub page via F5; you now should see the content of your local repo `your-new-repo`
- …
- You can delete a GitHub repository via the tab “Settings” -> “Options”, then scroll down -> “Danger Zone” -> “Delete this repository”

There is much more…

- Commit hygiene http://www.ericbmerritt.com/2011/09/21/commit-hygiene-and-git.html
- `.gitconfig`
- `.gitignore`
- …
- A nice tutorial in German: ”git - Der einfache Einstieg, eine einfache Anleitung, um git zu lernen. Kein Schnick-Schnack ;) https://rogerdudler.github.io/git-guide/index.de.html

References

[AR21]
Jesse M. Alston and Jessica A. Rick. A Beginner's Guide to Conducting Reproducible Research. *The Bulletin of the Ecological Society of America*, April 2021. doi:10.1002/bes2.1801.

[Bar18]
Lorena A. Barba. Terminologies for Reproducible Research. *arXiv:1802.03311 [cs]*, February 2018. arXiv:1802.03311.

[BD95]
Jonathan B. Buckheit and David L. Donoho. WaveLab and Reproducible Research. In Anestis Antoniadis, Georges Oppenheim, P. Bickel, P. Diggle, S. Fienberg, K. Krickeberg, I. Olkin, N. Wermuth, and S. Zeger, editors, *Wavelets and Statistics*, volume 103 of Lecture Notes in Statistics, pages 55–81. Springer New York, New York, NY, 1995. doi:10.1007/978-1-4612-2544-7.

[Div18]
Patrick Divine. String Case Styles: camel, Pascal, Snake, and Kebab Case. https://betterprogramming.pub/string-case-styles-camel-pascal-snake-and-kebab-case-981407998841, 2018.

[Fit12]
Michael Fitzgerald. *Introducing Regular Expressions*. O'Reilly, Beijing, Köln, 1. ed edition, 2012. ISBN 978-1-4493-9268-0.

[For17]
Tiago Forte. The PARA Method: a Universal System for Organizing Digital Information. https://fortelabs.co/blog/para/, February 2017.

[FP17]
Jeremy Freese and David Peterson. Replication in Social Science. *Annual Review of Sociology*, 43(1):147–165, July 2017. doi:10.1146/annurev-soc-060116-053450.

[GFI16]
Steven N. Goodman, Daniele Fanelli, and John P. A. Ioannidis. What does research reproducibility mean? *Science Translational Medicine*, 8(341):341ps12–341ps12, June 2016. doi:10.1126/scitranslmed.aaf5027.

[Hea19]
Kieran Healy. The Plain Person's Guide to Plain Text Social Science. https://plain-text.co/, October 2019.

[Lee14]
Thomas J. Leeper. Reproducible Research:what, Why, and How? 2014.

[Narkebski16]
Jakub Narębski. *Mastering Git: Attain Expert-Level Proficiency with Git for Enhanced Productivity and Efficient Collaboration by Mastering Advanced Distributed Version Control Features*. Packt Publishing, Birmingham, 2016. ISBN 978-1-78355-375-4.

[Pen11]
Roger D. Peng. Reproducible Research in Computational Science. *Science*, 334(6060):1226–1227, February 2011. doi:10.1126/science.1213847.

[PF16]
Stephen R. Piccolo and Michael B. Frampton. Tools and techniques for computational reproducibility. *GigaScience*, 5(1):30, December 2016. doi:10.1186/s13742-016-0135-4.

[SWeiss20]
Ines Schaurer and Bernd Weiß. Investigating selection bias of online surveys on coronavirus-related behavioral outcomes: an example utilizing the GESIS Panel Special Survey on the Coronavirus SARS-CoV-2 Outbreak in Germany. *Survey Research Methods*, 2(14):103–108, June 2020. doi:10.18148/SRM/2020.V14I2.7751.

[SLP14]
Victoria Stodden, Friedrich Leisch, and Roger D. Peng. *Implementing Reproducible Research*. CRC Press, Boca Raton, 2014.

Appendix

More Materials, Links etc.

Praxistipps des Open-Science-Magazins der ZBW – Leibniz-Informationszentrum Wirtschaft

- Teil 1: Praxistipps, wie Wirtschaftsforschende Open Science in ihrer täglichen wissenschaftlichen Arbeit umsetzen können
  - URL: https://open-science-future.zbw.eu/open-science-praxistipps-zum-download-i/
  - PDF: https://open-science-future.zbw.eu/wp-content/uploads/2021/09/os-magazin-print_teil1.pdf
- Teil 2: Tipps für die Wissenschaftskommunikation mit Social Media
  - URL: https://open-science-future.zbw.eu/open-science-praxistipps-zum-download-ii/
  - PDF: https://open-science-future.zbw.eu/wp-content/uploads/2021/09/os-magazin-print_teil2.pdf
- Teil 3: 20 Worksheets für mehr Open Science im wirtschaftswissenschaftlichen Alltag (**berührt konkret Themen des Workshops**)
  - URL: https://open-science-future.zbw.eu/open-science-praxistipps-zum-download-iii/
  - PDF: https://open-science-future.zbw.eu/wp-content/uploads/2021/09/os-magazin-print_teil3.pdf

By Bernd Weiß