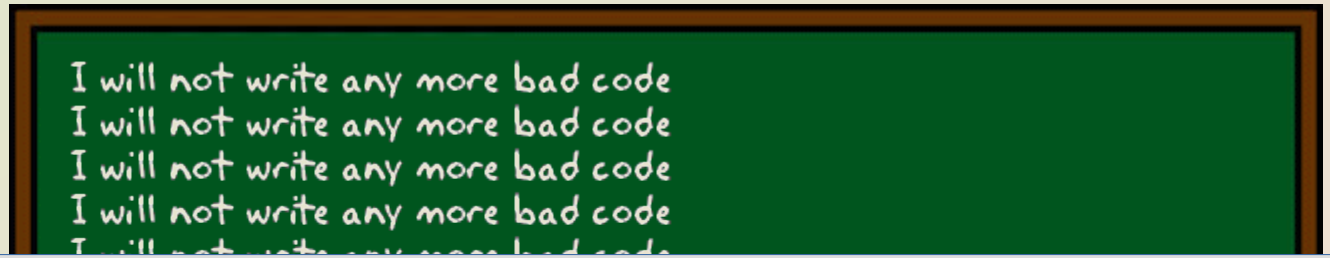
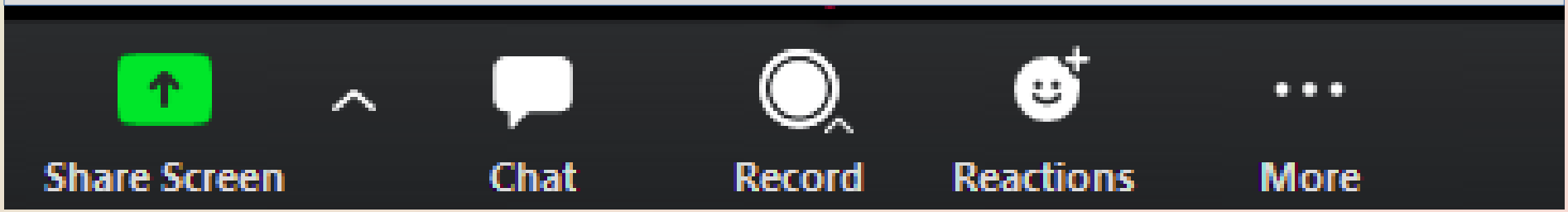
Funded by DFG




# This lecture will be recorded




Help-desk workshop for statistical analyses in R, 2021



UNIVERSITÄT BERN

Funded by DFG

# good practices in programming

Help-desk workshop for statistical analyses in R, 2021



UNIVERSITÄT BERN

Funded by DFG

# Main Sources of Content

Two-day workshop "Best practices in Programming"

ETH zürich

Scientific IT Services

SIB

Swiss Institute of Bioinformatics

**BES guide** :
https://www.britishecologicalsociety.org/wp-content/uploads/2019/06/BES-Guide-Reproducible-Code-2019.pdf?utm_source=web&utm_medium=web&utm_campaign=better_science
**Course** : https://www.sib.swiss/training/course/20210707_BPP



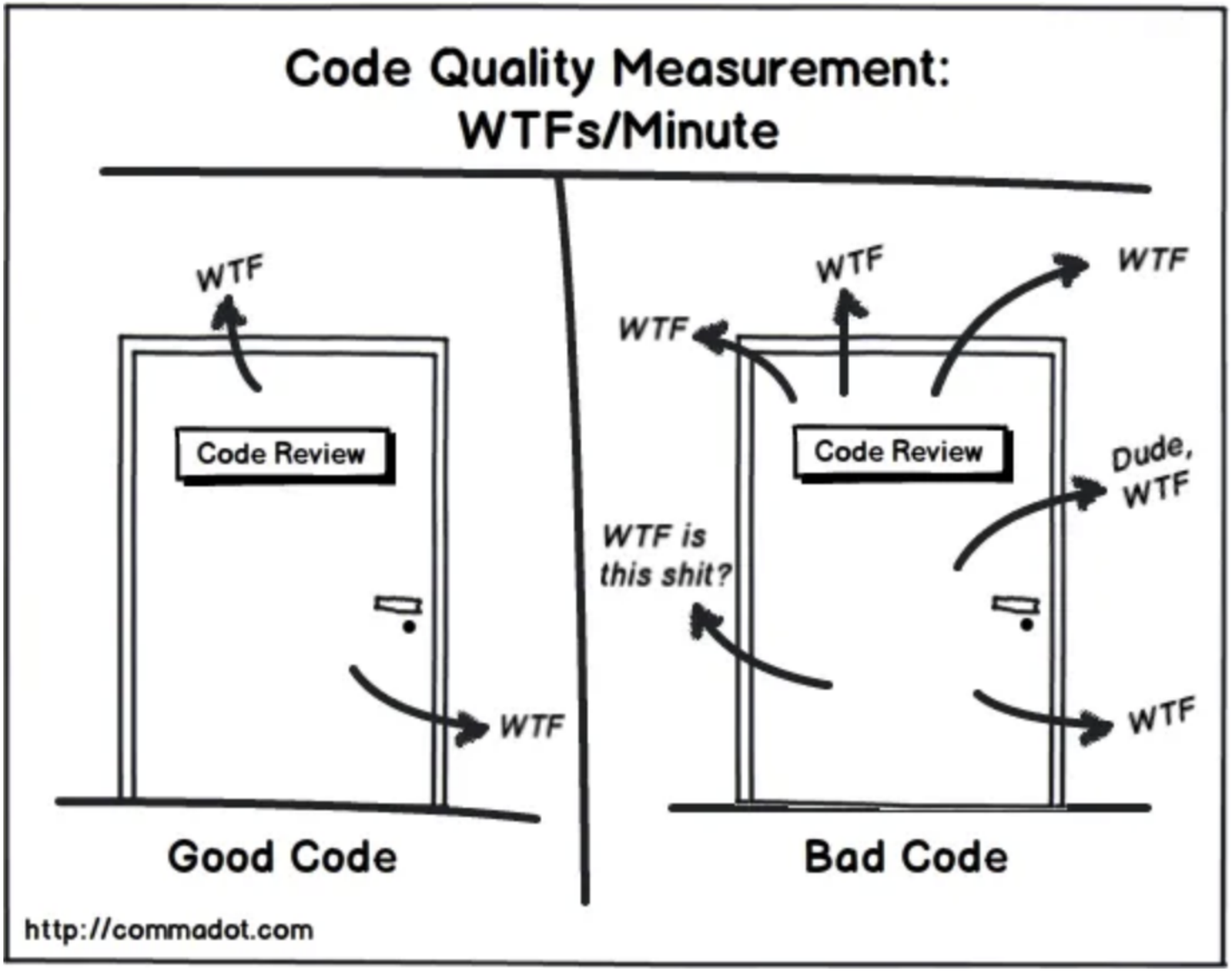
UNIVERSITÄT BERN

# Content

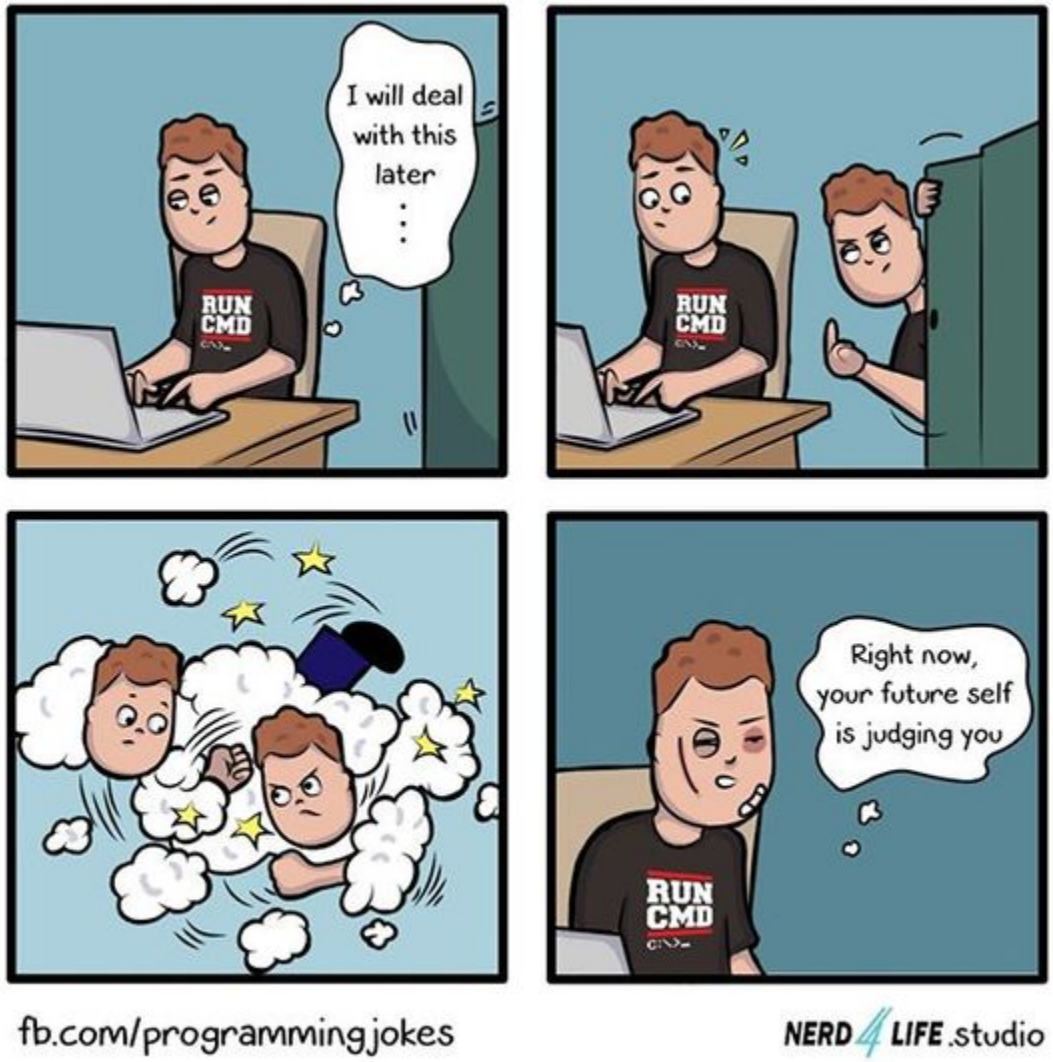
- why care?
- within your script
- around your scripts
  - version control
- around your project
- Take home/ Reflection
- additional material

# why care?

# why care? individual level



# treat future-you well



# why care? individual level

- decrease future wtfs per minute factor
  - happier programming experience
- keep things easier for you and your collaborators
- facilitates error-detection
- enables sharing code

"Code is read more often than it is written"

# why care? societal level

SORTEE

Society for Open, Reliable, and Transparent Ecology and Evolutionary Biology

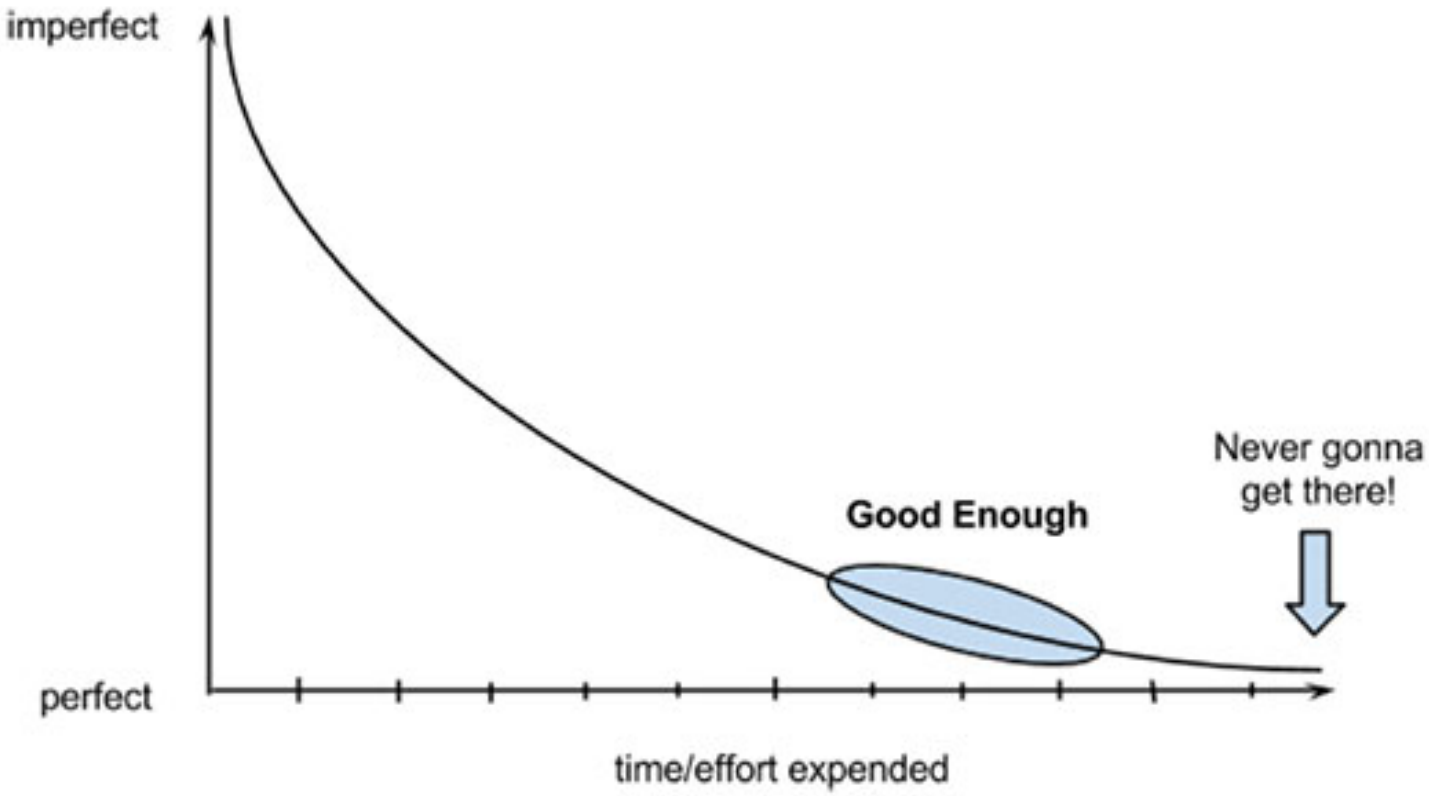
NEWS | PLANTS & ANIMALS

## Psychology's replication crisis inspires ecologists to push for more reliable research

New society aims for transparency and culture change in ecology

9 DEC 2020 • BY CATHLEEN O'GRADY

# about perfection ...



- within your script

# keep it simple!

```r
###
# EXAMPLE - KEEP IT SIMPLE!  ####
#
# bad example : all in one line
info_data <- data.table::fread(paste(pathtodata, "/data_assembly/helper_data/info_data.csv", sep = ""), header=T)
#
# good example : use many lines
info_data <- data.table::fread(
  paste(pathtodata,
        "/data_assembly/helper_data/info_data.csv",
        sep = ""),
  header=T)

# add structure to your file ####
# if you add four "#" in the end of the title, RStudio will allow
#   you to fold in and out the section
```

```r
17  ###
18  # EXAMPLE - KEEP IT SIMPLE!
30  # add structure to your file
```

- use the lines
- use indenting
- add structure to your file

https://github.com/biodiversity-exploratories-synthesis/useful_functions/blob/main/2021_synthesis_Rhelpdesk_good_programming_practices.R

# use a code style

- Good coding style is like correct punctuation: you can manage without it, **butitsuremakesthingseasiertoread**.
- e.g. the tidyverse style guide

```r
# Good
average <- mean(feet / 12 + inches, na.rm = TRUE)
# Bad
average<-mean(feet/12+inches,na.rm=TRUE)

# Good
do_something_very_complicated(
  something = "that",
  requires = many,
  arguments = "some of which may be long"
)
# Bad
do_something_very_complicated("that", requires, many, arguments,
                              "some of which may be long"
)
```

tidyverse style guide : https://style.tidyverse.org/index.html
short version : http://adv-r.had.co.nz/Style.html
R packages to help with code style : https://styler.r-lib.org/ and https://github.com/jimhester/lintr

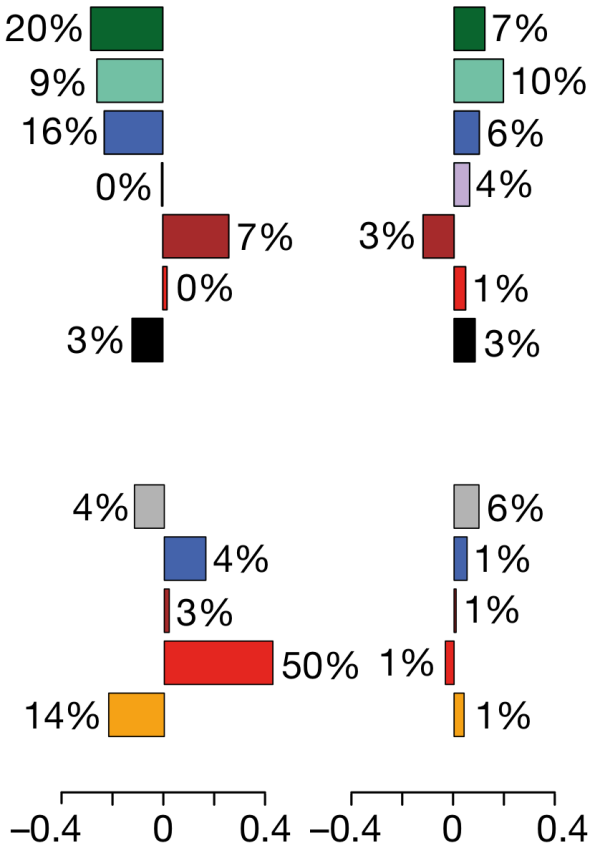
# use a code style

- Variable and function names with lowercase letters, separated by underscore "_"
  - Good : `day_one, day_1`
  - Bad : `DayOne, dayone`
- variable names should be nouns and function names should be verbs
  - `day_one`
  - `calculate_time_span`
- avoid re-using names of common functions and variables. This will cause confusion for the readers of your code
  - `Bad : T ← FALSE`
  - `Bad : mean ← function(x) sum(x)`
- more recommendation in the tidyverse styleguide

tidyverse style guide : https://style.tidyverse.org/index.html
short version : http://adv-r.had.co.nz/Style.html
R packages to help with code style : https://styler.r-lib.org/ and https://github.com/jimhester/lintr

# comment your code

- code without comments can be as useless as a plot without axis labels
- comment the unexpected
  - everything not obvious from your code
  - e.g. if you use a solution from the internet, provide the link in a comment
  - explain alternative values for parameters, or things you had to look up when you wrote the code
- better too much than too few comments




```
//When I wrote this, only God and I understood what I was doing
//Now, God only knows
```

# comment your code

```r
###
# EXAMPLE 2 - COMMENT YOUR CODE
#
# Bad example : comment the obvious
# read table example.csv
info_data <- data.table::fread("example.csv", header = T)
#
# Good example : comment the unexpected
# use package data.table to read in very large table "example.csv"
info_data <- data.table::fread("example.csv", header = T)
```

https://github.com/biodiversity-exploratories-synthesis/useful_functions/blob/main/2021_synthesis_Rhelpdesk_good_programming_practices.R

# make a script header

- Use the first few lines of a script to …
  - explain the purpose of the script
  - date and creator
  - state requirements / dependencies
    - e.g. previous script's output
  - (voluntary) include an example

```
#####################################
#                                   #
# GOOD PROGRAMMING PRACTICES        #
#                                   #
#####################################
# Biodiversity Exploratories Synthesis R helpdesk
# Noëlle Schenk 05.02.2021
#
# this file contains examples of practices to improve the
# readability of your code.
```

https://github.com/biodiversity-exploratories-synthesis/useful_functions/blob/main/2021_synthesis_Rhelpdesk_good_programming_practices.R

# make a script header

- libraries go first

```
library(data.table)
library(reshape2)
library(tidyr)
```

- hard-coded variables go second

```
path_to_data <- "example_data_synthesis_helpdesk/"
path_to_output <- "R_outputs_and_plots/"
```

  - relative paths over absolute paths

```
absolute_path <- "C:/user/yourname/Documents/Project1/Rscripts/data/info_data.csv"
relative_path <- "/data/info_data.csv"
```

- set a seed

```
set.seed(12)
```

https://towardsdatascience.com/best-practices-for-r-programming-ec0754010b5a

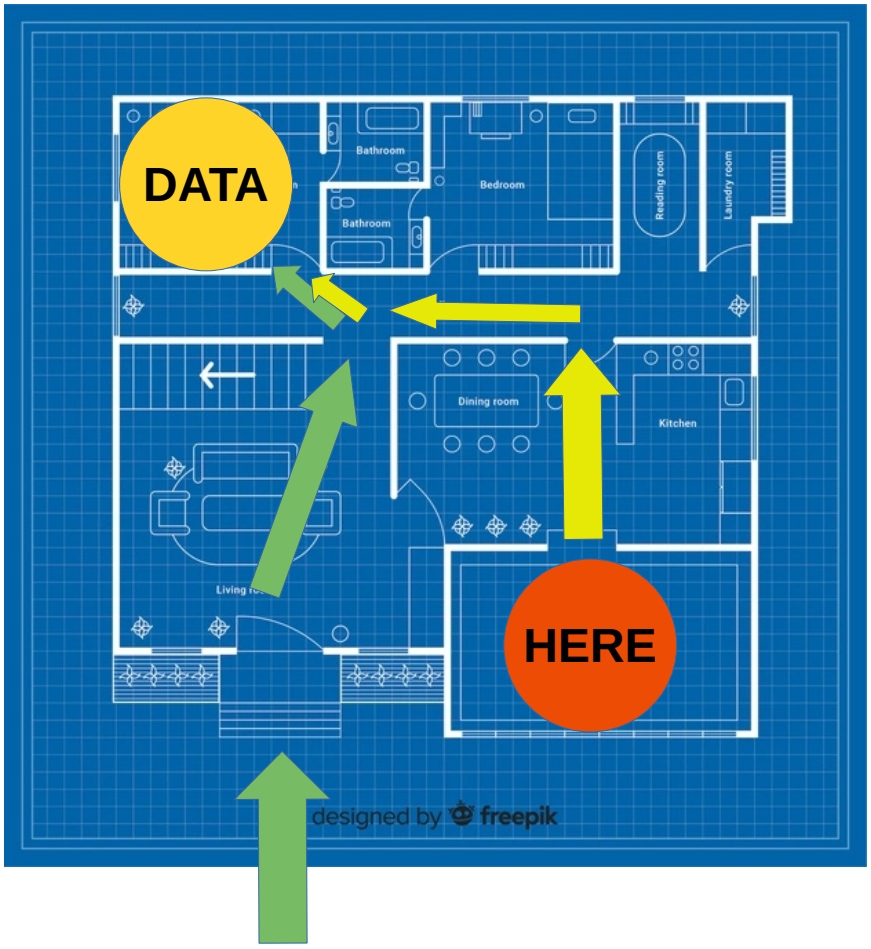
# prefer relative file paths

- relative file paths

describe path from your working directory on

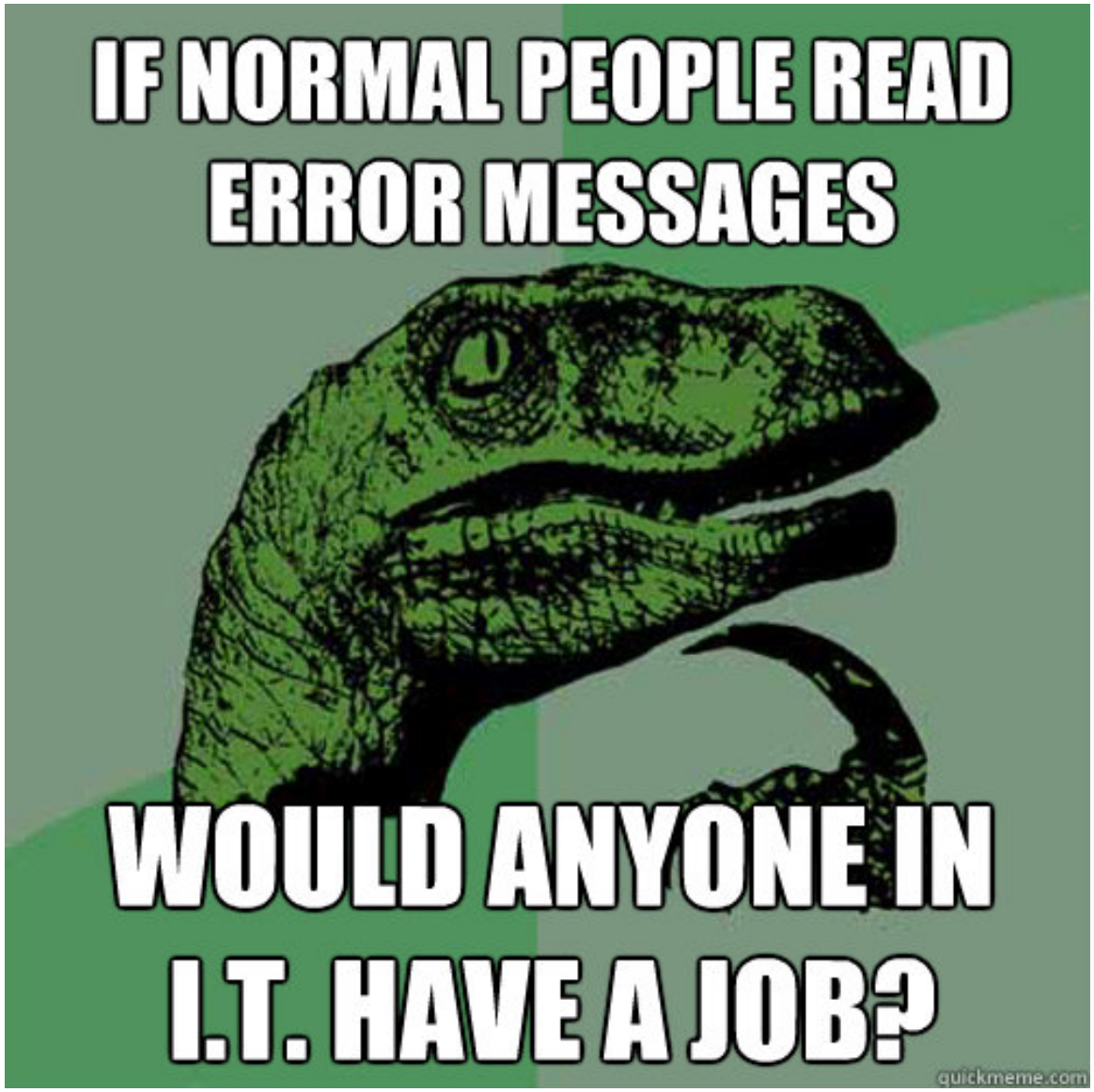
- absolute file paths

describe a path from your home directory on



```
absolute_path <- "C:/user/yourname/Documents/Project1/Rscripts/data/info_data.csv"
relative_path <- "/data/info_data.csv"
```

# read error messages



# read error messages

```
> # read the error messages
> x <- as.factor("A", "B", "C")
Error in as.factor("A", "B", "C") : unused arguments ("B", "C")
>
```

# read error messages

```
> # read the error messages
> x <- as.factor("A", "B", "C")
Error in as.factor("A", "B", "C") : unused arguments ("B", "C")
>
```

```
>
> help("as.factor")
>
```

factor {base} R Documentation

## Factors

### Description

The function `factor` is used to encode a vector as a factor (the terms 'category' and 'enumerated type' are also used for factors). If argument `ordered` is TRUE, the factor levels are assumed to be ordered. For compatibility with S there is also a function `ordered`.

`is.factor`, `is.ordered`, `as.factor` and `as.ordered` are the membership and coercion functions for these classes.

### Usage

```
factor(x = character(), levels, labels = levels,
       exclude = NA, ordered = is.ordered(x), nmax =

ordered(x, ...)
```

# read error messages

Error in as.factor("A", "B", "C") : unused arguments ("B", "C")

Alle | Bilder | News | Videos | Shopping | Mehr | Einstellungen | Suchfilter

Ungefähr 7'910'000 Ergebnisse (0.83 Sekunden)

Tipp: Begrenze die Suche auf **deutschsprachige** Ergebnisse. Du kannst deine Suchsprache in den Einstellungen ändern.

stackoverflow.com › questions › di... ▾ Diese Seite übersetzen

## Different result in the case of `factor` or `as.factor`? - Stack ...

03.04.2019 — Problem is as stated in the **error** message. You are passing **arguments** which are not present for as.**factor** . If you read ?as.**factor** you see the ...

1 Antwort

| | | |
|---|---|---|
| How to continue to use **factor** where one level is ... | 2 Antworten | 16. Dez. 2015 |
| **Unused arguments** in R - Stack Overflow | 5 Antworten | 22. Apr. 2012 |
| Why use as.**factor**() instead of just **factor** ... | 1 Antwort | 25. Apr. 2017 |
| **Error** with select function from dplyr - Stack Overflow | 1 Antwort | 20. Sept. 2017 |

Weitere Ergebnisse von stackoverflow.com

stat.ethz.ch › base › help › factor ▾ Diese Seite übersetzen

## Factors - R

The function **factor** is used to encode **a** vector as **a factor** (the terms 'category' and ... If **argument** ordered is TRUE , the **factor** levels are assumed to be ordered. ... are **unused** levels: in that case, **a factor** with the reduced level set is returned. ... z # ordered levels '**A** < **B** < **C**' **factor**(z, exclude = "**C**") # does exclude **factor**(z, ...

# read error messages

Error in as.factor("A", "B", "C") : unused arguments ("B", "C")

Alle | Bilder | News | Videos | Shopping | Mehr | Einstellungen | Suchfilter

Ungefähr 7'910'000 Ergebnisse (0.83 Sekunden)

Tipp: Begrenze die Suche auf **deutschsprachige** Ergebnisse. Du kannst deine Suchsprache in den Einstellungen ändern.

```
# the function encodes a vector, I gave the elements of a vector,
#    not the vector itself.
x <- as.factor(c("A", "B", "C"))
```

| | | |
|---|---|---|
| How to continue to use **factor** where one level is ... | 2 Antworten | 16. Dez. 2015 |
| **Unused arguments** in R - Stack Overflow | 5 Antworten | 22. Apr. 2012 |
| Why use as.**factor**() instead of just **factor** ... | 1 Antwort | 25. Apr. 2017 |
| **Error** with select function from dplyr - Stack Overflow | 1 Antwort | 20. Sept. 2017 |

Weitere Ergebnisse von stackoverflow.com

stat.ethz.ch › base › help › factor ▾ Diese Seite übersetzen

Factors - R

The function **factor** is used to encode **a** vector as **a factor** (the terms 'category' and ... If **argument** ordered is TRUE, the **factor** levels are assumed to be ordered. ... are **unused** levels: in that case, **a factor** with the reduced level set is returned. ... z # ordered levels 'A' < 'B' < 'C' **factor**(z, exclude = "C") # does exclude **factor**(z, ...

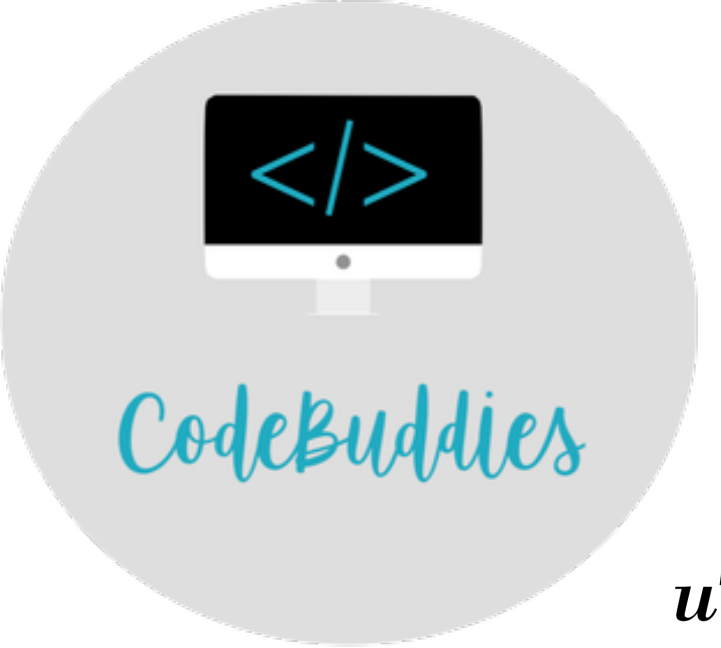


UNIVERSITÄT BERN

# how to improve my coding?

- be brave to try things out!
- step by step – keep next steps in mind (e.g. functions)
- read other people's code : find a code buddy
  - learn from them
  - improve their code quality
- exchange about coding

  https://www.tidytuesday.com/
- enjoy coding!

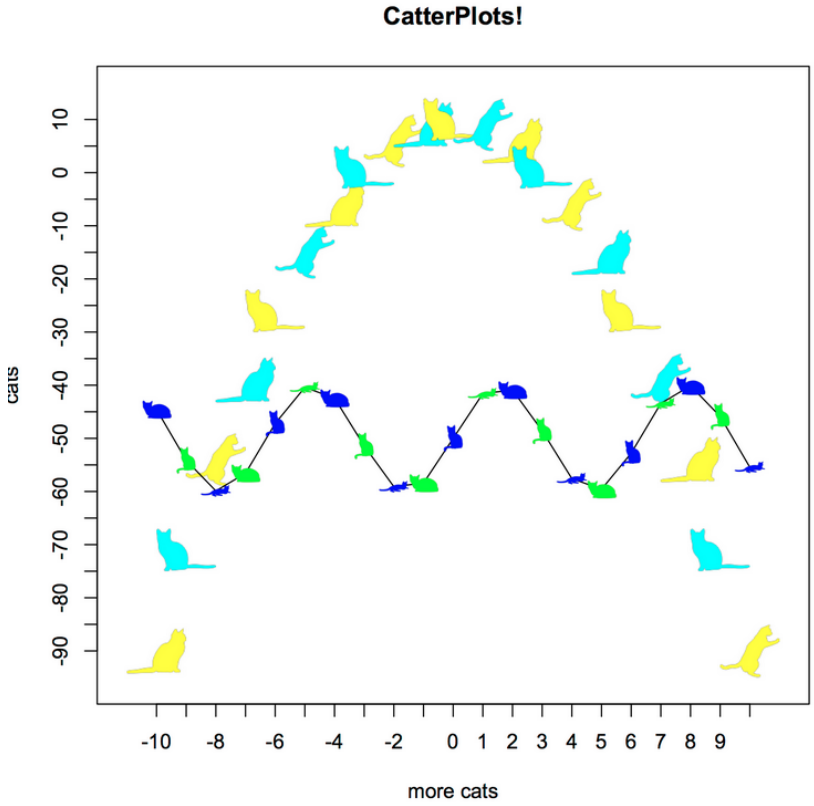


# enjoy coding

# enjoy programming

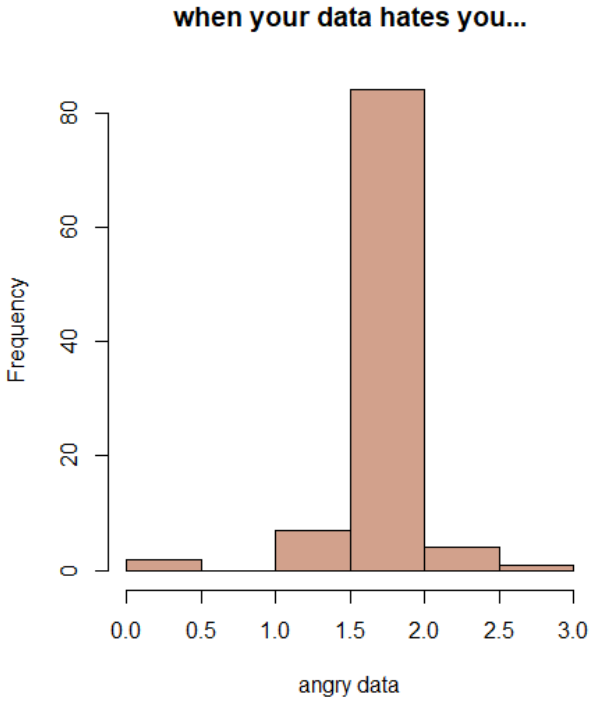
- funny packages :
  - CatterPlots : https://github.com/Gibbsdavidl/CatterPlots
  - praise : https://github.com/rladies/praise

```
#> [1] "You are super-excellent!"
```



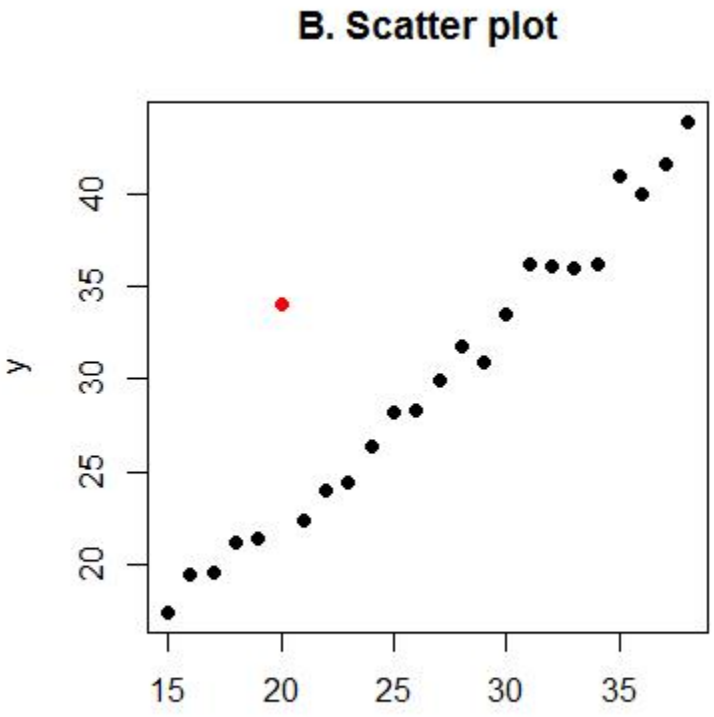

```r
# plotting of R objects
plot <- function (x, y, ...)
{
  if (is.function(x) &&
      is.null(attr(x, "class")))
  {
   if (missing(y))
      y <- NULL

   # check of ylab argument
   hasylab <- function(...)
     !all(is.na(
       pmatch(names(list(...)),
              "ylab")))
```



# Test your code

- Does the line you just wrote does what you were expecting?
  - simple summary stats (number of NA, min and max values)
  - visual checking : inspect table with “View()”, plot, print out a column
  - check the format (number of lines, data type (e.g. factor, character, numeric))
- keep this little checks in your script – if you change something later, it will help you to check if everything still works



# Test your code

```r
###
# EXAMPLE - Test your code
#
# create example data.frame
example <- data.frame(species = c("Erithacus rubecula", "Arion vulgaris",
                                  "Plantago Lanceolata"),
                      abundance = c(2, NA, 10))
# action : recode NA value to 0
#    because Arion vulgaris was measured, but no individuals were found
example[which(example$species == "Arion vulgaris"), 2] <- 0
#
# test : did I do what I wanted?
sum(is.na(example$abundance)) # no NA values any more
sum(is.na(example$species))
min(example$abundance)        # min is 0 now, as expected
max(example$abundance)
View(example)
plot(example)
example$abundance
```

# `unit testing` (advanced)

- entangled software **breaks upon tiny changes** (e.g. new R version, package updates, bug fixes, ...)
- write integrated test functions which check if your code works right
- unit testing in R :
- e.g. **`testthat`**

https://towardsdatascience.com/unit-testing-in-r-68ab9cc8d211

# unit testing (advanced)

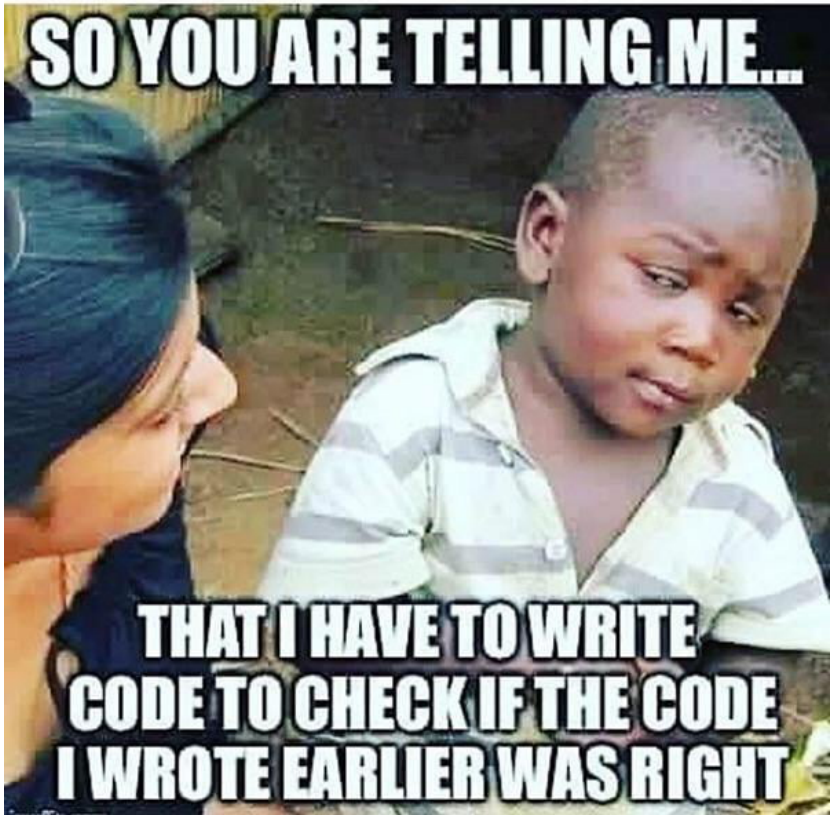
## The Art of Software Testing, Third Edition

Editor(s): Glenford J. Myers, Tom Badgett, Corey Sandler

First published: 2 January 2012

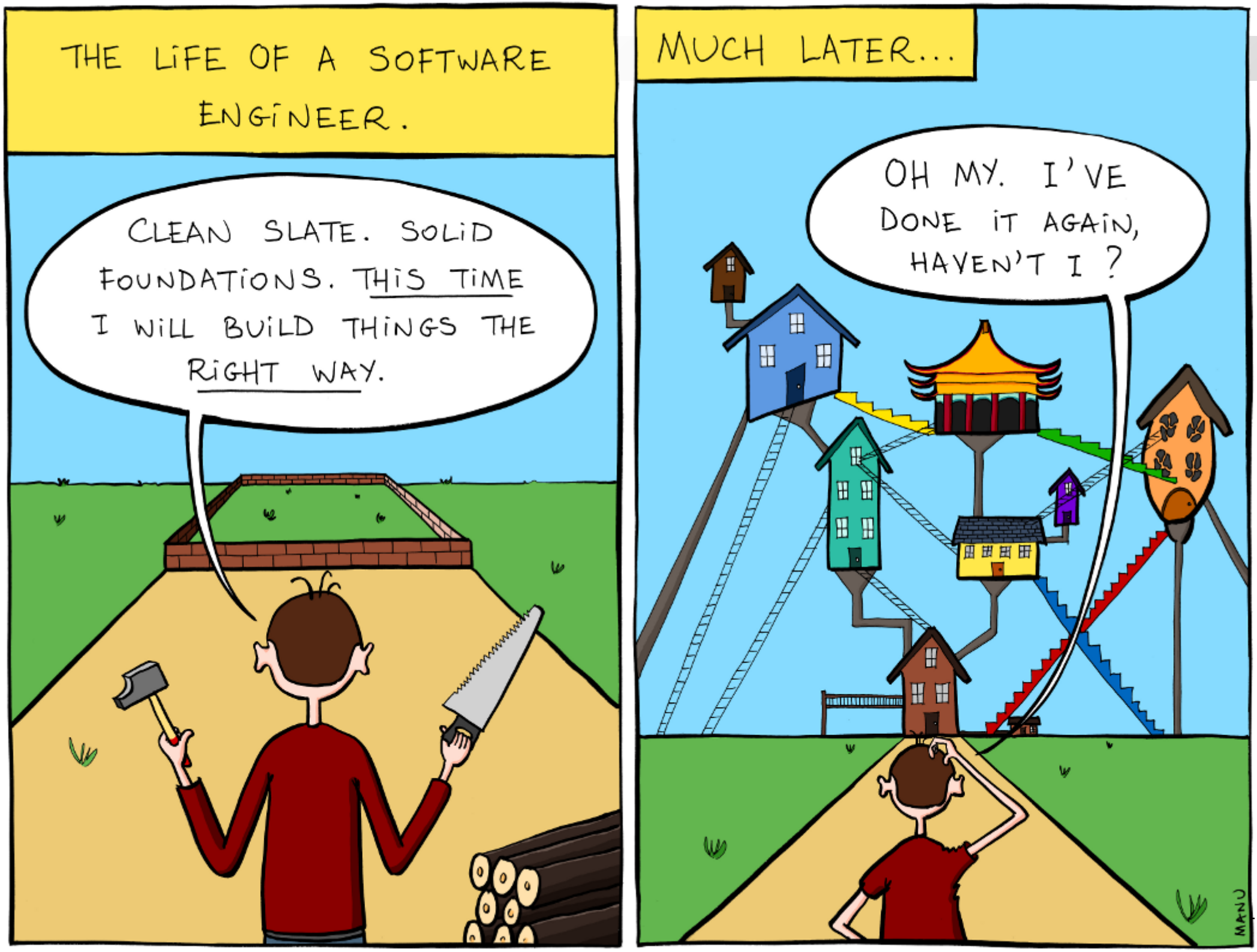
Print ISBN: 9781118031964 | Online ISBN: 9781119202486 | DOI: 10.1002/9781119202486

Copyright © 2012 by Word Association, Inc. All rights reserved



# Dare to start over

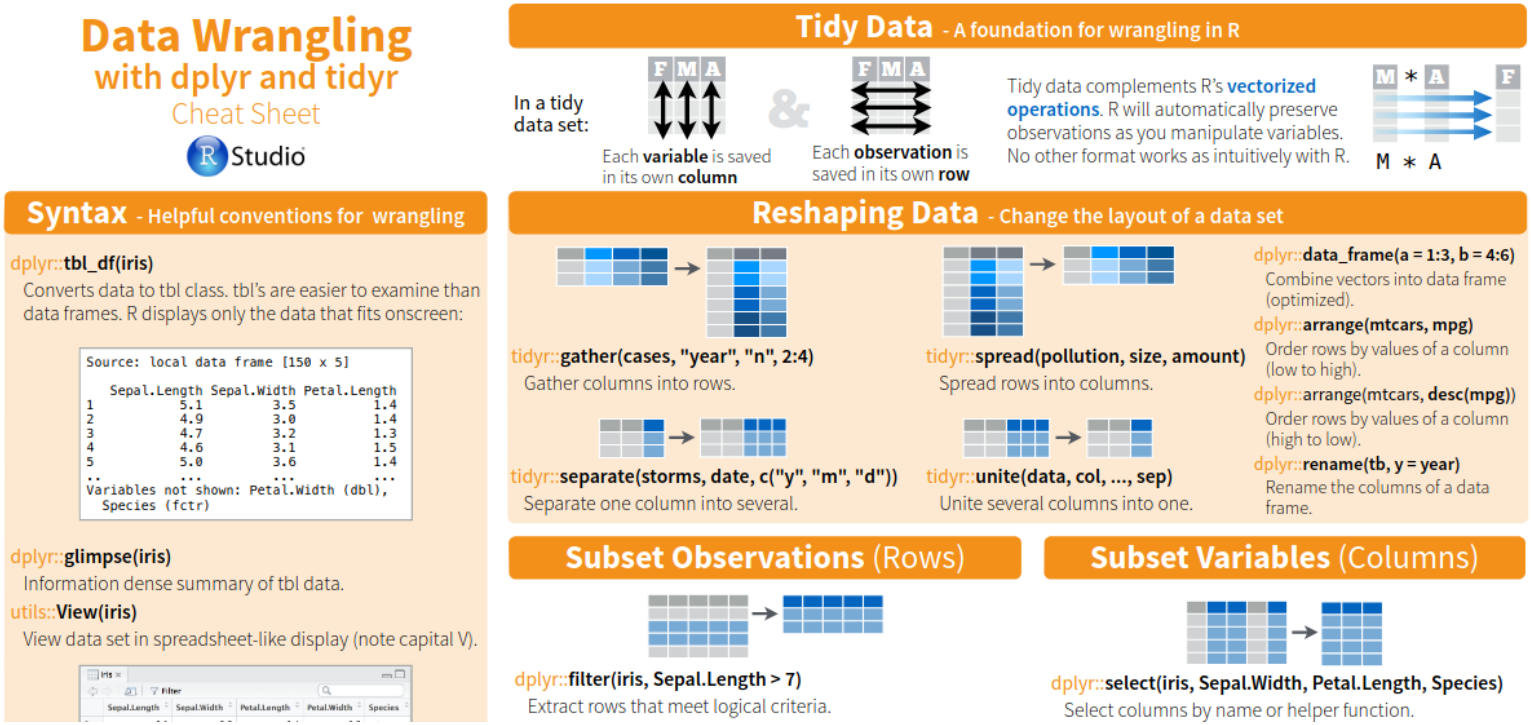
- be brave to start over
- keep the old version (see version control later today)
- get inspired by your old version, but don’t copy it.
- take the chance to improve
  - new functions
  - new coding techniques


THE LIFE OF A SOFTWARE ENGINEER.
CLEAN SLATE. SOLID FOUNDATIONS. THIS TIME I WILL BUILD THINGS THE RIGHT WAY.
MUCH LATER...
OH MY. I'VE DONE IT AGAIN, HAVEN'T I ?
MANU

- **around your script**

# use cheat sheets

- make your own or use cheatsheets provided by others
- RStudio > Help > Cheatsheets or Google “Cheatsheet <topic>”

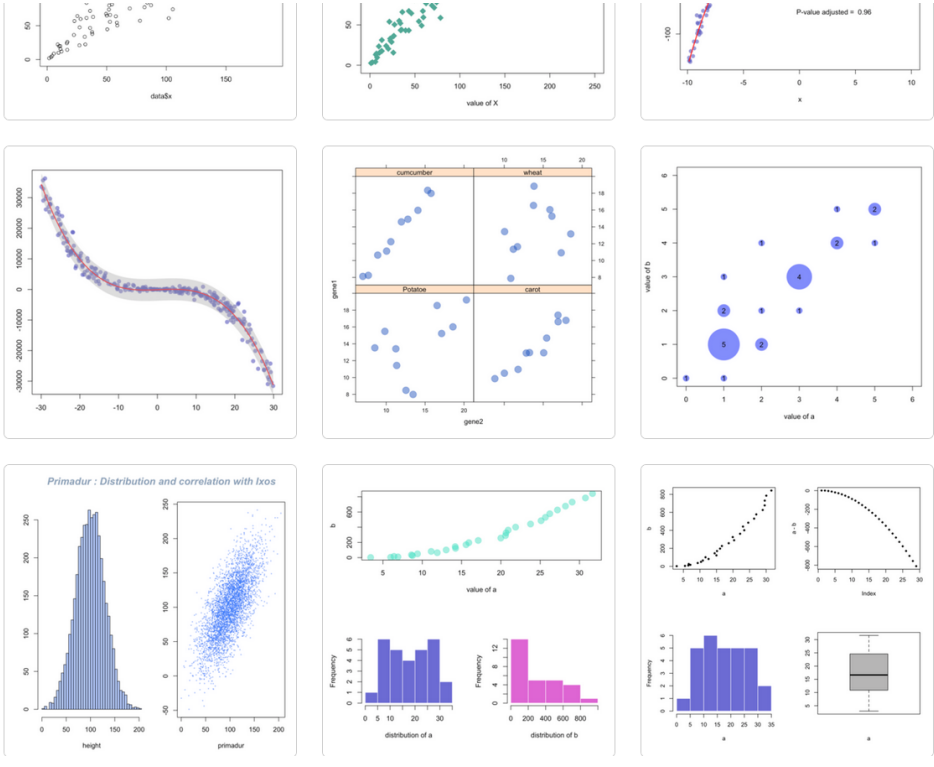



Data wrangling cheatsheet example : https://rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf
Cheatsheet page of RStudio : https://rstudio.com/resources/cheatsheets/

# Use other people’ s solutions

- Most coding problems have been experienced before by someone.
- There is a good chance that someone has posted the solution on the internet.
  - ideally : R package
  - https://stackoverflow.com/
  - http://www.sthda.com/english/
  - (https://stats.idre.ucla.edu/r/dae/robust-regression/

    https://www.statmethods.net/stats/correlations.html)
  - https://www.r-graph-gallery.com/
- ask a more experienced person



# Example : R Packages as solutions to common problems

- Problem : Search for taxonomic names and resolve them
- Solution : use specialised R package : **`taxsize`**
  - e.g. correct misspelled species names : *Helianthos annus* to *Helianthus annuus*
  - e.g. many synonyms for a species :
  - 

```
c("Helianthus annuus ssp. jaegeri",
"Helianthus annuus ssp. lenticularis",
"Helianthus annuus ssp. texanus",
"Helianthus annuus var. lenticularis",
"Helianthus annuus var. macrocarpus",
"Helianthus annuus var. texanus")
```

→

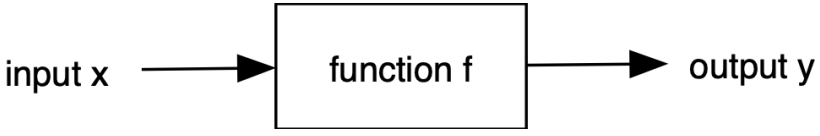
```
acceptedName
"Helianthus annuus"
```

- Alternative solution (e.g. similar problem, but not related to taxonomic names)
- Use a more general R package : e.g. **`stringr`**
  - e.g. normalise synonyms in questionnaire answers :
  - "`increased`" and "`improved`" → "`positive`"



UNIVERSITÄT BERN

# Use other people’ s solutions with care



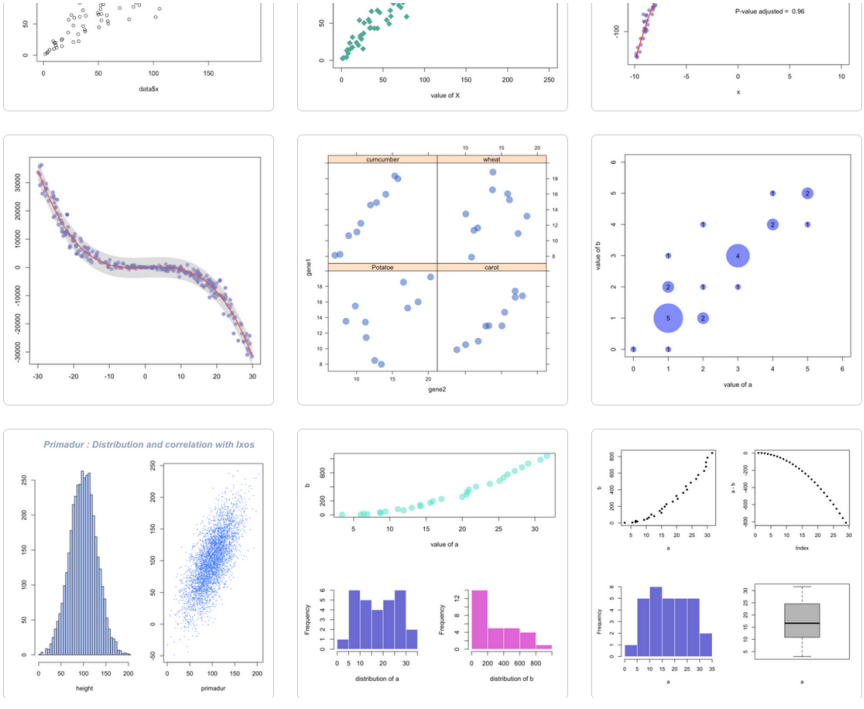

- If you use someone elses solution:
  - make sure that you try it out (e.g. with an easy example)
  - carefully chek if it does what you want it to do.
- Example :
  - View(), head() and str() , table()
  - expected number of : rows, columns,
  - column/ rowwise : expected number of NAs,

    expected range of values, names still correct?
  - what happens if you give a cornercase input:

    e.g. a reduced/ empty dataset as input,

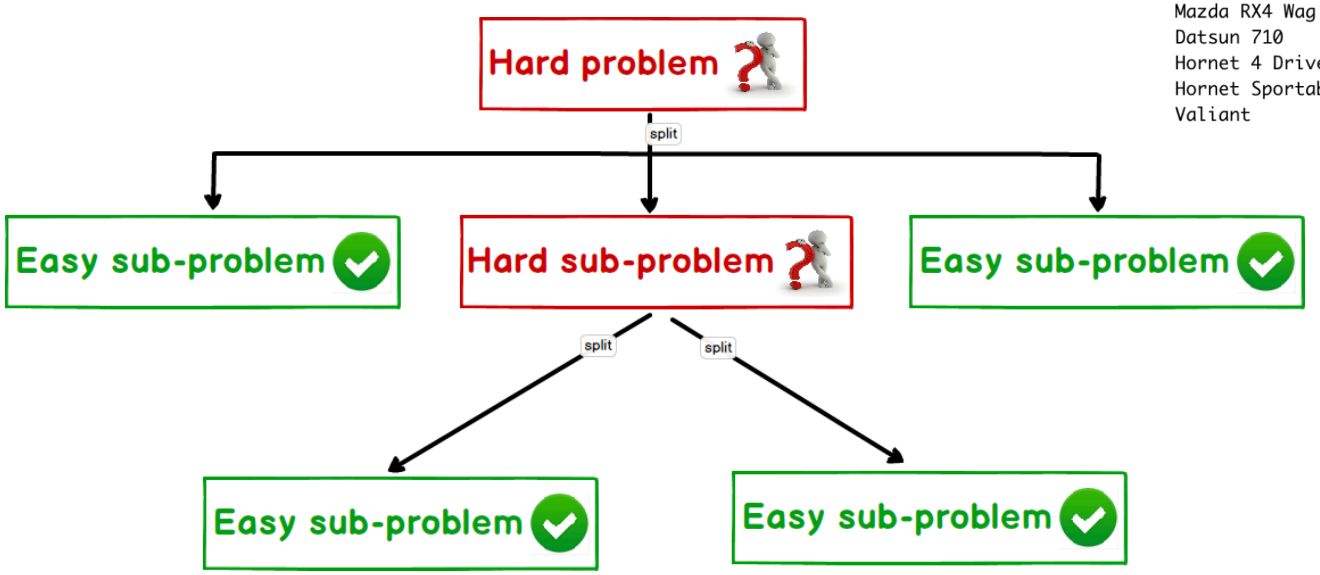
    an invalid argument,

    a very large/ negative number, zero



# 6. Plan your program

- Try out – Think over – Repeat
- Write down the problem you have and the steps you need to do in order to solve it.
- Programming means breaking down the problem into small sub-problems and addressing each sub-problem separately.
  - sub-divide your dataset




| | mpg | cyl | disp | hp | drat | wt | qsec | vs | am | gear | carb |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Mazda RX4 | 21.0 | 6 | 160.0 | 110 | 3.90 | 2.620 | 16.46 | 0 | 1 | 4 | 4 |
| Mazda RX4 Wag | 21.0 | 6 | 160.0 | 110 | 3.90 | 2.875 | 17.02 | 0 | 1 | 4 | 4 |
| Datsun 710 | 22.8 | 4 | 108.0 | 93 | 3.85 | 2.320 | 18.61 | 1 | 1 | 4 | 1 |
| Hornet 4 Drive | 21.4 | 6 | 258.0 | 110 | 3.08 | 3.215 | 19.44 | 1 | 0 | 3 | 1 |
| Hornet Sportabout | 18.7 | 8 | 360.0 | 175 | 3.15 | 3.440 | 17.02 | 0 | 0 | 3 | 2 |
| Valiant | 18.1 | 6 | 225.0 | 105 | 2.76 | 3.460 | 20.22 | 1 | 0 | 3 | 1 |

| | mpg | cyl | disp | hp |
|---|---|---|---|---|
| Mazda RX4 | 21.0 | 6 | 160.0 | 110 |
| Mazda RX4 Wag | 21.0 | 6 | 160.0 | 110 |
| Datsun 710 | 22.8 | 4 | 108.0 | 93 |

# 6. Plan your program

```
###
# EXAMPLE 6 - PLAN YOUR PROGRAM
#
# Describe your problem and the steps you need to do to solve it.
# Example problem : Plot a normal distribution with mean = 0 and sd = 1.
# Example planning :
#     - generate a normally distributed dataset with mean = 0 and sd = 1
#         - search internet which function can generate normally distributed data
#         - look up help of this function in R
#         - try out function
#         - make vector x with normally distributed data
#     - make the plot
#         - search internet which function to use for plotting
#         - make title and axis labels

help(rnorm)
rnorm(4, mean = 0, sd = 1)
x <- rnorm(40, mean = 0, sd = 1) # first parameter : length of vector
plot(x)
# plot() just shows the data.points. Maybe a Histogram would fit better?
#TODO plot histogram.
```

https://github.com/biodiversity-exploratories-synthesis/useful_functions/blob/main/2021_synthesis_Rhelpdesk_good_programming_practices.R

# 6. Plan your program

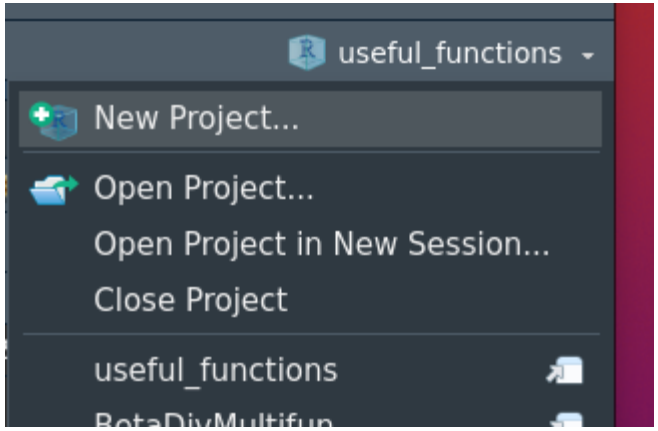
```
###
# EXAMPLE 6 - PLAN YOUR PROGRAM
#
# Describe your problem and the steps you need to do to solve it.
# Example problem : Plot a normal distribution with mean = 0 and sd = 1.
# Example planning :
#     - generate a normally distributed dataset with mean = 0 and sd = 1
#         - search internet which function can generate normally distributed data
#         - look up help of this function in R
#         - try out function
#         - make vector x with normally distributed data
#     - make the plot
#         - search internet which function to use for plotting
#         - make title and axis labels

help(rnorm)
rnorm(4, mean = 0, sd = 1)
x = rnorm(40, mean = 0, sd = 1) # first parameter : length of vector
plot(x)
# plot() just shows the data.points. Maybe a Histogram would fit better?
#TODO plot histogram.
```

https://github.com/biodiversity-exploratories-synthesis/useful_functions/blob/main/2021_synthesis_Rhelpdesk_good_programming_practices.R

# use RStudio projects

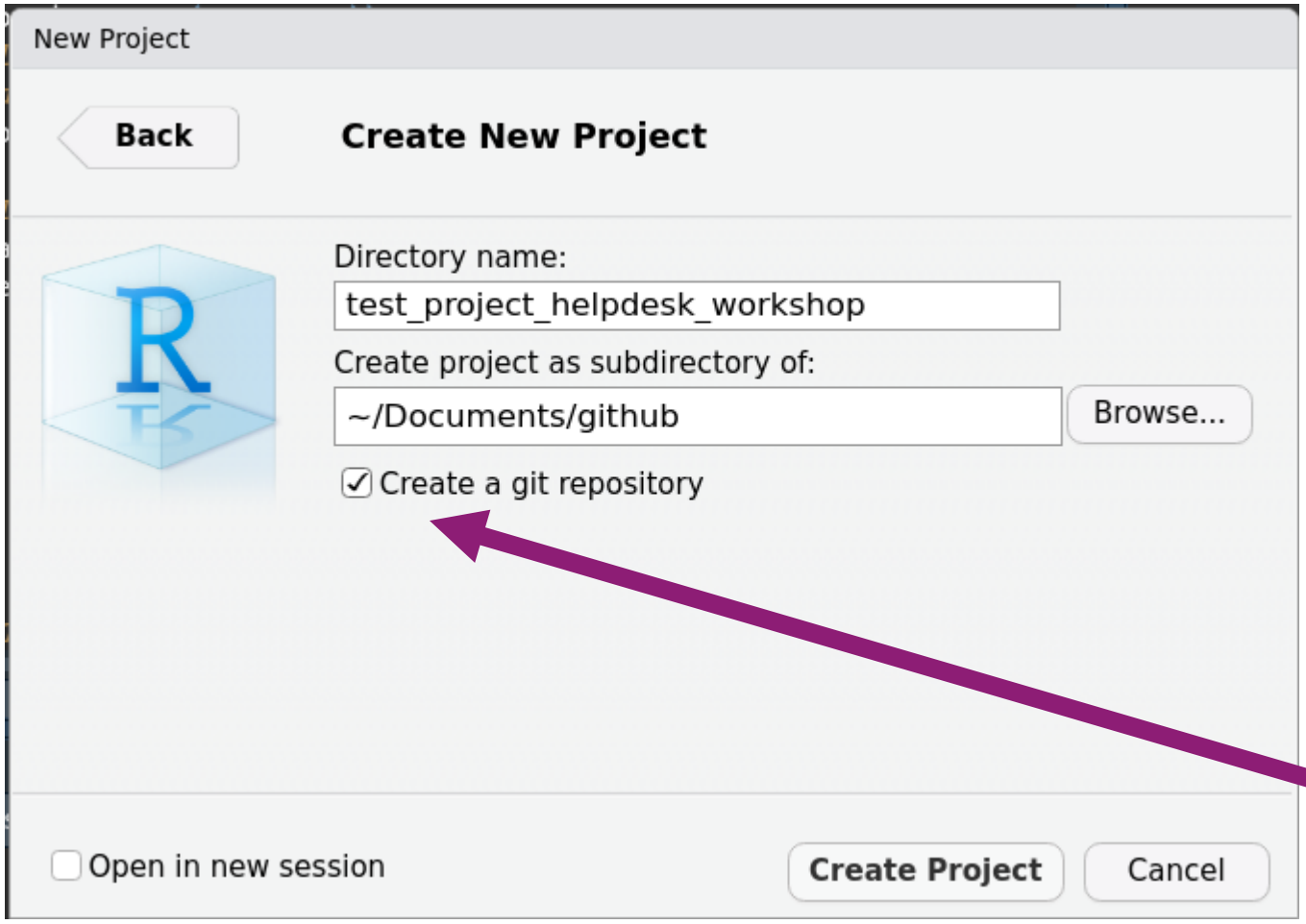
- divide your work into contexts
  - each with it's own working directory
  - workspace (opened documents)
  - history
- Create new project:
  - in a brand new directory
  - in an existing directory
  - by cloning a version control (e.g. git, see later)




Tutorial creating RStudio project : https://support.rstudio.com/hc/en-us/articles/200526207-Using-Projects

# use RStudio projects

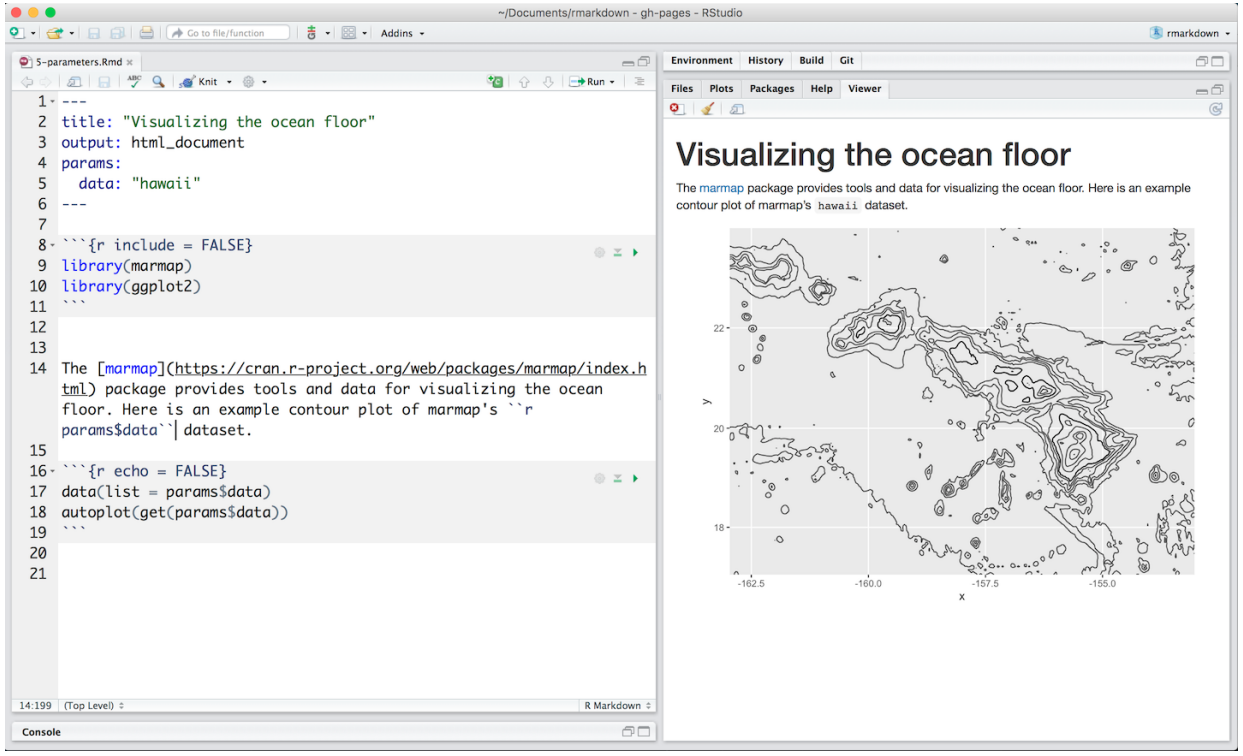
- create a new RStudio project




**will be important for version control**

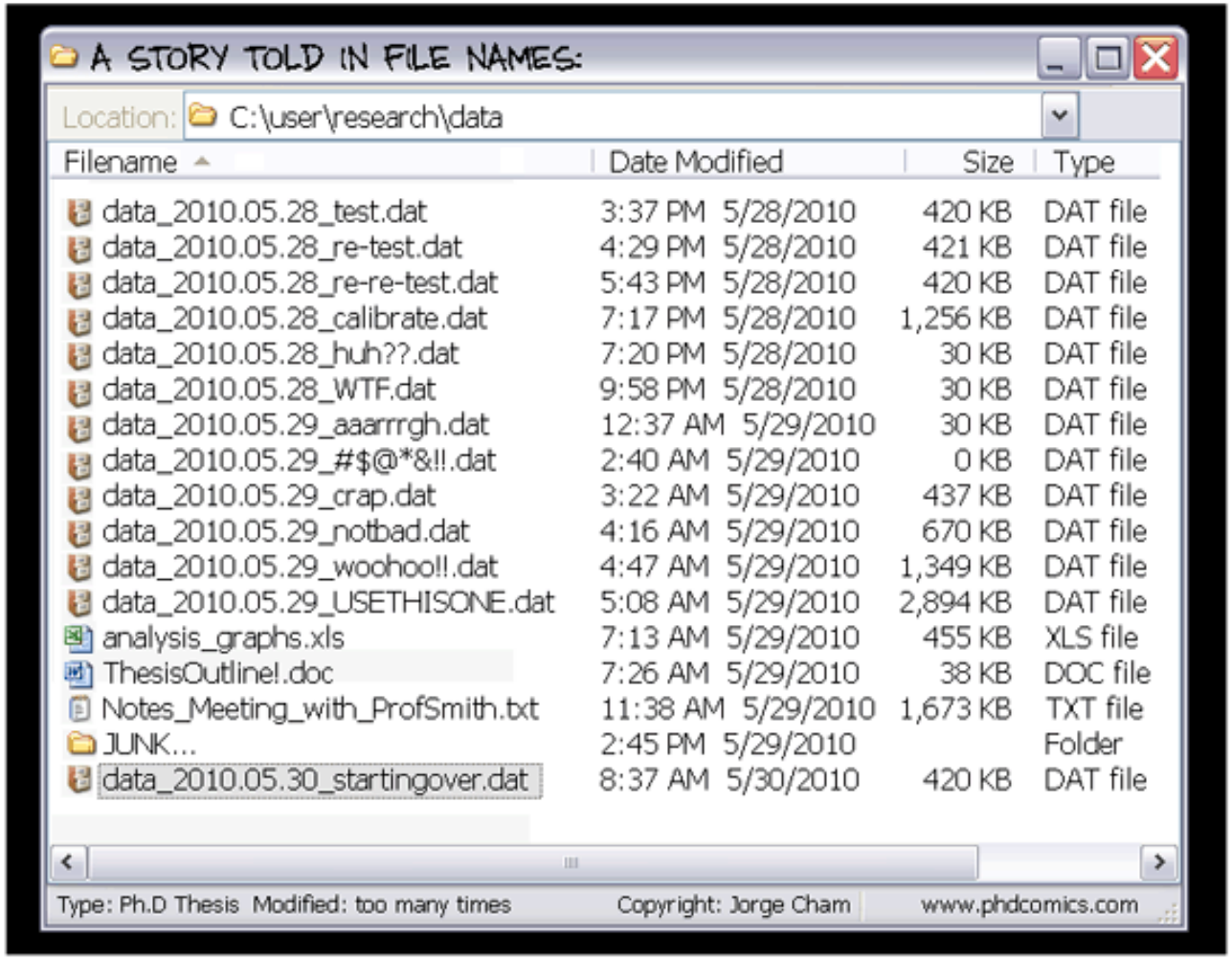
# Rmarkdown

- combining R script with documentation
- Suitable to document workflow
  - combine analysis and documentation
    https://rmarkdown.rstudio.com/lesson-1.html
- Distill package for scientific appearance : https://rstudio.github.io/distill/

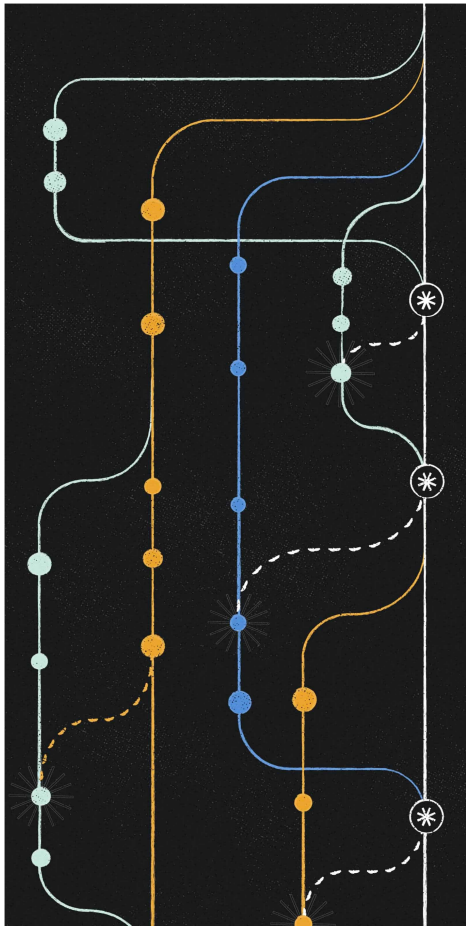


- around your script
- version control

# purposes of version control

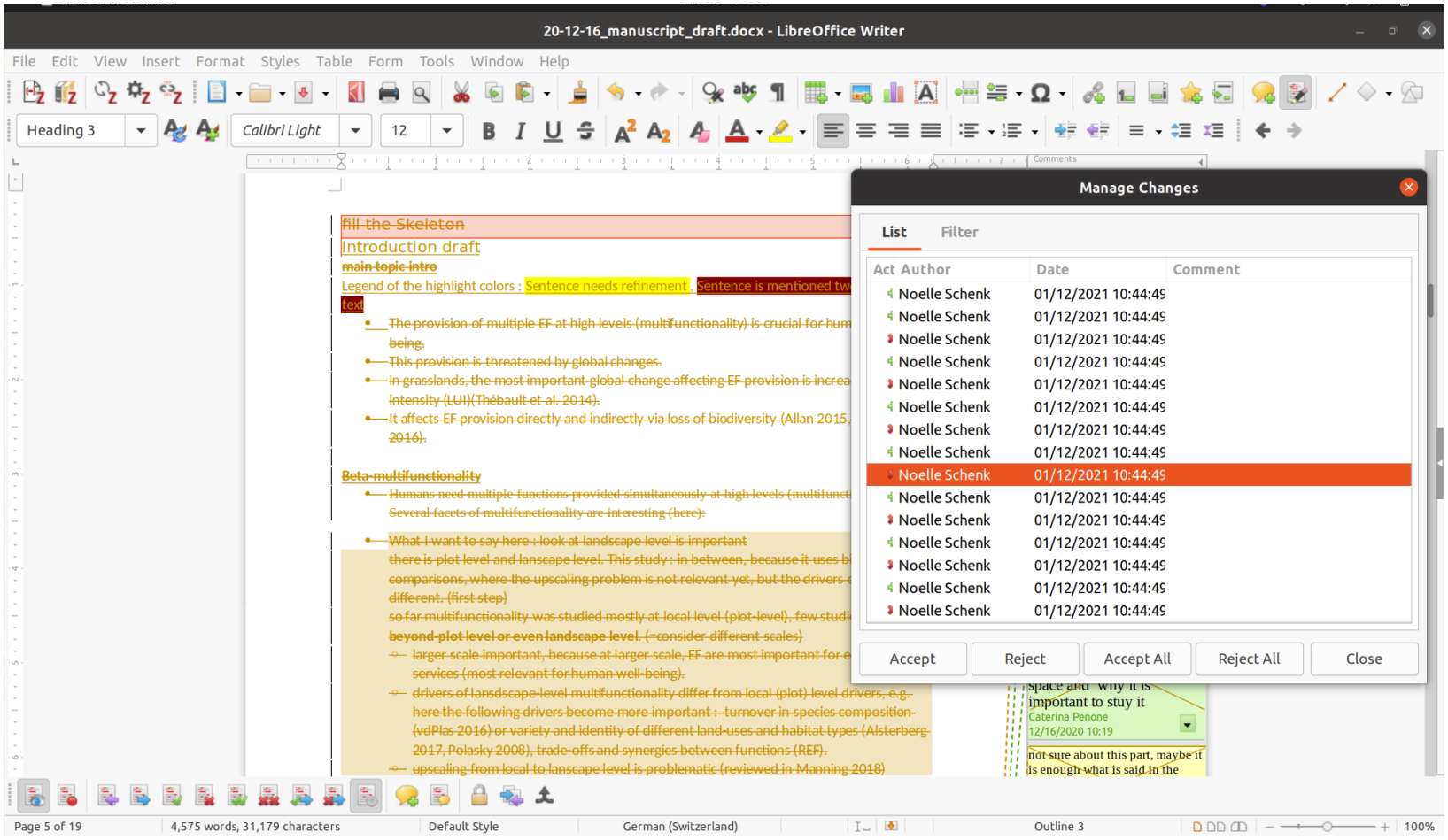


# purposes of version control



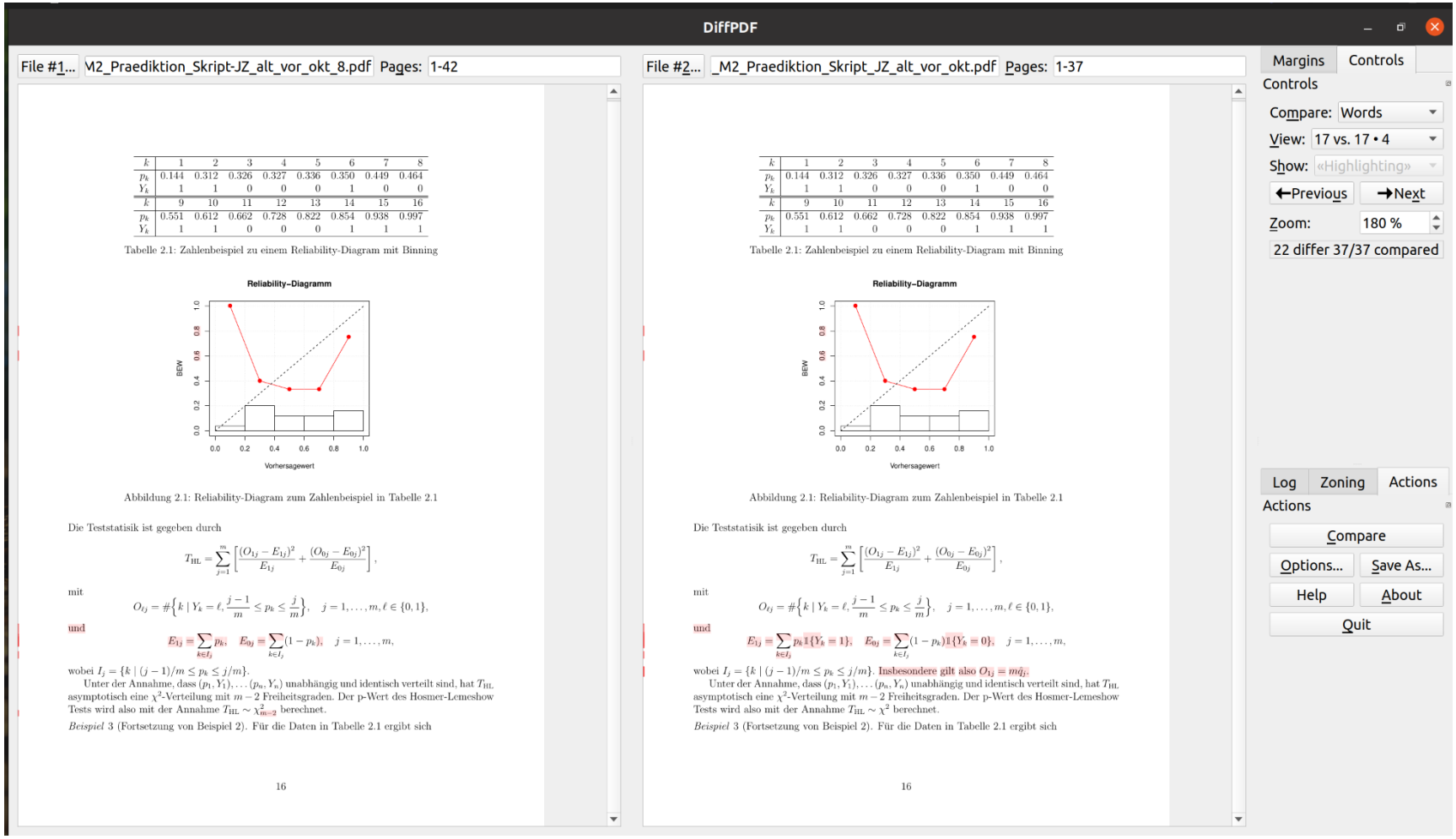
- keep track of changes
- keep scripts clear and structured
- understand previous changes
- compare different versions
- facilitate collaboration
- short- and long-term undo

# what is version control?




- Track Changes is a “one time version control”
- Edit > Track Changes > Compare document
- No commenting, just two documents

# what is version control?




- Comparing two documents (e.g. pdfs) creates a “diff” file
- Shows what’s different between files
- Version control for code : saves only first version and all following “diffs”

# Version Control for code

git

SUBVERSION

**Etymology 1** [ edit ]

From Middle English *get* ("[illegitimate] offspring"). A southern variant of Scots *get* ("illegitimate child, brat"), related to *beget*.[1]

**Noun** [ edit ]

**git** (*plural* **gits**)

1. (*Britain*, *slang*, *derogatory*) A silly, incompetent, stupid, annoying or childish person (usually a man). [quotations ▼]

**Usage notes** [ edit ]

- *Git* is usually used as an insult, more severe than *twit* but less severe than a true profanity like *wanker* or *arsehole*, and may often be used affectionately between friends. *Get* can also be used, with a subtle change of meaning. "You cheeky get!" is slightly less harsh than "You cheeky git!".
- *Git* is frequently used in conjunction with another word to achieve a more specific meaning. For instance a "smarmy git" refers to a person of a slimy, ingratiating disposition; a "jammy git" would be a person with undeserved luck. The phrase "grumpy old git", denoting a cantankerous old man, is used with particular frequency.
- In parts of northern England, Northern Ireland and Scotland, *get* is still used in preference to *git*. In the Republic of Ireland, *get*, rather than *git* is used.
- The word has been ruled by the Speaker of the House of Commons to be unparliamentary language.[2][3]

**Translations** [ edit ]

**± A silly, incompetent, stupid, annoying or childish person** [show ▼]



UNIVERSITÄT BERN

# Version Control for code

```
GIT - the stupid content tracker

"git" can mean anything, depending on your mood.

 - random three-letter combination that is pronounceable, and not
   actually used by any common UNIX command.  The fact that it is a
   mispronunciation of "get" may or may not be relevant.
 - stupid. contemptible and despicable. simple. Take your pick from the
   dictionary of slang.
 - "global information tracker": you're in a good mood, and it actually
   works for you. Angels sing, and a light suddenly fills the room.
 - "goddamn idiotic truckload of sh*t": when it breaks

This is a stupid (but extremely fast) directory content manager.  It
doesn't do a whole lot, but what it _does_ do is track directory
contents efficiently.
```

- initial git commit of README by Linus Torvalds, creator of git

# Version Control for code

git

SUBVERSION

**plot overviewbars of multiple models : added option for dodged barplo…**

Browse files

…ts for better overview

master

noschenk committed 19 days ago  1 parent 6eb43e4  commit aacc9b9df485ec99079b966b69e38c2773b8c5b1

```
271  273    #' @export
272  274    ### FUNCTION
273       - create_single_funs_overviewbar_plot <- function(singleF_restab, legend = F){
     275  + create_single_funs_overviewbar_plot <- function(singleF_restab, legend = F, pos = "stack"){
274  276      if(legend){
275  277      # return a plot with legend
276  278        print("not implemented yet.")
277  279      } else {
278  280        sf_ov <- ggplot(singleF_restab, aes(x = variable, y = value, fill = color)) +
279       -      geom_bar(stat = "identity", color = "black") +
     281  +      geom_bar(stat = "identity", color = "black", position = pos) +
280  282          scale_fill_identity("", labels = singleF_restab$ground,
281  283                              breaks = singleF_restab$color,  guide = "legend") +
282  284          # coord_flip() +
```

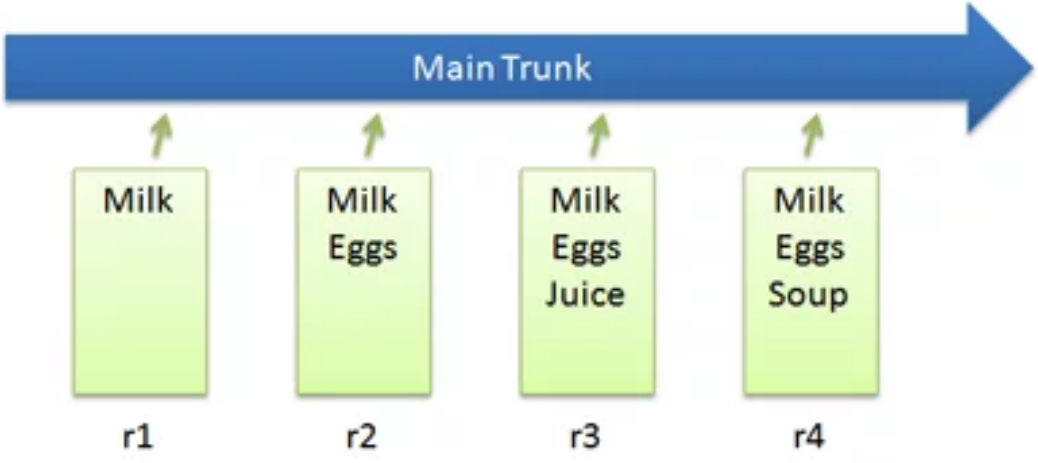
- Instead of saving versions of files, only save the “diff”
- Comment on every change
- Reconstruct any previous version as you need it



UNIVERSITÄT BERN

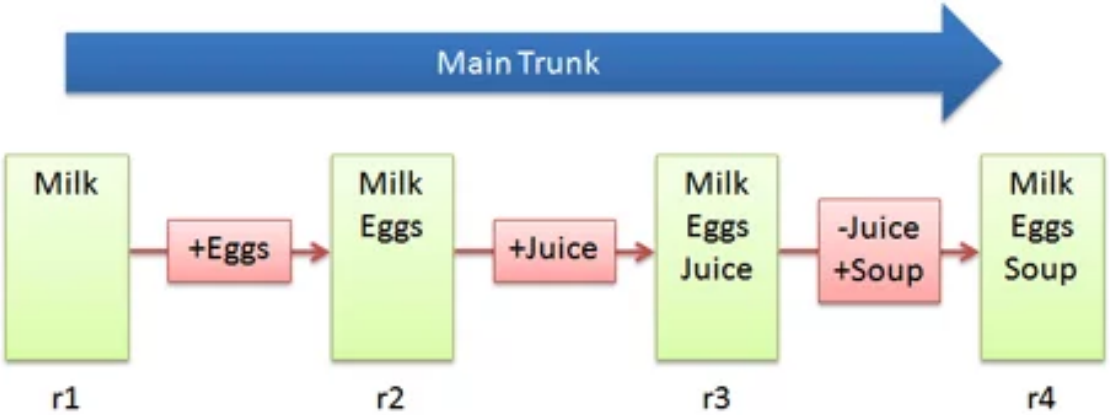
# what is version control?

- easiest case : the “project” just consists of one file which experiences changes :

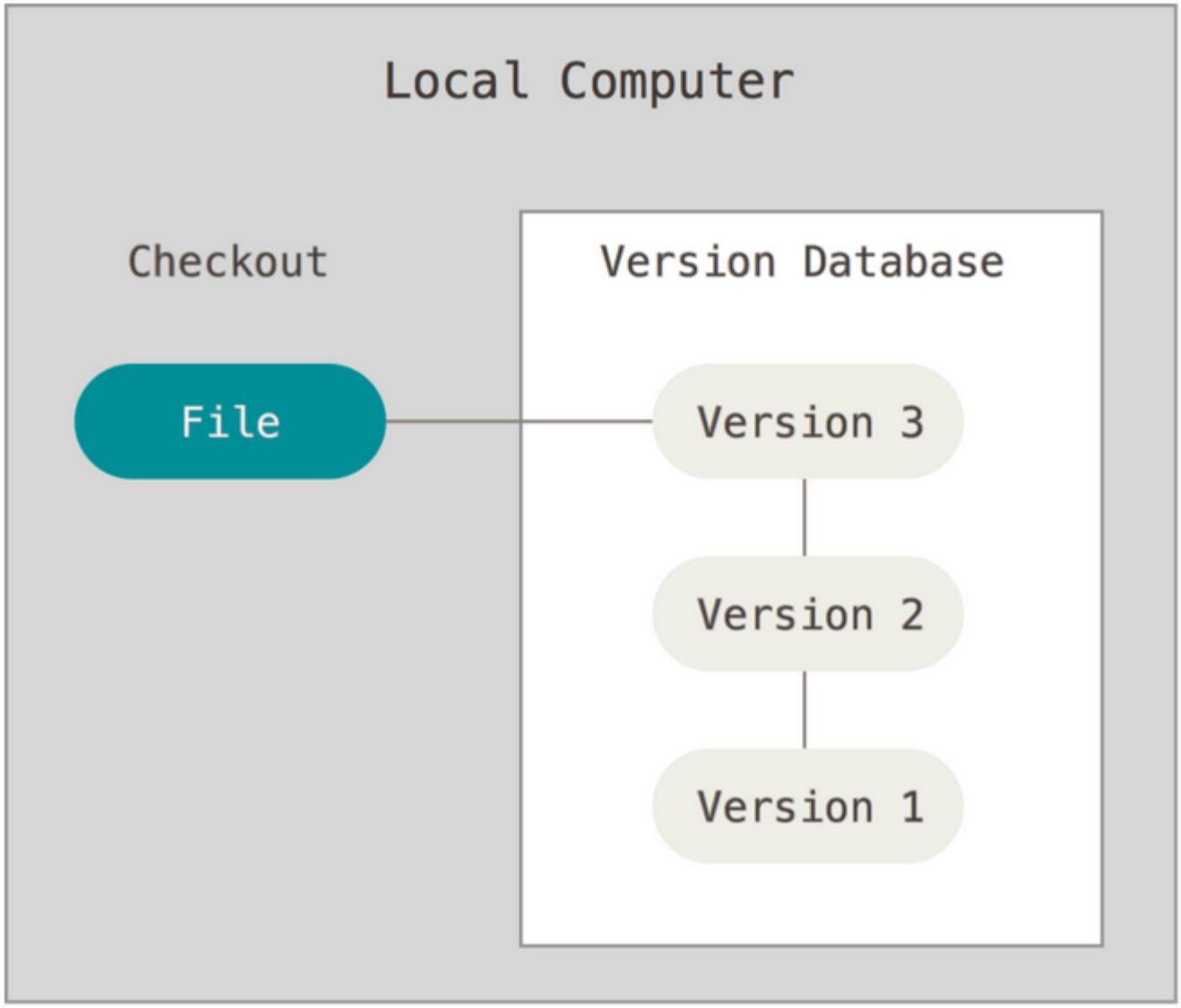


# what is version control?

- easiest case : the “project” just consists of one file which experiences changes
- diffs : Are a precise description of the changes made to a file.
- The changes to your file are stored as diff : this does not use a lot of storage and is easier to look at (you don’t have to search where exactly in your file were the changes)



# what is version control?



# version control in RStudio

- git

2021_synthesis_Rhelpdesk_good_...

Source on Save | Run | Source

```
85  ###
86  # NAMING THINGS
87  #
88  # good names reduce comments
89  60 <- time_tolerance # seconds
90  60 <- time_tolerance_in_seconds
91
92
93  ###
94  # READ ERROR MESSAGES
95  #
96  # read the error messages
97  x <- as.factor("A", "B", "C")
98  help("as.factor")
99  # the function encodes a vector, I gave the elements of a vector,
100 #    not the vector itself.
101 x <- as.factor(c("A", "B", "C"))
102
```

92:1 (Untitled) R Script

Console | Terminal | Jobs

~/Documents/github/useful_functions/

```
> # READ ERROR MESSAGES
> #
> # read the error messages
> x <- as.factor("A", "B", "C")
Error in as.factor("A", "B", "C") : unused arguments ("B", "C")
>
> help("as.factor")
> help("as.factor")
> x <- as.factor(c("A", "B", "C"))
>
```

Environment | History | Connections | Git

Diff | Commit | main

| Staged | Status | Path |
|---|---|---|
| | ? ? | .gitignore |
| | M | 2021_synthesis_Rhelpdesk_good_...ogramming_practices.R |
| | ? ? | useful_functions.Rp... |

Files | Plots | Packages | Help | Viewer

R: Factors | Find in Topic

factor {base} R Documentation

## Factors

### Description

The function `factor` is used to encode a vector as a factor (the terms 'category' and 'enumerated type' are also used for factors). If argument `ordered` is TRUE, the factor levels are assumed to be ordered. For compatibility with S there is also a function `ordered`.

`is.factor`, `is.ordered`, `as.factor` and `as.ordered` are the membership and coercion functions for these classes.

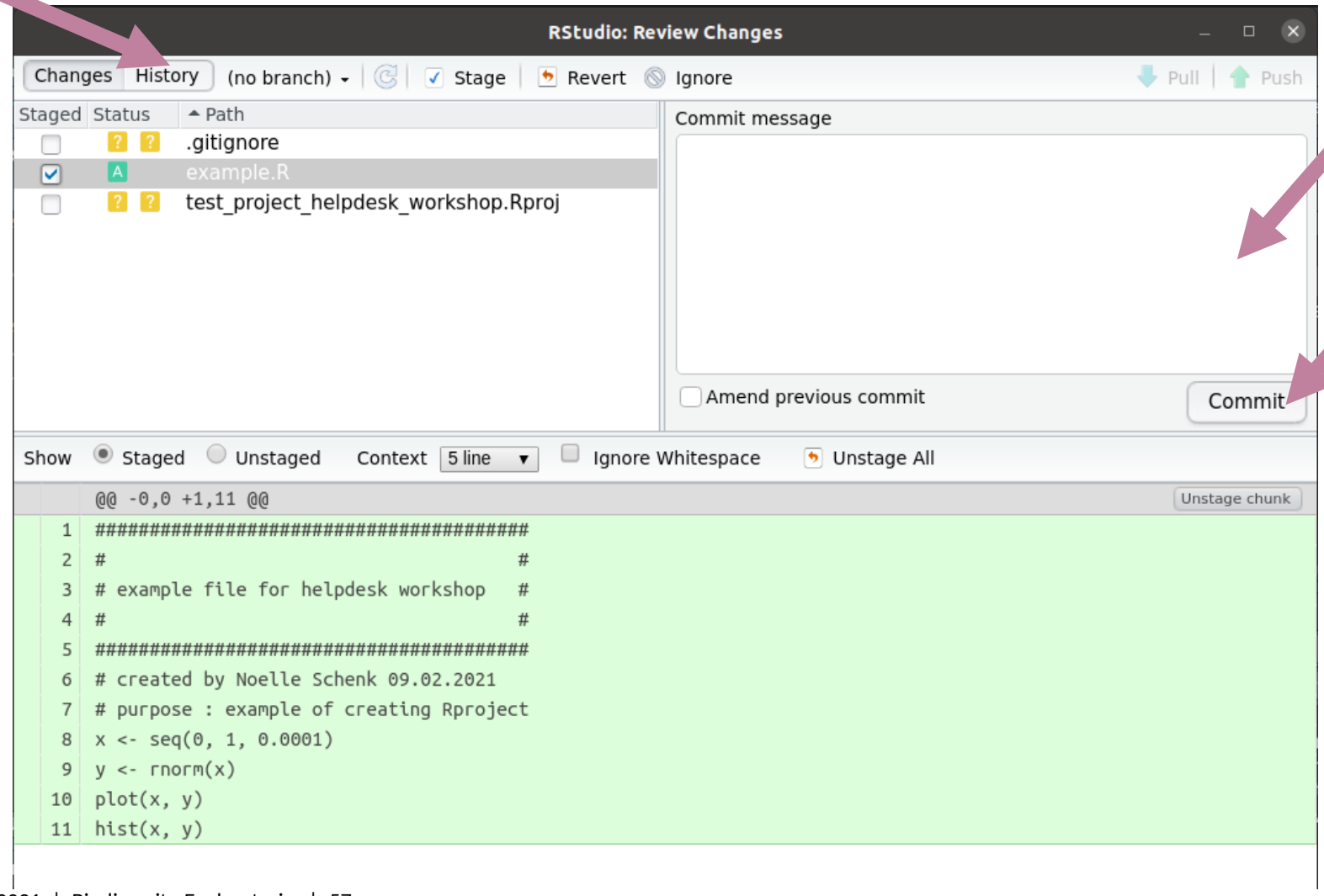
### Usage

```
factor(x = character(), levels, labels = levels,
       exclude = NA, ordered = is.ordered(x), nmax =

ordered(x, ...)
```

# guiding through version control

- make a change, check the box, click commit, write commit message, commit, leave window and repeat



# version control in RStudio

- git
- manual to set up git and RStudio
  https://happygitwithr.com/big-picture.html
- requires your folder to be an Rproject
- looking at history : see the commits you have done in the past

# GitHub

- GitHub is an external storage (owned by
- Windows) where repositories can be stored remotel
  - repository = the directory with your scripts in it
  - remotely = not in your local computer, but at a server in another place
- A place where you can share code and collaborate
  - but not for code publishing (e.g. zenodo)!
- If you want RStudio to work with GitHub, you also need “Push” and “Pull”.
- synthesis Github repository : https://github.com/biodiversity-exploratories-synthesis

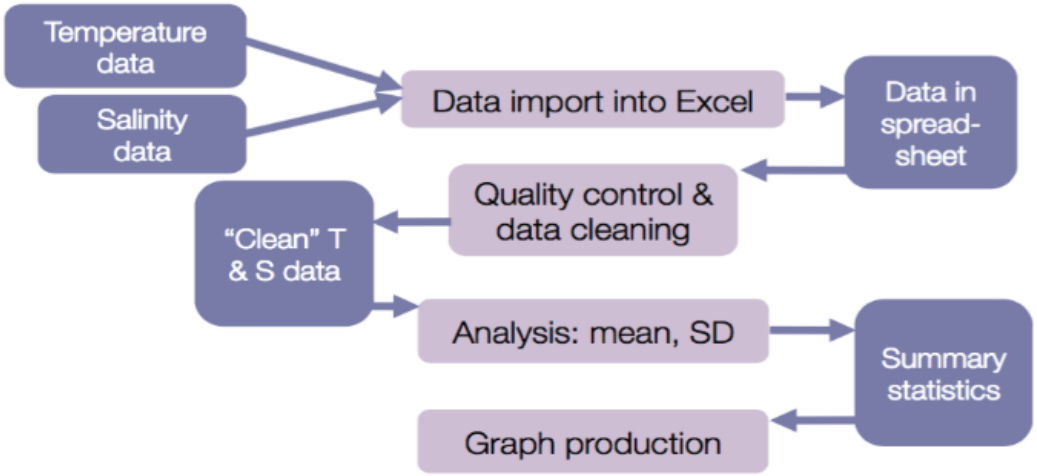
- **around your desktop**

# Document your workflow

- “all science is a workflow, but we rarely document it and rarely can we reconstruct all parts of it” – *unknown person in the internet*

*Workflow*: how you get from the raw data to the final products of your research

Simple workflow: flow chart



# keep raw data untouched

- make changes with R
- (document any changes)
- create derived data from raw data
- record all steps to process data – data manipulation is an integral part of your analysis!
- store together with metadata

# create analysis-friendly data

- “tidy” data
- during ata collection : collection-friendly data
- prefer long-table formats over wide-tables
- each column : is a variable, units belong to variable  name
- each row : is an observaton (1 row per plot)
- header : 1 row with columnames
- additional info go to metadata
- names
- atomize

messy

| id | trt | work.T1 | home.T1 | work.T2 | home.T2 |
|---|---|---|---|---|---|
| 1 | treatment | 0.08513597 | 0.6158293 | 0.1135090 | 0.05190332 |
| 2 | control | 0.22543662 | 0.4296715 | 0.5959253 | 0.26417767 |
| 3 | treatment | 0.27453052 | 0.6516557 | 0.3580500 | 0.39879073 |
| 4 | control | 0.27230507 | 0.5677378 | 0.4288094 | 0.83613414 |

tidier

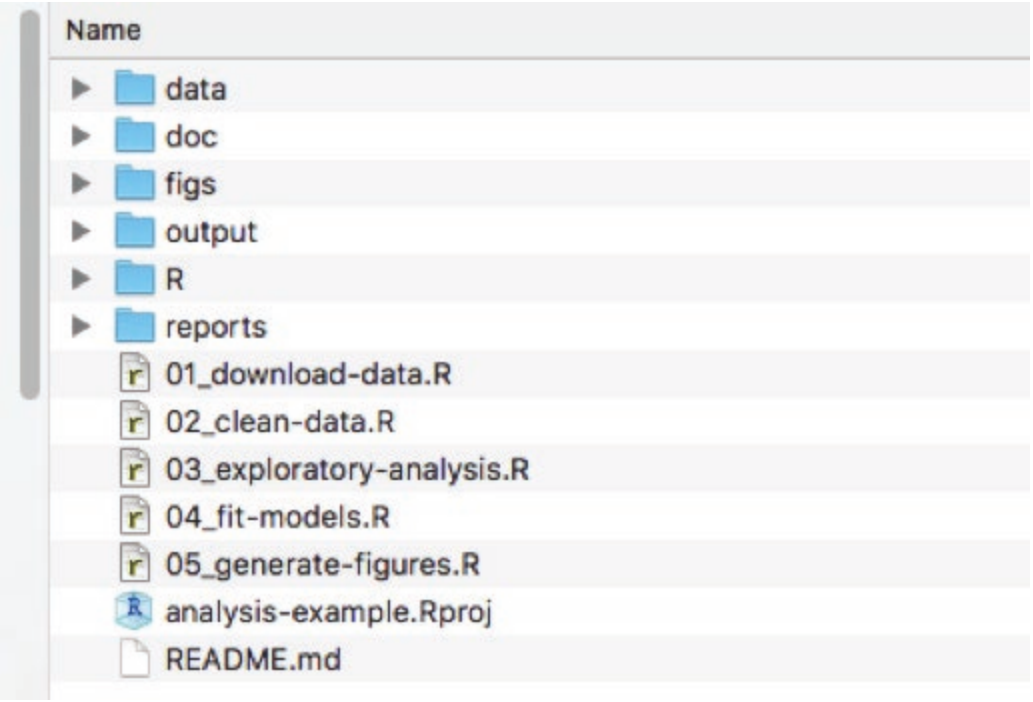
| id | trt | key | time |
|---|---|---|---|
| 1 | treatment | work.T1 | 0.08513597 |
| 2 | control | work.T1 | 0.22543662 |
| 3 | treatment | work.T1 | 0.27453052 |
| 4 | control | work.T1 | 0.27230507 |
| 1 | treatment | home.T1 | 0.61582931 |
| 2 | control | home.T1 | 0.42967153 |
| 3 | treatment | home.T1 | 0.65165567 |
| 4 | control | home.T1 | 0.56773775 |
| 1 | treatment | work.T2 | 0.11350898 |
| 2 | control | work.T2 | 0.59592531 |
| 3 | treatment | work.T2 | 0.35804998 |
| 4 | control | work.T2 | 0.42880942 |
| 1 | treatment | home.T2 | 0.05190332 |
| 2 | control | home.T2 | 0.26417767 |
| 3 | treatment | home.T2 | 0.39879073 |
| 4 | control | home.T2 | 0.83613414 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 5 | | | | | | | |
| 6 | Address | | Number | Street | City | State | Zip |
| 7 | 415 20th St, Oakland CA 94612 | VS | 415 | 20th St | Oakland | CA | 94612 |
| 8 | | | | | | | |

Hadley Wickham “R for Data Science” : https://r4ds.had.co.nz/tidy-data.html

# folder structure

- each project to its own directory
- /data : raw data and metadata.
  - intermediate data : to separate subfolder in /data OR to /results
- /doc : manuscripts, documentations for code, lab notebook, …
- /figs, /output or /results : statistical tables, figures and publication-ready figures (subdirectories)
- /R : R scripts
- README
- /old




BES reproducible code manual :
https://www.britishecologicalsociety.org/wp-content/uploads/2019/06/BES-Guide-Reproducible-Code-2019.pdf

# use open file formats

- working reproducible → be able to open files in 10 years
- .csv and .txt (not .xslx, this is proprietary)
  - comma or semi-colon (;) separation, point decimal separator
- GIF, JPEG and PNG for images
- encoding : ASCII or UTF

- Excel
  - can be used to open, edit and save .csv files
  - if your system language is german : uses semi-colon to separate columns and comma as decimal separator (4,3 instead of 4.3)
  - do not : merge cells, rely on formatting (e.g. coloured cells) or empty lines

*"There are only two hard things in Computer Science: cache invalidation and naming things."*

Phil Carton

# naming things

- file names :
  - meaningful
  - end in .R
  - no special characters but numbers, letters, - and _
- if files should be run in particular order, prefix them with numbers (and pad with zero).

```
# Good
fit_models.R
utility_functions.R

# Bad
fit models.R
foo.r
stuff.r
```

```
00_download.R
01_explore.R
...
09_model.R
10_visualize.R
```

# naming things

- use names to reveal intention
  - “area_rectangle” instead of “size”
- good names reduce comments

```
60 <- time_tolerance # seconds
60 <- time_tolerance_in_seconds
```

- capital letters for constants
  - BYTES_PER_GB = 1073741824
- use tab completion for names in RStudio
  - no excuse to not use long names
- use names that everyone can understand
  - avoid ambiguous names (e.g. “cat” = category or catalogue?)

# be consistent

- consistent naming
- either “index” or “idx”, not both
- consistent physical units (SI units)
- consistent style

# Take home

# Reflections

- Which of the practices will you apply?
- How much of the practices did you already know, how much were new?
- Which practices, you think, will help you the most in the future?
- Which practices seem less useful to you?

- Do you think the amount of information was too much, or too less, or just right?

# Take home

- Chose the practices that you think help you the most and focus on them
- practice version control
- keep the other practices in mind and address them after

# references and further reading

# references and resources

- the links provided in this presentation
- ETH Zürich Science IT course : "Best practices in Programming" https://sis.id.ethz.ch/services/consultingtraining/programming_best_practices.html
- Exploratories Synthesis page : https://github.com/biodiversity-exploratories-synthesis
- BES Guide for Reproducible Code : https://www.britishecologicalsociety.org/wp-content/uploads/2019/06/BES-Guide-Reproducible-Code-2019.pdf
- paper Wilson 2017 "Good enough practices in scientific computing" : https://doi.org/10.1371/journal.pcbi.1005510
- Florian Schneider resources , e.g. : basic R intro : http://fdschneider.de/r_basics/
- books of Hadley Wickham
  - R for Data Science : https://r4ds.had.co.nz/
  - Advanced R : https://adv-r.hadley.nz/
  - tidyverse : https://www.tidyverse.org/
- Introduction Course to Data Science by Brian Enquist : https://www.researchgate.net/publication/310798573_How_to_think_About_Your_Data_Introduction_to_Data_Science_Management_what_they_don%27t_-_but_should_-_teach_you_about_the_scientific_method



UNIVERSITÄT BERN

- **Exercises**

r4ds - main - RStudio
Mutating joins
Prices of 50,000 round cut diamonds by carat and cut
Quality of the cut
Ideal
Premium
Very Good
Good
Fair
Price in US dollars
Weight (carats)

# Exercises

try solving on your own and ask questions

- Git
- Unit testing
- generating a reproducible Example
- Divide coding problem into sub-problems
- Rmarkdown
- Renv
- **TODO**

- additional material

# why care? societal level

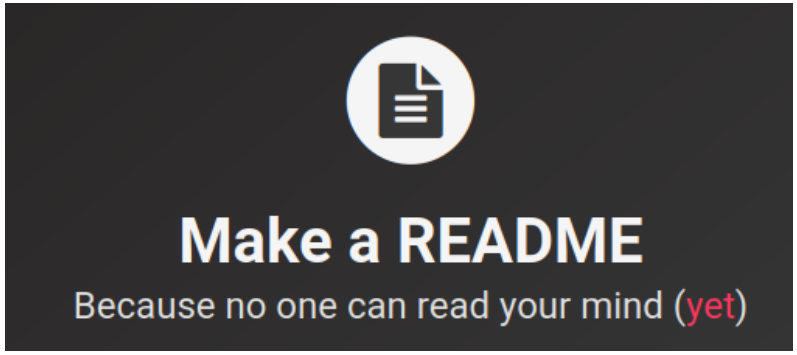
- Reproducibility
- FAIR principle

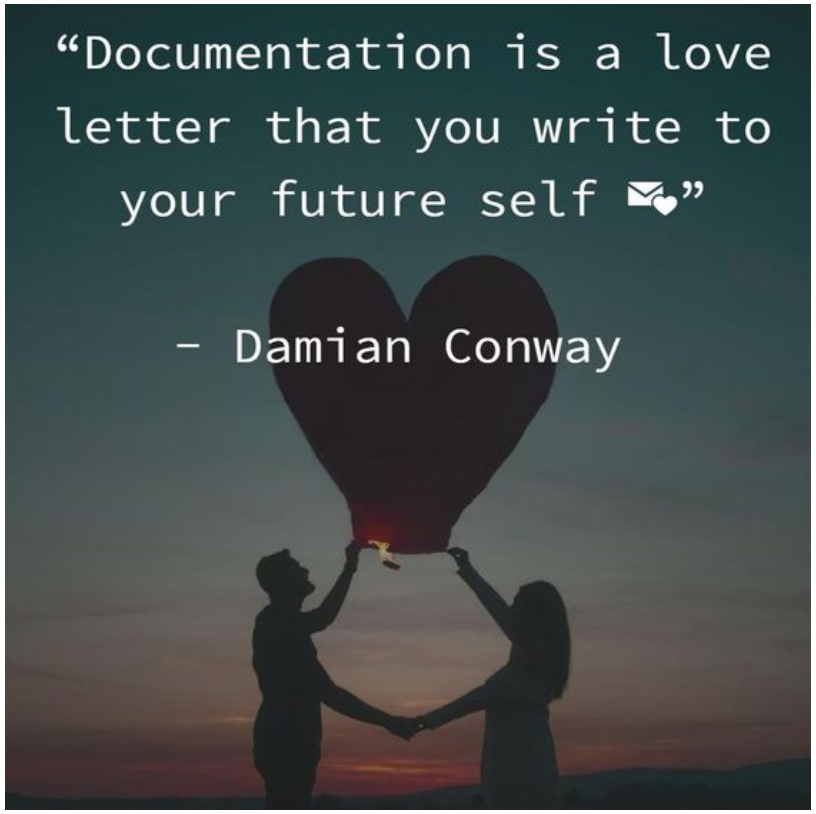
# How many R scripts should I write?

- Many!
- one script for one purpose
  - data cleaning
  - analysis
  - making figure
  - …
- → easier debugging, more flexible
- there are alternatives (debated)

# README

- describe the project (shortly) and your name and date
- provide basic orientation in the project folder
  - describe all column names and variables, define units
  - describe measurements
  - link relevant papers or data
- describe file naming scheme
- include answers to questions that a person trying to reproduce your project might have




Awesome README files: https://github.com/matiassingers/awesome-readme

# automated workflows

- what?
  - Code which can be run from input to output without you manually changing input or selecting subsections of the code.
- how ?
  - don’ t comment out parts of the script to adjust, but use if/else statements
  - use an executing script and a master script
  - if you need to do the same with several datasets : write code which is not dataset-specific, make a second script which runs the executing script for all datasets. Instead of having two scripts (D.R.Y.)

<DRY>
DRY
</DRY>
DON'T REPEAT YOURSELF

# Don’t repeat yourself

- try to write code only once
- re-use good pieces of your work
- reduce the risk of errors : you always have to update code in *all* places
- watch out for copy-pasting
- use functions

# functions

- should do what you mean
- no side effects
- describe parameters
- keep functions small
- few parameters
- return early
- for those who don’t know how to use functions yet : keep in mind to learn it

# reduce mental overhead

- in german "mentaler Aufwand"
- Programming alone is already demanding, try to relieve your brain with a good overview.
- causes of mental overhead :
  - related code is not together
  - a lot of scrolling needed
  - little empty lines
  - no sections or visually distinguishable parts
  - very long lines (scrolling from left to right)
  - nested code (part of the code here, bt is dependent on manually running other part in different file)

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


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

# unit testing (advanced)

- entangled software **breaks upon tiny changes** (e.g. new R version, package updates, bug fixes, ...)
- write integrated test functions which check if your code works right
- unit testing in R :
- e.g. testthat

# unit testing (advanced)

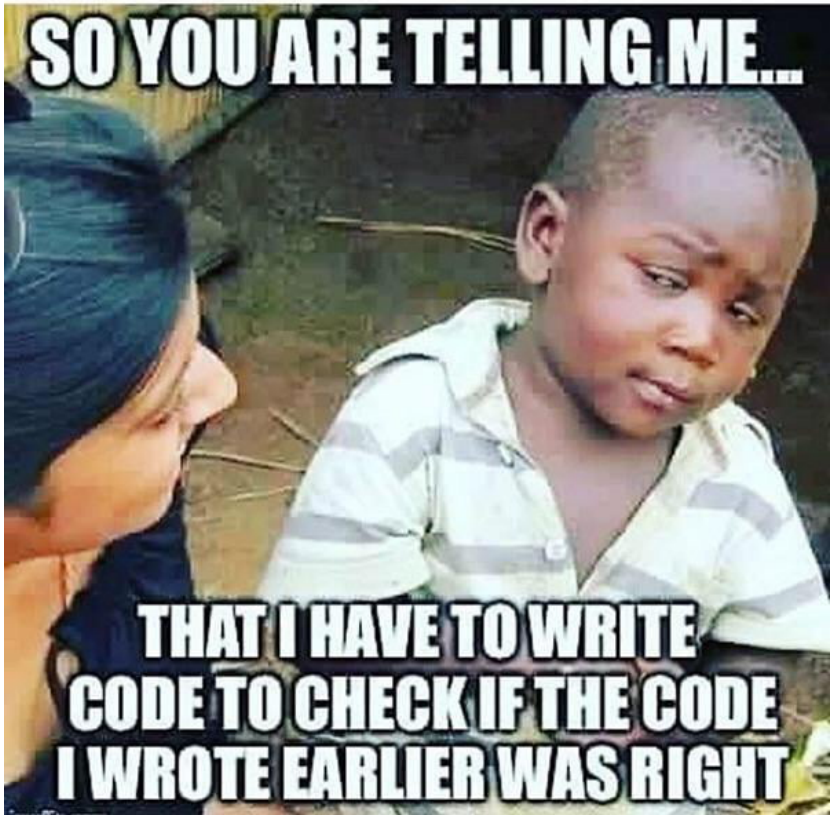
## The Art of Software Testing, Third Edition

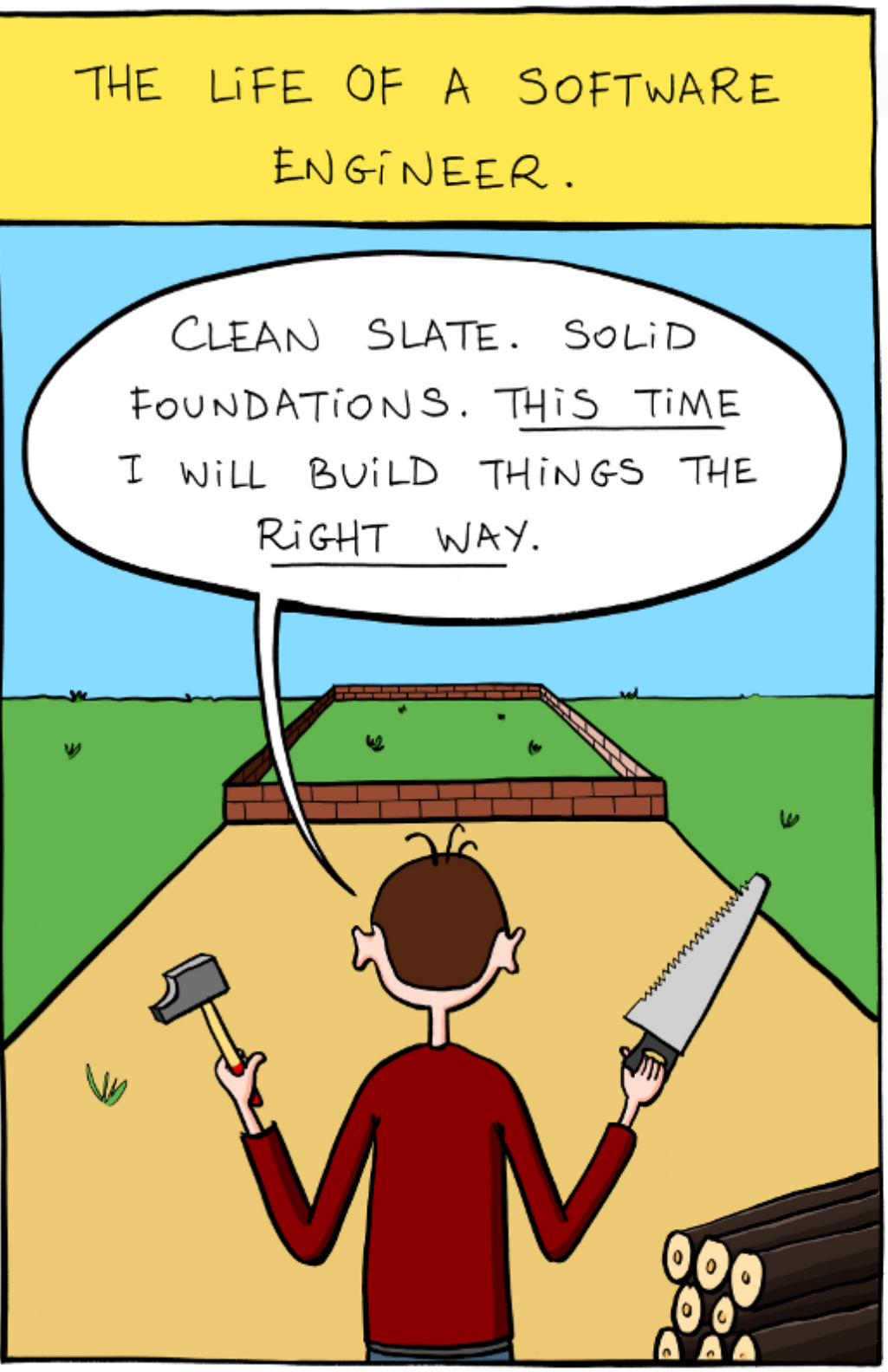
Editor(s): Glenford J. Myers, Tom Badgett, Corey Sandler

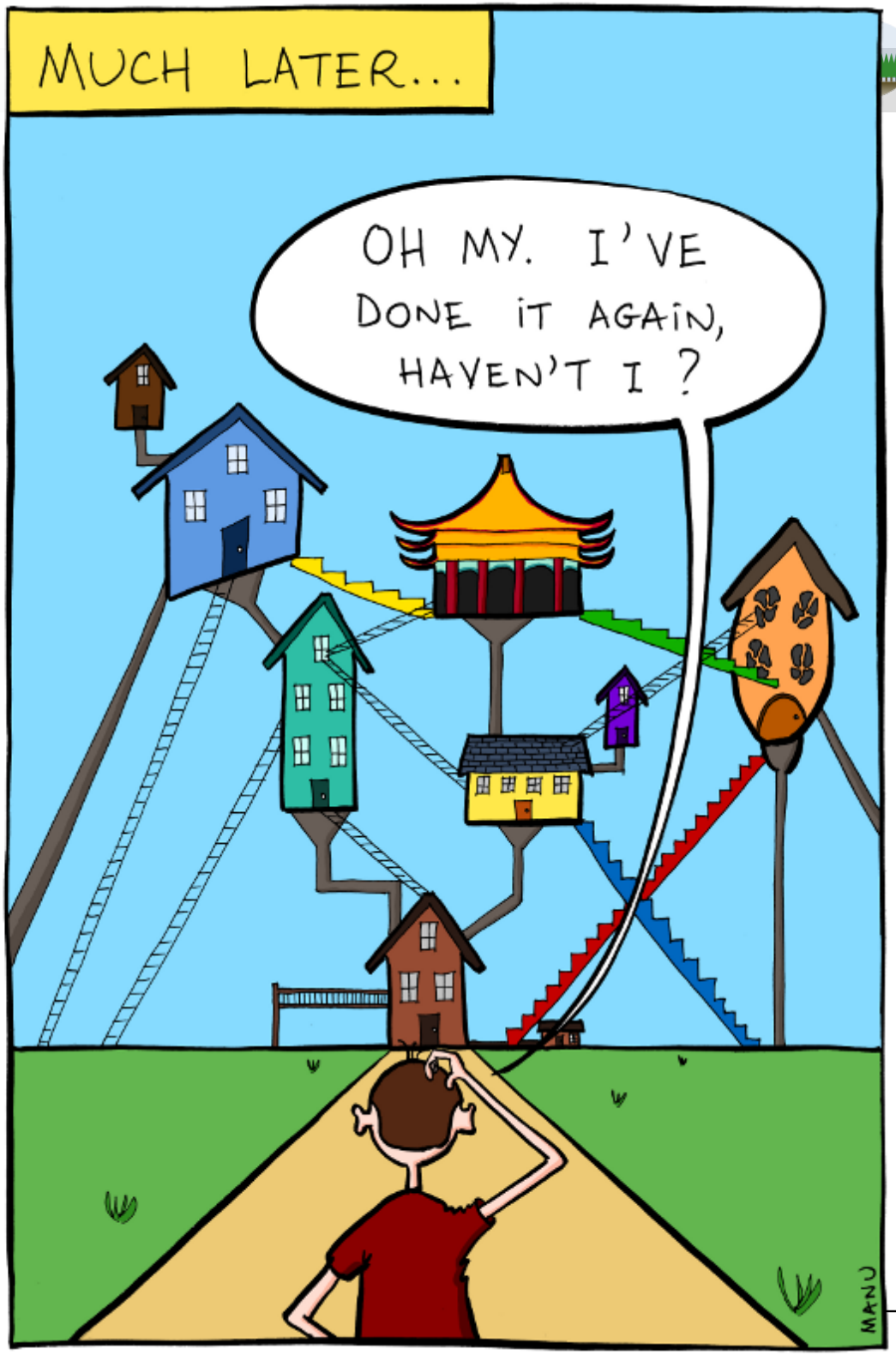
First published: 2 January 2012

Print ISBN: 9781118031964 | Online ISBN: 9781119202486 | DOI: 10.1002/9781119202486

Copyright © 2012 by Word Association, Inc. All rights reserved



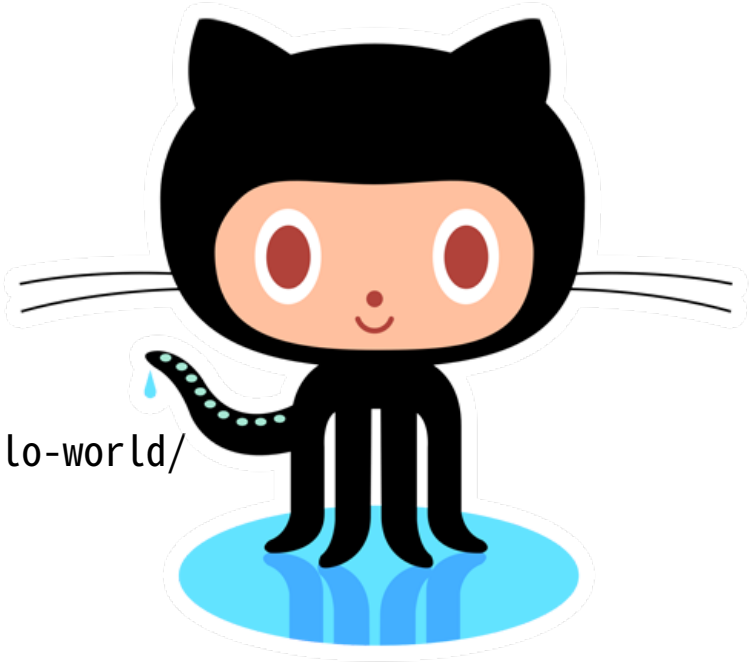

THE LIFE OF A SOFTWARE ENGINEER.
CLEAN SLATE. SOLID FOUNDATIONS. THIS TIME I WILL BUILD THINGS THE RIGHT WAY.


MUCH LATER...
OH MY. I'VE DONE IT AGAIN, HAVEN'T I ?
MANU

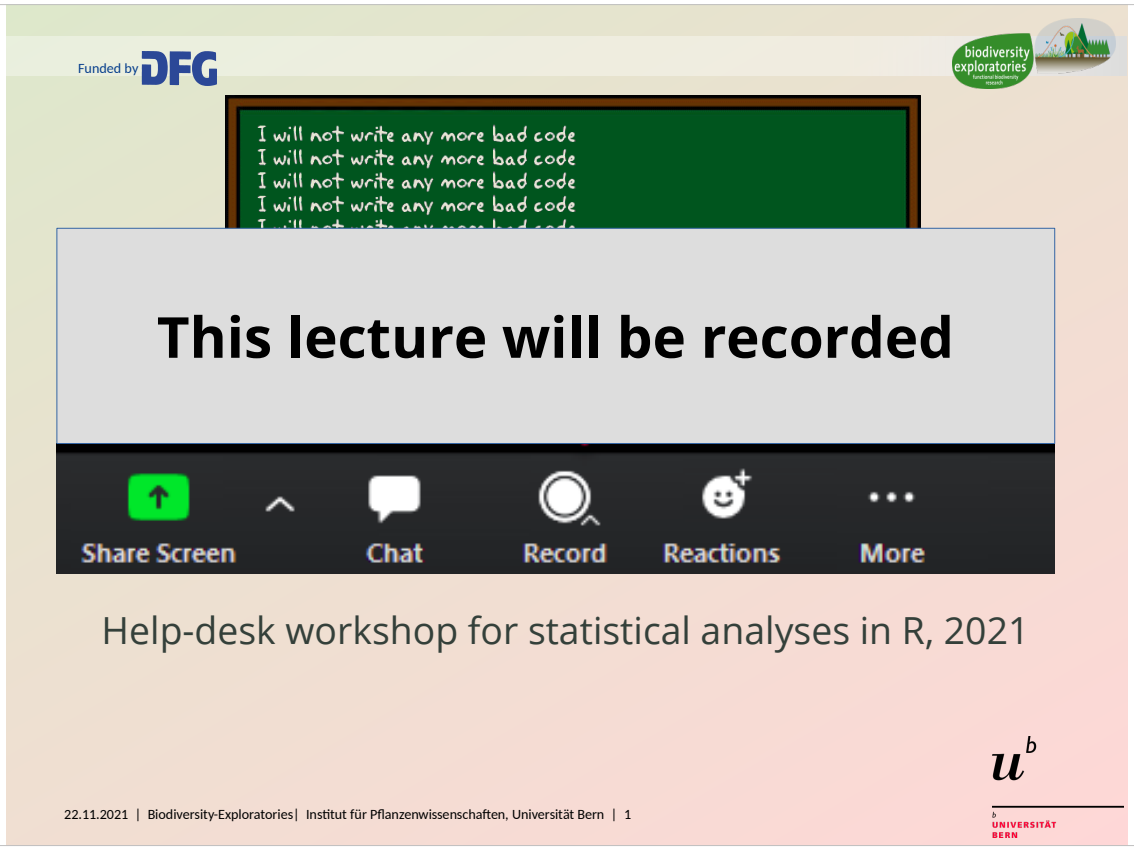
# code refractoring

- refractoring : going through your code and make it easier to understand without changing in- and output
  - difference to bug-fixing : there was no bug
- when should I refractor? all the time!
- things to address address :
  - cryptic names for variables: rename them
  - break long functions/ script bodies down to smaller units
  - replace algorithm if you learn a better one
  - combine duplicated pieces of code
  - add functions / loops after you learned them
  - TODO : make more advanced stuff slide, put this more basic
  - include new stuff, go over your code over again. once it works, its fine if it works, but if you want to improve you can do.

# GitHub and longtime storage



- GitHub : a possibility to store
- the versions of your file
- remotely (outside your computer)
- Guide to GitHub: https://guides.github.com/activities/hello-world/
- Connect with other programmers
- Share your code with the public
- GitHub doesn’ t offer longtime sotrage, if using it, publish finished code in other repository such as dryad or zenodo (has integration with github).




- This lecture will be recorded, do you agree?

# good practices in programming

Help-desk workshop for statistical analyses in R, 2021



- for IPS : helpdesk course from synthesis team : offer courses to all members of the Exploratories.mo
- goals
  - give inspiration, practices and tools in your hands
    - help programming
    - help others to understand your code
    - improve your coding skills
- programming is …
    - learn functions and how to use them
      - → like learning vocabulary for language
    - but also a whole world on its own with different rules
      - good to know
      - what other programmers find important
      - many manuals and recommendations
      - from ecologists and from software engineers

- diversity of levels
- → maybe even more than with programming itself
- because
- it is not compulsatory to know about good practices

- most suggestions will make you slower AT FIRST
    - pay off :
    - example : like prepare powerpoint:
      - prepare layout for each slide separately or use master slide formatter
      - both work
      - second option : need to learn something, but pays off if you do more than one presentation
    - your code : you will for sure run more than one time

- Give the word to you :
    - Does this seems familiar to one of you? Or do you have any ideas or thoughts that you could share?
    - which problems do you face?
    - why do you think this could be important for you?
    - what do you expect from this presentation?

Funded by DFG

# Main Sources of Content

Two-day workshop "Best practices in Programming"

ETH zürich

Scientific IT Services

SIB Swiss Institute of Bioinformatics

**BES guide** : https://www.britishecologicalsociety.org/wp-content/uploads/2019/06/BES-Guide-Reproducible-Code-2019.pdf?utm_source=web&utm_medium=web&utm_campaign=better_science

**Course** : https://www.sib.swiss/training/course/20210707_BPP



UNIVERSITÄT BERN

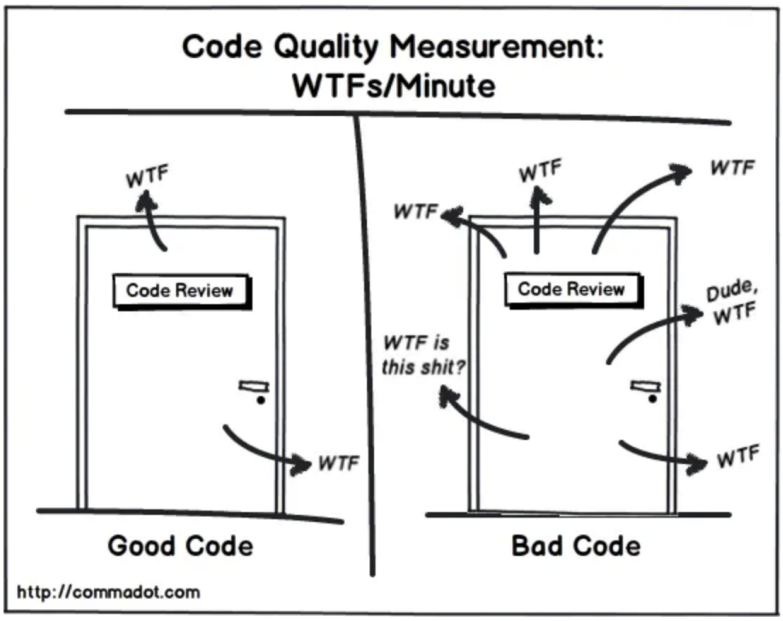
- of corse this is not my personal advice, or my personal way of coding,
- but a lot of experienced people
- mostly software engineers
- spent loads of time developing such good practices
- Scientists :
- mostly code alone
- share code at a late stage
- find out late if it was understandable
- (too late)
- Software engineers :
- mostly code in groups
- share code from the beginning
- you find out early if it is understandable
- → develop good practices

# Content

- why care?
- within your script
- around your scripts
  - version control
- around your project
- Take home/ Reflection
- additional material

what happens if your code is not very readable?
if it is hard to understand?

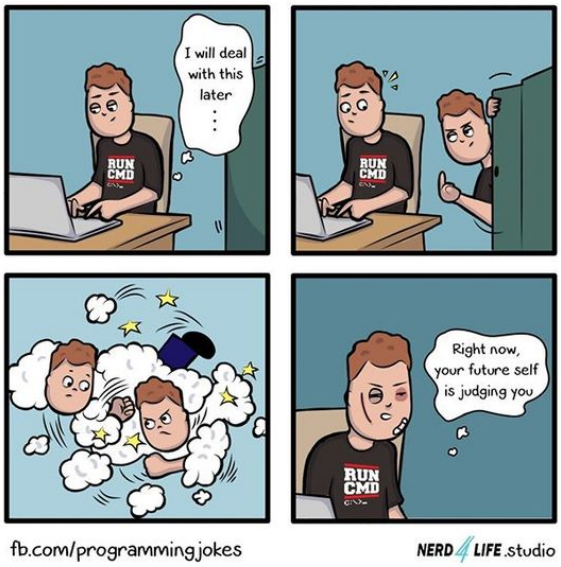
# why care? individual level






- I assume you are all familiar with the acronym wtf.... what the function
- one reason : improve quality of your code
    - code quality is hard to measure
        - people got creative
        - index of WTF/s per minute

# treat future-you well






- So this is probably very familiar to most of us :
  - sometimes we don’t care enough abou future-us.
  - sometimes future-us can get very angry on us
- It is always a good idea to treat future-you well
  - also in programming

## why care? individual level

- decrease future wtfs per minute factor
  - happier programming experience
- keep things easier for you and your collaborators
- facilitates error-detection
- enables sharing code

"Code is read more often than it is written"

- your scripts need to be understandable to you and to others
  - not perfect
- increase understandability
  - → increase happiness during programming
- Much easier to find errors
- If your code is very messy
  - low chances that you share it
  - code will be public → additional motivation
  - more and more funding agencies and journals require data to be published and accessible to the readers.

# why care? societal level

SORTEE

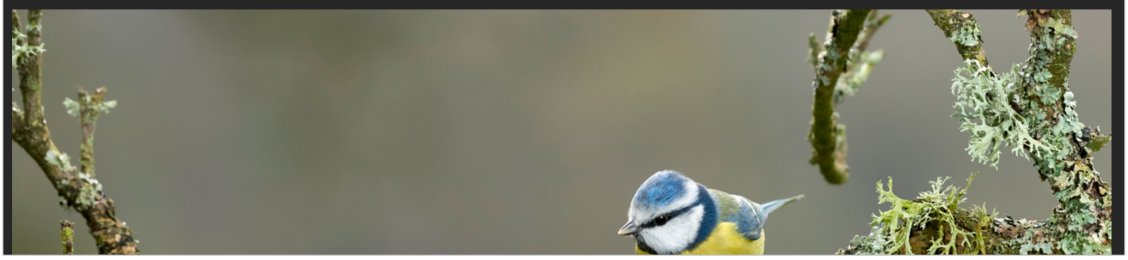
Society for Open, Reliable, and Transparent Ecology and Evolutionary Biology

NEWS | PLANTS & ANIMALS

**Psychology's replication crisis inspires ecologists to push for more reliable research**

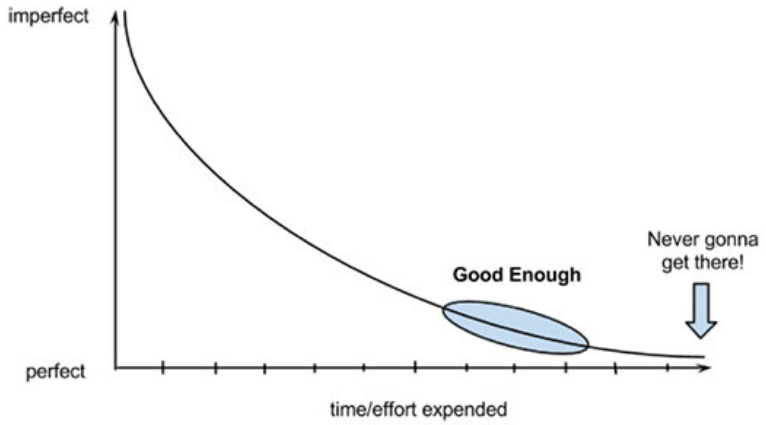
New society aims for transparency and culture change in ecology

9 DEC 2020 • BY CATHLEEN O'GRADY



- you probably heared about replication crisis in psychology
- e.g. a study : ½ to 1/3 of results are replicable
- is very likely to be a general problem in science
- also ecology
- Need to build trust in science
- Transparency and openness
- Science is funded by public → needs to be accessible AND understandable to public
- sharing code
- sharing UNDERSTANDABLE code
- crisis inspired development of  many practices :
- e.g. SORTEE Society
- most relevant today :
- (open science)
- reproducibility

# about perfection ...






- you don’t have to apply everything now and perfectly,
  - but find good balance of what pays off to you
  - what you think is important
  - → start with that.

- If you are software engineer : really apply them
  - for us : necessary to apply most of them
  - good to apply many
  - perfect to apply all = not possible

- today : a selection of the ones I find most important
- and easiest to start with
  - = the ones that appear in most recommendations
  - variety of other practices available

lets jump to the first scale and look at some practices that you can apply within your script.

# keep it simple!

```r
###
# EXAMPLE - KEEP IT SIMPLE!  ####
#
# bad example : all in one line
info_data <- data.table::fread(paste(pathtodata, "/data_assembly/helper_data/info_data.csv", sep = ""), header=T)
#
# good example : use many lines
info_data <- data.table::fread(
  paste(pathtodata,
        "/data_assembly/helper_data/info_data.csv",
        sep = ""),
  header=T)

# add structure to your file ####
# if you add four "#" in the end of the title, RStudio will allow
#   you to fold in and out the section
```

```r
###
# EXAMPLE - KEEP IT SIMPLE!
# add structure to your file
```

- use the lines
- use indenting
- add structure to your file

https://github.com/biodiversity-exploratories-synthesis/useful_functions/blob/main/2021_synthesis_Rhelpdesk_good_programming_practices.R



- with simple, I mean
  - simple for your brain
  - simple for your eyes
  - easier to grasp what the script is about

  - If you open a script, you don't get overwhelmed by it
  - but  you are able to hold on to a structure and
    - orientate yourself

- there is no limit to lines in a script
  - --> you are free to use many lines.
  - This helps the eye to grasp what is happening.

- use indent
  - = the little spaces before a line starts
- empty lines
- make something like sections
  - e.g. 4 hasthags # at the end of a line
  - → allow to fold in and out the section

# use a code style

- Good coding style is like correct punctuation: you can manage without it, **butitsuremakesthingseasiertoread**.
- e.g. the tidyverse style guide

```r
# Good
average <- mean(feet / 12 + inches, na.rm = TRUE)
# Bad
average<-mean(feet/12+inches,na.rm=TRUE)

# Good
do_something_very_complicated(
  something = "that",
  requires = many,
  arguments = "some of which may be long"
)
# Bad
do_something_very_complicated("that", requires, many, arguments,
                              "some of which may be long"
)
```

tidyverse style guide : https://style.tidyverse.org/index.html
short version : http://adv-r.had.co.nz/Style.html
R packages to help with code style : https://styler.r-lib.org/ and https://github.com/jimhester/lintr



- all style guides
  - are fundamentally opinionated
    - some decisions make code easier to use (especially matching indenting to programming structure)
    - but many decisions are arbitrary

- most important thing about a style guide
    - that it is consistent
  - makes code :
    - easier to read
    - easier to write
    - → you need to take fewer decisions

- R packages to help you :
- lintr performs automated checks to confirm that you conform to the style guide.
- styler interactively restiles selected texts.
- I personally don't use one of these packages

# use a code style

- Variable and function names with lowercase letters, separated by underscore "_"
  - Good : `day_one, day_1`
  - Bad : `DayOne, dayone`
- variable names should be nouns and function names should be verbs
  - `day_one`
  - `calculate_time_span`
- avoid re-using names of common functions and variables. This will cause confusion for the readers of your code
  - `Bad : T ← FALSE`
  - `Bad : mean ← function(x) sum(x)`
- more recommendation in the tidyverse styleguide

tidyverse style guide : https://style.tidyverse.org/index.html
short version : http://adv-r.had.co.nz/Style.html
R packages to help with code style : https://styler.r-lib.org/ and https://github.com/jimhester/lintr



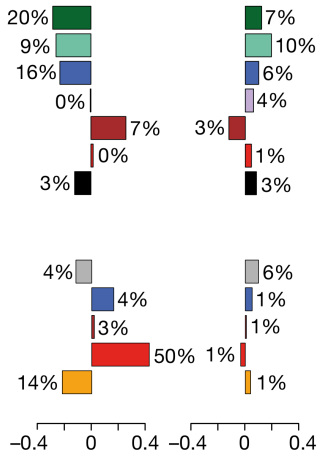
- all style guides
  - are fundamentally opinionated
    - some decisions make code easier to use (especially matching indenting to programming structure)
    - but many decisions are arbitrary

- most important thing about a style guide
    - that it is consistent
  - makes code :
    - easier to read
    - easier to write
    - → you need to take fewer decisions

- R packages to help you :
- lintr performs automated checks to confirm that you conform to the style guide.
- styler interactively restiles selected texts.
- I personally don't use one of these packages

# comment your code

- code without comments can be as useless as a plot without axis labels
- comment the unexpected
  - everything not obvious from your code
  - e.g. if you use a solution from the internet, provide the link in a comment
  - explain alternative values for parameters, or things you had to look up when you wrote the code
- better too much than too few comments




```
//When I wrote this, only God and I understood what I was doing
//Now, God only knows
```

# comment your code

```r
###
# EXAMPLE 2 - COMMENT YOUR CODE
#
# Bad example : comment the obvious
# read table example.csv
info_data <- data.table::fread("example.csv", header = T)
#
# Good example : comment the unexpected
# use package data.table to read in very large table "example.csv"
info_data <- data.table::fread("example.csv", header = T)
```

https://github.com/biodiversity-exploratories-synthesis/useful_functions/blob/main/2021_synthesis_Rhelpdesk_good_programming_practices.R

# make a script header

- Use the first few lines of a script to …
  - explain the purpose of the script
  - date and creator
  - state requirements / dependencies
    - e.g. previous script's output
  - (voluntary) include an example

```
#####################################
#                                   #
# GOOD PROGRAMMING PRACTICES        #
#                                   #
#####################################
# Biodiversity Exploratories Synthesis R helpdesk
# Noëlle Schenk 05.02.2021
#
# this file contains examples of practices to improve the
# readability of your code.
```

https://github.com/biodiversity-exploratories-synthesis/useful_functions/blob/main/2021_synthesis_Rhelpdesk_good_programming_practices.R



- use the possibilities to format your script
  - small time investment
  - with big gain: much faster finding your way around scripts

# make a script header

- libraries go first

```
library(data.table)
library(reshape2)
library(tidyr)
```

- hard-coded variables go second

```
path_to_data <- "example_data_synthesis_helpdesk/"
path_to_output <- "R_outputs_and_plots/"
```

- relative paths over absolute paths

```
absolute_path <- "C:/user/yourname/Documents/Project1/Rscripts/data/info_data.csv"
relative_path <- "/data/info_data.csv"
```

- set a seed

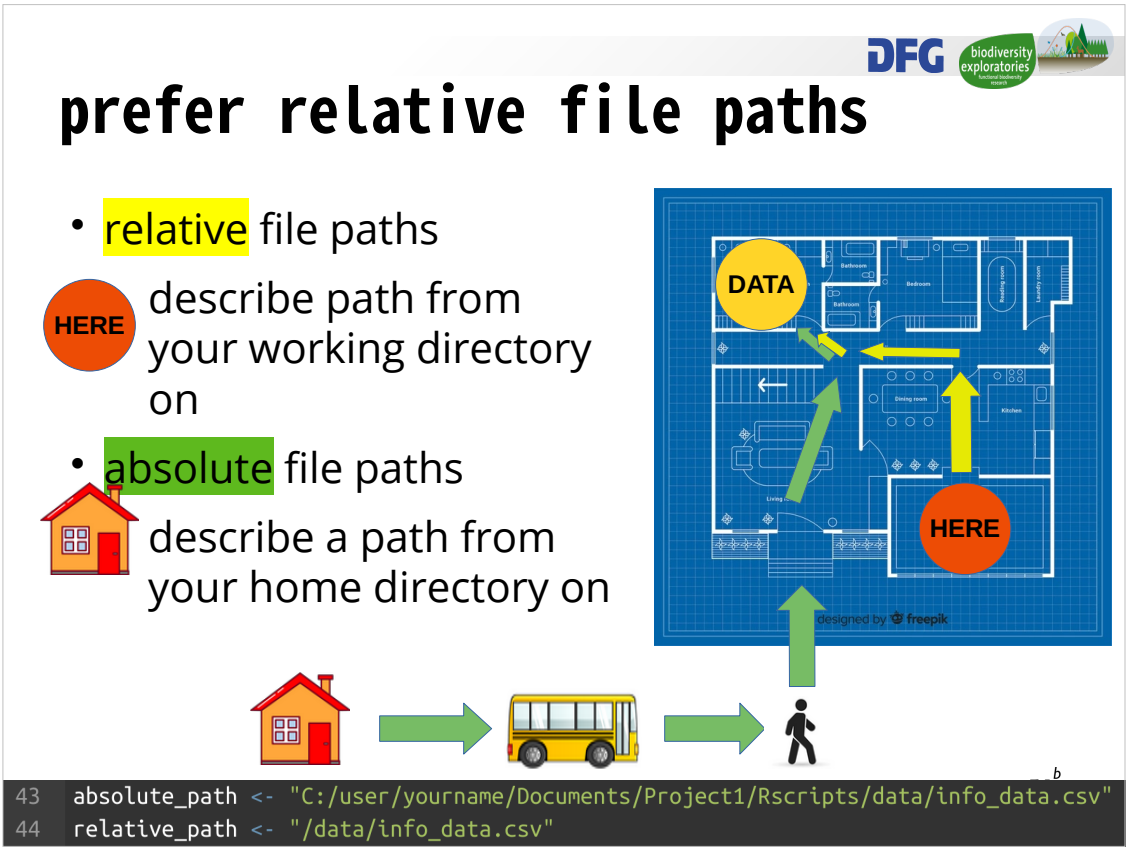
```
set.seed(12)
```

https://towardsdatascience.com/best-practices-for-r-programming-ec0754010b5a



- after the written header :
- give dependencies :
- all used packages
- all hard-coded variables
- usually paths to files
- in general : relative > absolute
- the path to a file is not the same in all computers
- → make code more general with relative path
- document and share your folder structure
- → others can replicate
- (see later)

- if you set a seed in general : do at the beginning

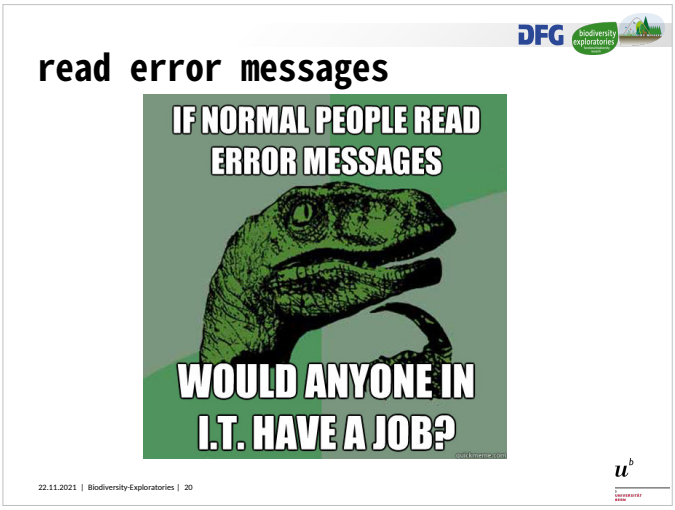



The difference between absolute and relative file paths
might not be very easy to grasp
might not be obvious why relative paths are preferred

Differ between relative and absolute :
relative ALWAYS start from your current working directory
(where you tell R that you are)
absolute ALWAYS start from “home”
the home directory of your file system

Reproducibility :
other people can set up the same project folder structure
if you tell them how to
others CAN NOT reproduce the folder structure of your whole computer

→ relative paths are easier to reproduce




This meme is a bit harsh

- of corse, usually computers tell us what is wrong
  but we don’t understand

but

- I found particularly hard to understand them at the beginning
- Who reads the error messages?
- Who tried to read the error messages?

- I think it can be quite hard to learn the language of error messages,
  - so the Picture is not really right I my opinion, because it acutally takes some experience to get used to read and understand the messages. But you can learn it!

- first step to learn it is to get an error
  that's maybe the easiest part
- then of corse, read the message.
  Even if you don't understand it
- take apart :
  error – copied function from above – unused arguments
  → aparently I gave to many arguments to the function, and now it doesn't work

- what would be a good next step in case you don't understand the message after trying to read it? What do you think?

# read error messages

```
> # read the error messages
> x <- as.factor("A", "B", "C")
Error in as.factor("A", "B", "C") : unused arguments ("B", "C")
>
```

```
>
> help("as.factor")
>
```

factor {base} R Documentation

## Factors

**Description**

The function `factor` is used to encode a vector as a factor (the terms 'category' and 'enumerated type' are also used for factors). If argument `ordered` is TRUE, the factor levels are assumed to be ordered. For compatibility with S there is also a function `ordered`.

`is.factor`, `is.ordered`, `as.factor` and `as.ordered` are the membership and coercion functions for these classes.

**Usage**

```
factor(x = character(), levels, labels = levels,
       exclude = NA, ordered = is.ordered(x), nmax =

ordered(x, ...)
```



- possible next steps:
    - ask the help function
        - it's similar as with the error messages :
            - – it takes some time to understand it.
        - how is it for you?
            - Do you often use it? Do you think it is hard to understand?

what are alternatives if you still don't understand?

- package documentation
- forum in the internet
- ask your search engine (google) directly

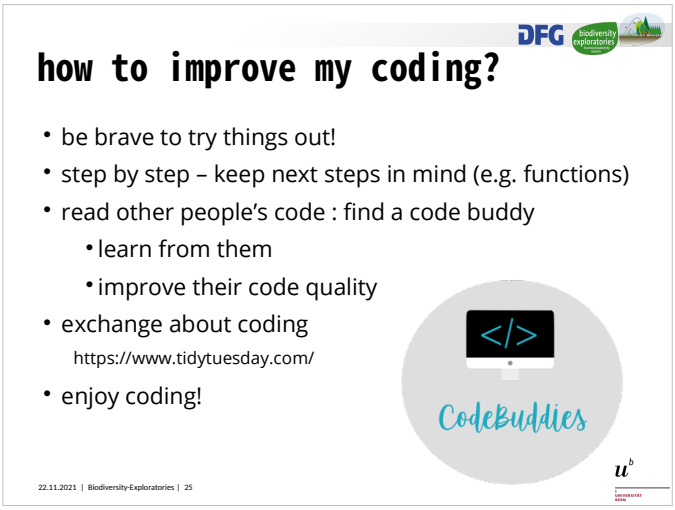
- copy paste the whole error message to google and look for answers.

- Ideally, only **the second part** of the error message where the error type is described or just put everything as I did here.

- good answers in forums

eventually find out :

- give a vector
- if not, function thinks there were 3 arguments




- the best way to learn programming is to do it.

  - be brave to try things out :
    - you just learn, playing a bit around with it

  - go step by step
    - keep next steps in mind

- read other peoples code :
  - we learn coding alone
  - coding differs from person to person
  - very useful package
  - a different strategy to address problems

- exchange with your collegues about coding
    - if you learned a new thing that you find useful
    - share your struggles

- enjoy coding!
    - celebrate when you mastered something hard

# enjoy coding

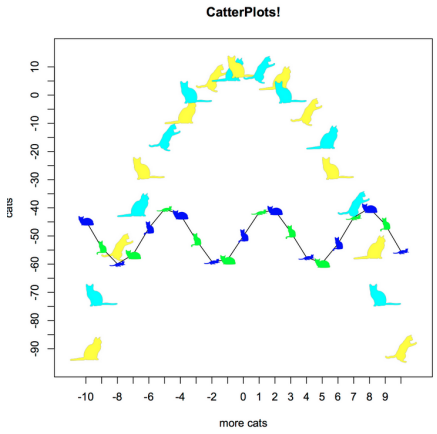
- the two stages of every programmer :
  - if you are on the right side - believe that it will change again
  - if you are on the left side – enjoy it!
    … and forget that it will change again

# enjoy programming

- funny packages :

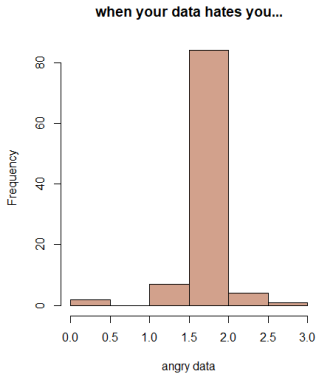
```
#> [1] "You are super-excellent!"
```

  - CatterPlots : https://github.com/Gibbsdavidl/CatterPlots
  - praise : https://github.com/rladies/praise




```r
# plotting of R objects
plot <- function (x, y, ...)
{
  if (is.function(x) &&
      is.null(attr(x, "class")))
  {
   if (missing(y))
      y <- NULL

    # check of ylab argument
    hasylab <- function(...)
      !all(is.na(
        pmatch(names(list(...)),
               "ylab")))
```




- some things may seem like procrastination
  but they really help you learn programming

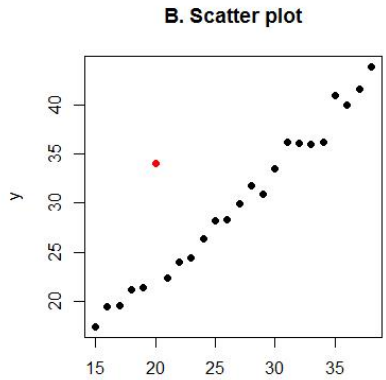
- find funny packages
  catter plot
  praise me

- or sometimes your plot looks funny, like this one showing the finger.

- play around with RStudio,
  use different appearances,
  try out working with Rmarkdown,
  play around with plotting
  colour palettes

# Test your code

- Does the line you just wrote does what you were expecting?
  - simple summary stats (number of NA, min and max values)
  - visual checking : inspect table with “View()”, plot, print out a column
  - check the format (number of lines, data type (e.g. factor, character, numeric))
- keep this little checks in your script – if you change something later, it will help you to check if everything still works



## Test your code

```r
###
# EXAMPLE - Test your code
#
# create example data.frame
example <- data.frame(species = c("Erithacus rubecula", "Arion vulgaris",
                                  "Plantago Lanceolata"),
                      abundance = c(2, NA, 10))
# action : recode NA value to 0
#    because Arion vulgaris was measured, but no individuals were found
example[which(example$species == "Arion vulgaris"), 2] <- 0
#
# test : did I do what I wanted?
sum(is.na(example$abundance)) # no NA values any more
sum(is.na(example$species))
min(example$abundance)        # min is 0 now, as expected
max(example$abundance)
View(example)
plot(example)
example$abundance
```



Go to RStudio and show it

# `unit testing (advanced)`

- entangled software **breaks upon tiny changes** (e.g. new R version, package updates, bug fixes, ...)
- write integrated test functions which check if your code works right
- unit testing in R :
- e.g. **`testthat`**

https://towardsdatascience.com/unit-testing-in-r-68ab9cc8d211

# unit testing (advanced)

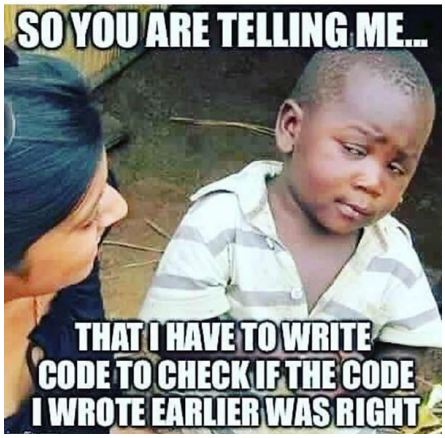
## The Art of Software Testing, Third Edition

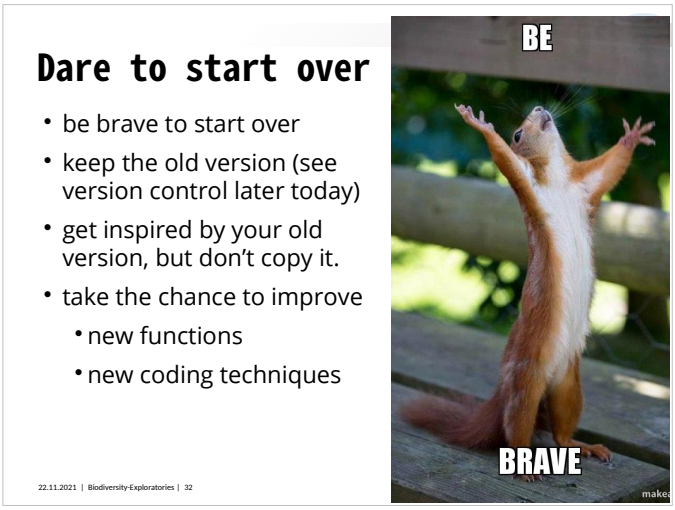
Editor(s): Glenford J. Myers, Tom Badgett, Corey Sandler

First published: 2 January 2012

Print ISBN: 9781118031964 | Online ISBN: 9781119202486
| DOI: 10.1002/9781119202486

Copyright © 2012 by Word Association, Inc. All rights reserved





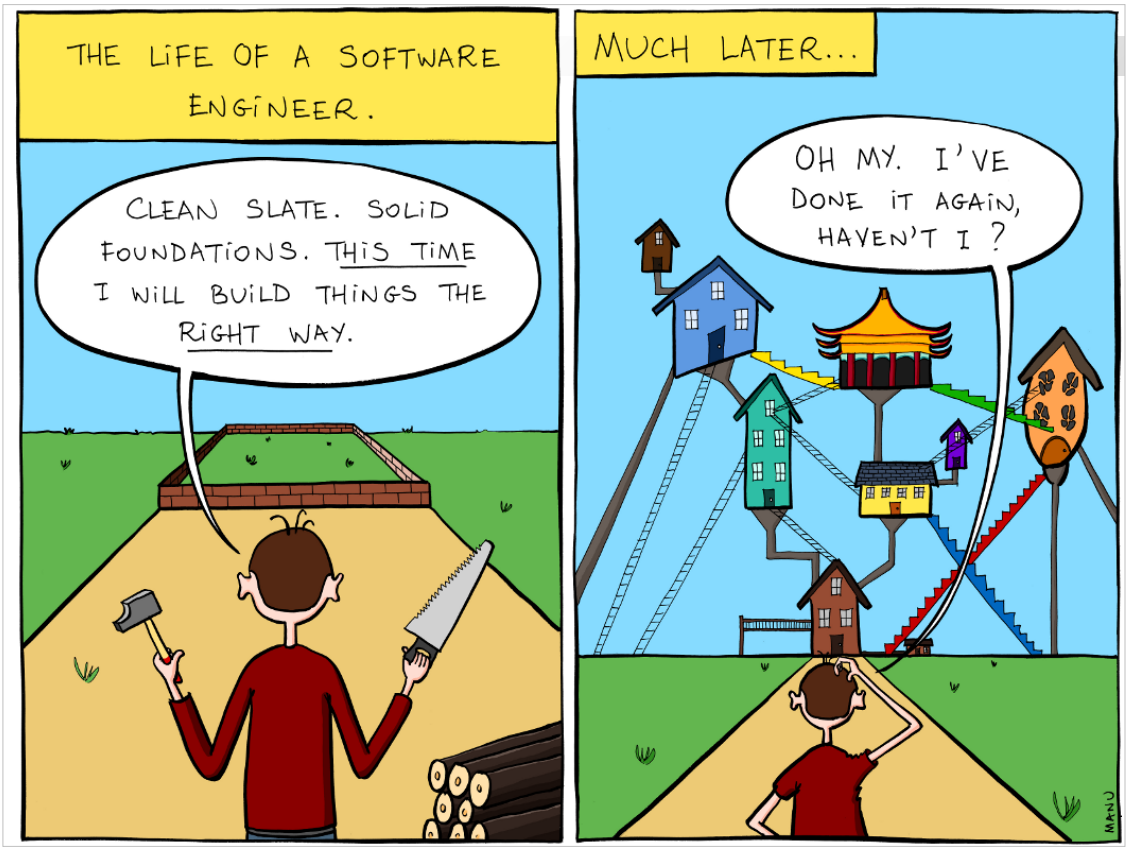

- sometimes :
  - we just code a mess
  - the script is buggy and does not work
  - we have to scroll up and down to find things
  - we coded in past and now know much better how to do
  - → just not comfortable to work with the script any more.

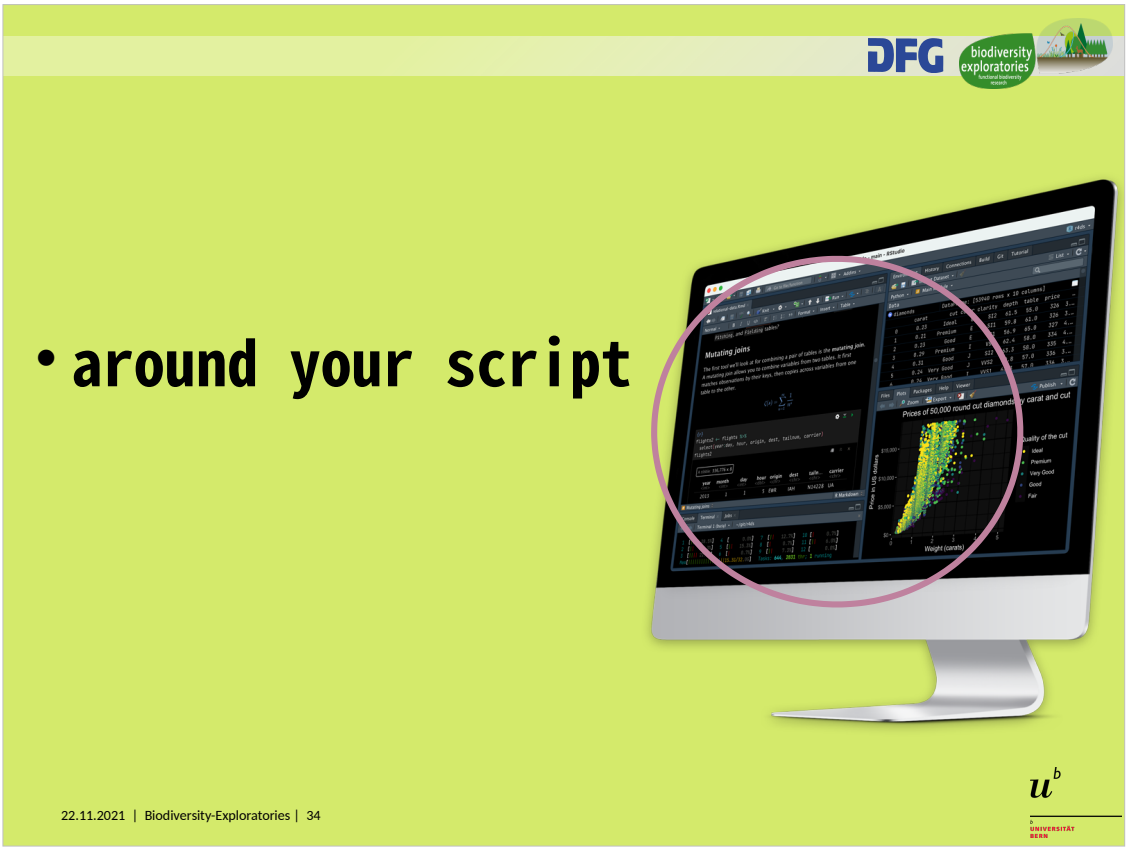
- Then, you can be brave and start over with a new script
  - or better say:
    - a new version of the script
  - this is called refractoring :
    - change your script without changin in- and ouput

- most probably :
  - it pays off to have a clean script
    - which is easy to work with
  - because most probably you will use it again in the future –
    - maybe to look up something
    - or to adjust
    - or try out something new for your analysis.

- good opportunity to
  - learn and practice your new coding skills.

- if you write new code :
  - don't just copy paste these lines,
    - but go through them and
      - improve where it can be improved
      - take out unneccessary lines
      - adapt your comments to the new situation, ...

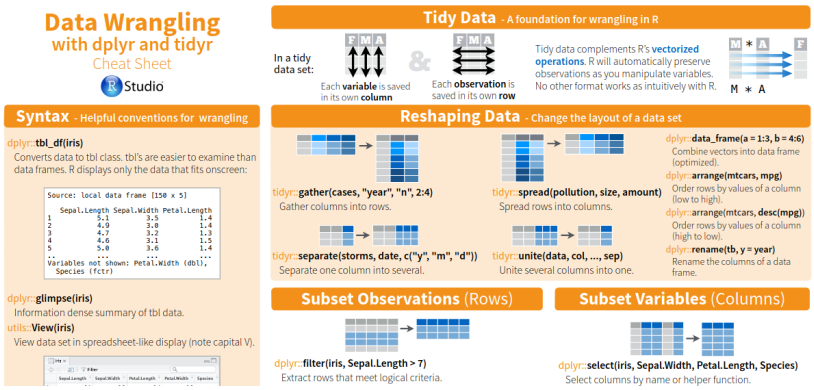

THE LIFE OF A SOFTWARE ENGINEER.
CLEAN SLATE. SOLID FOUNDATIONS. THIS TIME I WILL BUILD THINGS THE RIGHT WAY.
MUCH LATER...
OH MY. I'VE DONE IT AGAIN, HAVEN'T I ?
MANU




- zoom out and find out what you can do around your scripts

# use cheat sheets

- make your own or use cheatsheets provided by others
- RStudio > Help > Cheatsheets or Google “Cheatsheet <topic>”




Data wrangling cheatsheet example : https://rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf
Cheatsheet page of RStudio : https://rstudio.com/resources/cheatsheets/



- cheat sheets are
  - a collection of most important functions
  - and how they can be used

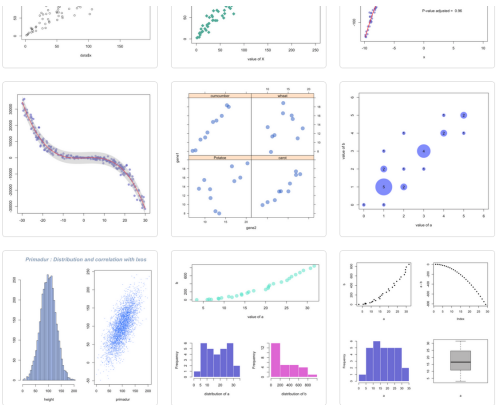
- make your own cheatsheets
  - put on any new and useful thing that you learn in R
  - option to share with others, e.g. at RStudio cheatsheet page
  - find empty templates there
- or you just use cheatsheets made by others
  - far less time consuming

- where to get them from?
  - some come with RStudio
  - go to RStudio cheatsheet page
  - google

- I personally enjoy most :
  - using the cheatsheets from others and
  - writing my own notes on it.

# Use other people’ s solutions

- Most coding problems have been experienced before by someone.
- There is a good chance that someone has posted the solution on the internet.
  - ideally : R package
  - https://stackoverflow.com/
  - http://www.sthda.com/english/
  - (https://stats.idre.ucla.edu/r/dae/robust-regression/ https://www.statmethods.net/stats/correlations.html)
  - https://www.r-graph-gallery.com/
- ask a more experienced person





- most problems about coding have been experienced before by someone else
  - chance high that :
    - this someone posted the solution to the internet
- take time to search for a solution
  - before you spend too much time on it
- ideally :R package
    - piece of code
    - at least a hint
- how?
  - google
    - vocabulary
  - coding forum
  - ask questions yourself!
    - respect the rules :
      first search yourself
      provide a reproducible example
      one that doesn’t need your data (make a data.frame)
      we can look at if you want

- ask more experienced person
- ask us

# Example : R Packages as solutions to common problems

- Problem : Search for taxonomic names and resolve them
- Solution : use specialised R package : **`taxsize`**
  - e.g. correct misspelled species names : *Helianthos annus* to *Helianthus annuus*
  - e.g. many synonyms for a species :
  - 

```
c("Helianthus annuus ssp. jaegeri",
"Helianthus annuus ssp. lenticularis",
"Helianthus annuus ssp. texanus",
"Helianthus annuus var. lenticularis",
"Helianthus annuus var. macrocarpus",
"Helianthus annuus var. texanus")
```

→

```
acceptedName
"Helianthus annuus"
```

- Alternative solution (e.g. similar problem, but not related to taxonomic names)
- Use a more general R package : e.g. **`stringr`**
  - e.g. normalise synonyms in questionnaire answers :
  - "`increased`" and "`improved`" → "`positive`"

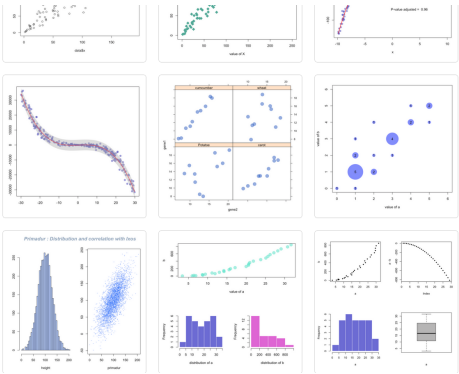


UNIVERSITÄT BERN

# Use other people’ s solutions with care

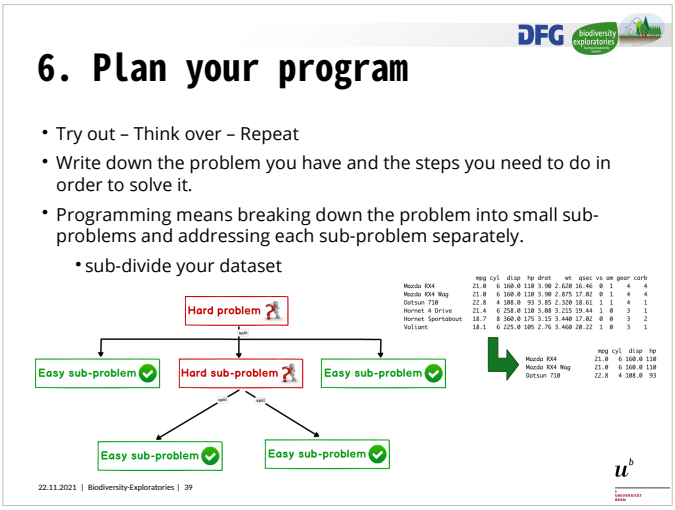
input x → function f → output y

- If you use someone elses solution:
  - make sure that you try it out (e.g. with an easy example)
  - carefully chek if it does what you want it to do.
- Example :
  - View(), head() and str() , table()
  - expected number of : rows, columns,
  - column/ rowwise : expected number of NAs, expected range of values, names still correct?
  - what happens if you give a cornercase input: e.g. a reduced/ empty dataset as input, an invalid argument, a very large/ negative number, zero





- If you found a good solution
- use it with care :
  - try it out with a subset of your data
  - does it do what you want it to do?
  - try to understand why it works
  - test the outcome
  - test **cornercase** input
    - e.g. empty dataset
    - if you expect an input number of range 1-100, test 1, 0, 100 and 101
  - be aware of specialities of your dataset
    - what you found out while exploring it, e.g.
      - NA values
      - zero or negative values
      - additional columns




- Plan your program
- easier to address many easy sub-problems
    - that a large and maybe hard one
- try to subdivide your problem into sub-problems

- how?
    - write it down
        - first describe the problem
        - while describing :
            - check if you detect sub-problems crystallising
            - write them as tasks
        - start with first sub-problem (try out)
            - find out if it can be subdivided while addressing (think over)
        - start with first sub-problem (try out)
        - think over
        - …

- If it seems insolvable :
- search internet
- consider asking a forum
- there you will need to describe the problem anyway
- good that you already did this

- subdividing the problem
- can also mean to
- subdivide your table!
    - only use first few lines and columns
    - much easier to spot errors/ find out if solution works!
- check if solution works
- increase size of your table
- test

# 6. Plan your program

```
###
# EXAMPLE 6 - PLAN YOUR PROGRAM
#
# Describe your problem and the steps you need to do to solve it.
# Example problem : Plot a normal distribution with mean = 0 and sd = 1.
# Example planning :
#    - generate a normally distributed dataset with mean = 0 and sd = 1
#        - search internet which function can generate normally distributed data
#        - look up help of this function in R
#        - try out function
#        - make vector x with normally distributed data
#    - make the plot
#        - search internet which function to use for plotting
#        - make title and axis labels

help(rnorm)
rnorm(4, mean = 0, sd = 1)
x <- rnorm(40, mean = 0, sd = 1) # first parameter : length of vector
plot(x)
# plot() just shows the data.points. Maybe a Histogram would fit better?
#TODO plot histogram.
```

https://github.com/biodiversity-exploratories-synthesis/useful_functions/blob/main/2021_synthesis_Rhelpdesk_good_programming_practices.R

# 6. Plan your program

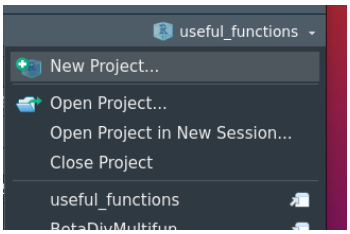
```
###
# EXAMPLE 6 - PLAN YOUR PROGRAM
#
# Describe your problem and the steps you need to do to solve it.
# Example problem : Plot a normal distribution with mean = 0 and sd = 1.
# Example planning :
#    - generate a normally distributed dataset with mean = 0 and sd = 1
#        - search internet which function can generate normally distributed data
#        - look up help of this function in R
#        - try out function
#        - make vector x with normally distributed data
#    - make the plot
#        - search internet which function to use for plotting
#        - make title and axis labels

help(rnorm)
rnorm(4, mean = 0, sd = 1)
x = rnorm(40, mean = 0, sd = 1) # first parameter : length of vector
plot(x)
# plot() just shows the data.points. Maybe a Histogram would fit better?
#TODO plot histogram.
```

https://github.com/biodiversity-exploratories-synthesis/useful_functions/blob/main/2021_synthesis_Rhelpdesk_good_programming_practices.R

# use RStudio projects

- divide your work into contexts
  - each with it's own working directory
  - workspace (opened documents)
  - history
- Create new project:
  - in a brand new directory
  - in an existing directory
  - by cloning a version control (e.g. git, see later)




Tutorial creating RStudio project : https://support.rstudio.com/hc/en-us/articles/200526207-Using-Projects

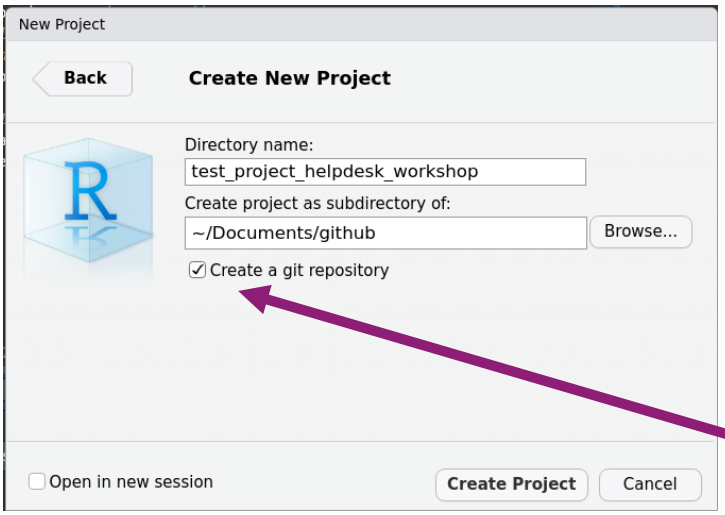


- RStudio offers some good tools
  - most you might know
    - autocompletion
    - suggesting functions
    - workspace overview (you see an overview of your variables, ...)
  - one of it : RStudio projects

- provide a context for your project
  - each project has it's own
    - working directory
      - directory / folder where
        - you are working (working directory)
        - your scripts are stored
        - R looks for files to read in and
        - R will write your output
    - workspace
      - which files are open
      - assigned variables
      - plots
      - opened help pages
    - history
      - where commands you made are saved
      - **show it! + show how to make Rproject (Typora)**

# use RStudio projects

- create a new RStudio project

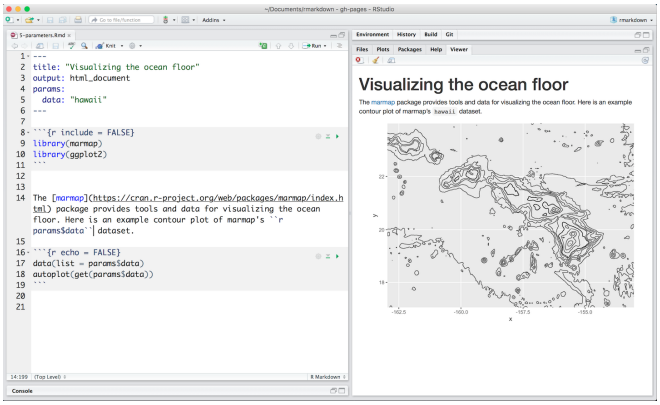



will be important
for version control

# Rmarkdown

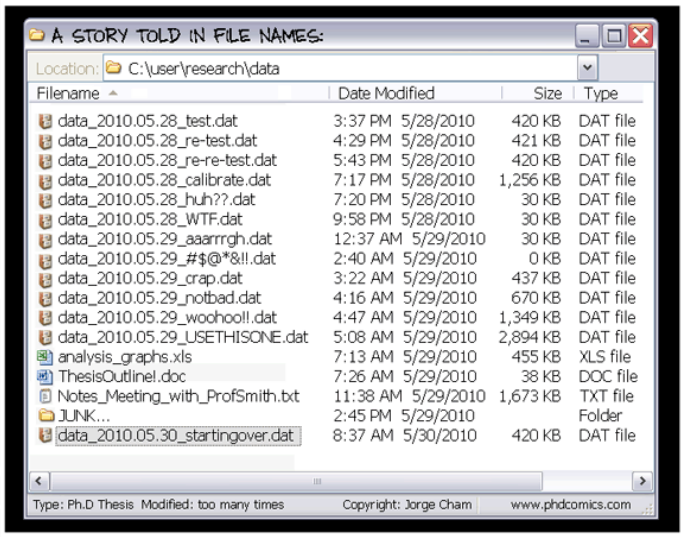
- combining R script with documentation
- Suitable to document workflow
  - combine analysis and documentation
    https://rmarkdown.rstudio.com/lesson-1.html

- Distill package for scientific appearance : https://rstudio.github.io/distill/



- around your script
- version control

# purposes of version control



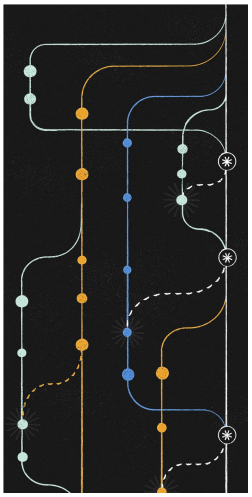



- new project
  - straightforward to keep track of the files
- can become unmanageable very quickly
  - if revisions happen frequently

- **go through example on slide**

- usage example without version control: You create an R script which reads and analyses dataset A. Some time later, dataset A gets updated, some errors were corrected and some new data came in. You would like to adapt your R script now to this new dataset, but at the same time, you don't want to lose the old version. One is of corse tempted to copy and paste the script, keeping one for the old dataset and adapting the other copy to the new dataset. So you do that. But then, suddenly, working on the adaptations of your script to the updated dataset, you find an error in the code and you correct it. Hopefully, you remember to correct it in the other version for the old dataset, too. Also, you do a lot of reformatting and write some new comments. More and more, your two versions of the script get different and different, and the stable version for the old dataset soon does not seem to be very handy (or correct) any more.

# purposes of version control



- keep track of changes
- keep scripts clear and structured
- understand previous changes
- compare different versions
- facilitate collaboration
- short- and long-term undo



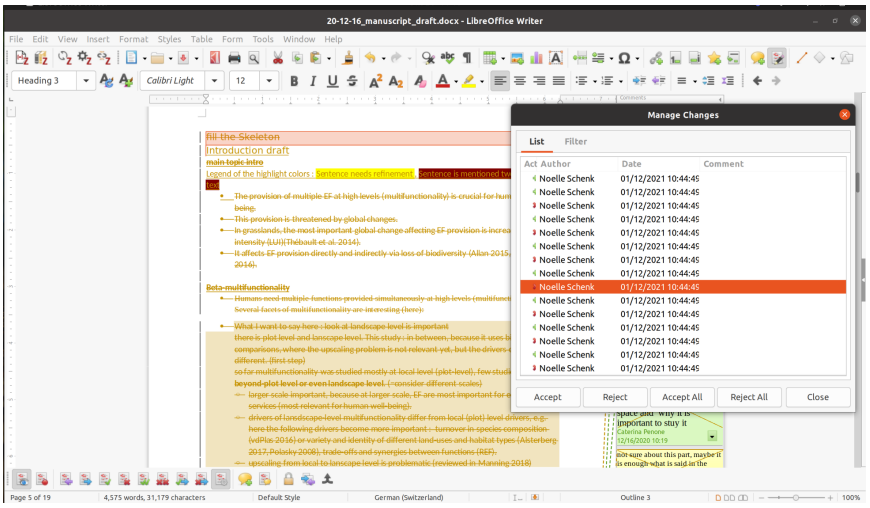
version control
- imagine as series of snapshots of project's changes
- stores entire history of project
- each snapshot
  - saves state of project at that time
  - travel back

save version of your project after making changes
- essential habit
- without version control :
  - tedious and confusing quickly :
  - how much to save? only changed files or everything?
  - naming the versions?
  - if you are very organised : stick to comprehensive naming scheme
  - how do you know what is different in these versions?

- version control :
  - addresses this types of issues
  - there is only one current version
  - everything else :
    - neatly packed up inside version control
  - on demand :
    - request older version

to summarise : SLIDE

# what is version control?

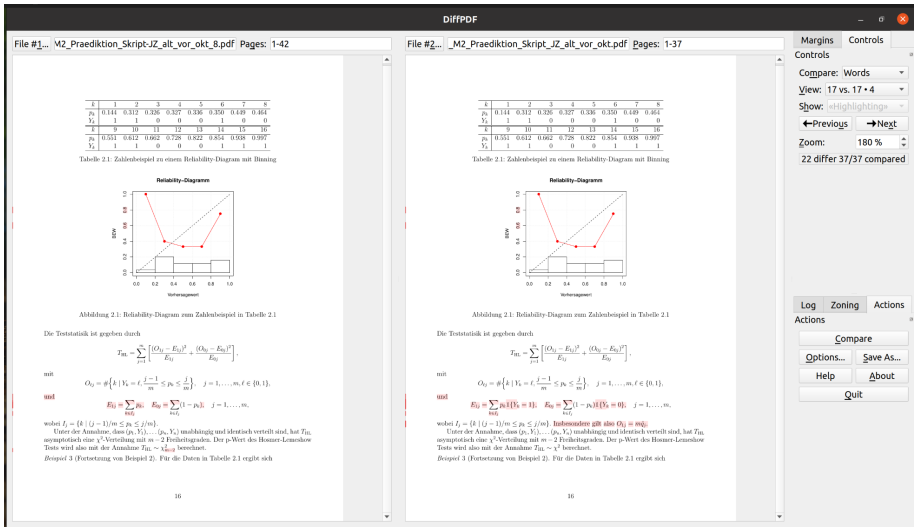



- Track Changes is a “one time version control”
- Edit > Track Changes > Compare document
- No commenting, just two documents



- now hopefully I could convince you to use a version control
- but you probably still don’t know what it is
- probably know “track changes” in writing programs such as word :
  - is a one-time version control
  - it tracks changes and you can comment on them
  - but after you accept them, they disappear
  - sometimes : save milestones versions and even comment on changes → quite near to version control for code

# what is version control?




- Comparing two documents (e.g. pdfs) creates a “diff” file
- Shows what’s different between files
- Version control for code : saves only first version and all following “diffs”



- Also possible : compare two documents of any type, e.g. pdf
- comparing results in diff
- diff : a file showing the differences between files
- Version control for code :
- save complete first version
- everytime you change, only saves the diff to previous version
- → uses much less storage
- as a user you don’t care, the software handles all the details for you
- but it could happen that you look at a diff

# Version Control for code

git

SUBVERSION

**Etymology 1** [ edit ]

From Middle English *get* ("[illegitimate] offspring"). A southern variant of Scots *get* ("illegitimate child, brat"), related to *beget*.[1]

**Noun** [ edit ]

**git** (*plural* **gits**)

1. (*Britain*, *slang*, *derogatory*) A silly, incompetent, stupid, annoying or childish person (usually a man). [quotations ▼]

**Usage notes** [ edit ]

- *Git* is usually used as an insult, more severe than *twit* but less severe than a true profanity like *wanker* or *arsehole*, and may often be used affectionately between friends. *Get* can also be used, with a subtle change of meaning. "You cheeky get!" is slightly less harsh than "You cheeky git!".
- *Git* is frequently used in conjunction with another word to achieve a more specific meaning. For instance a "smarmy git" refers to a person of a slimy, ingratiating disposition; a "jammy git" would be a person with undeserved luck. The phrase "grumpy old git", denoting a cantankerous old man, is used with particular frequency.
- In parts of northern England, Northern Ireland and Scotland, *get* is still used in preference to *git*. In the Republic of Ireland, *get*, rather than *git* is used.
- The word has been ruled by the Speaker of the House of Commons to be unparliamentary language.[2][3]

**Translations** [ edit ]

**±A silly, incompetent, stupid, annoying or childish person** [show ▼]



version control : specific software for version control for code

For code :
two main softwares : git and Subversion
we look at git (more support an d tutorials)

usually, when people introduce something,
they say something about the origin of the name
here, I don't know

# Version Control for code

git SUBVERSION

```
GIT - the stupid content tracker

"git" can mean anything, depending on your mood.

 - random three-letter combination that is pronounceable, and not
   actually used by any common UNIX command.  The fact that it is a
   mispronunciation of "get" may or may not be relevant.
 - stupid. contemptible and despicable. simple. Take your pick from the
   dictionary of slang.
 - "global information tracker": you're in a good mood, and it actually
   works for you. Angels sing, and a light suddenly fills the room.
 - "goddamn idiotic truckload of sh*t": when it breaks

This is a stupid (but extremely fast) directory content manager.  It
doesn't do a whole lot, but what it _does_ do is track directory
contents efficiently.
```

- initial git commit of README by Linus Torvalds, creator of git



- Officially, the creator of git, Linis Torvalds was interviewed about the reasons for the name

  - easy to say
  - not occupied by other command yet

# Version Control for code

git SUBVERSION

**plot overviewbars of multiple models : added option for dodged barplo...** Browse files

...ts for better overview

master

**noschenk** committed 19 days ago 1 parent 6eb43e4 commit aacc9b9df485ec99079b966b69e38c2773b8c5b1

```
271  273    #' @export
272  274    ### FUNCTION
273       - create_single_funs_overviewbar_plot <- function(singleF_restab, legend = F){
     275  + create_single_funs_overviewbar_plot <- function(singleF_restab, legend = F, pos = "stack"){
274  276      if(legend){
275  277      # return a plot with legend
276  278        print("not implemented yet.")
277  279      } else {
278  280        sf_ov <- ggplot(singleF_restab, aes(x = variable, y = value, fill = color)) +
279       -      geom_bar(stat = "identity", color = "black") +
     281  +      geom_bar(stat = "identity", color = "black", position = pos) +
280  282        scale_fill_identity("", labels = singleF_restab$ground,
281  283                                breaks = singleF_restab$color,  guide = "legend") +
282  284        # coord_flip() +
```

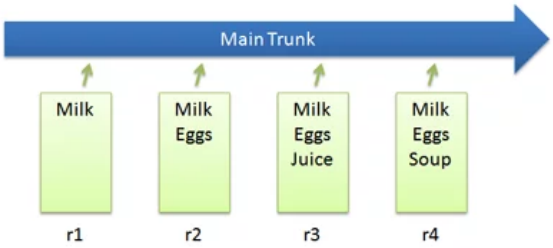
- Instead of saving versions of files, only save the “diff”
- Comment on every change
- Reconstruct any previous version as you need it



- Here :
  - diff file in git :
    - shows difference between 2 versions of code :
      - added a plotting parameter to a function in 2 places
      - rest of the file is the same
    - → instead of saving whole new file, saves only the changes = the “diff”
- that means : version control is file-specific
  - for each file, you decide individually if it’s under VC or not
  - git and subversion are really not suitable to put other documents than plain text files (like code) under VC
    - no word document
    - no pdf
    - no powerpoint
    - nothing that uses a fancy GUI

# what is version control?

- easiest case : the “project” just consists of one file which experiences changes :

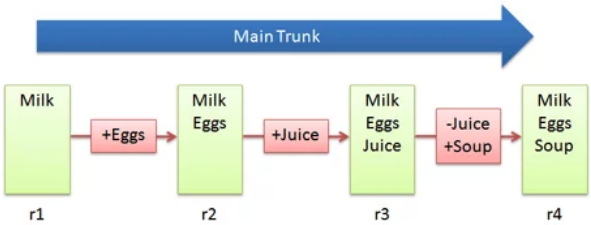





- I will explain how it works with another example
- easiest case :
  - your project only contains one script
  - this script experiences changes
- the folder in which your script is stored is called a “repository”
- this is version control slang
- here, script contains the word “Milk”
    - after, you add “Eggs”
    - some time after, you add “Juice”
    - and later, you replace “Juice” by “Soup”
- And then eventually you realise that this is not a very helpful R script

# what is version control?

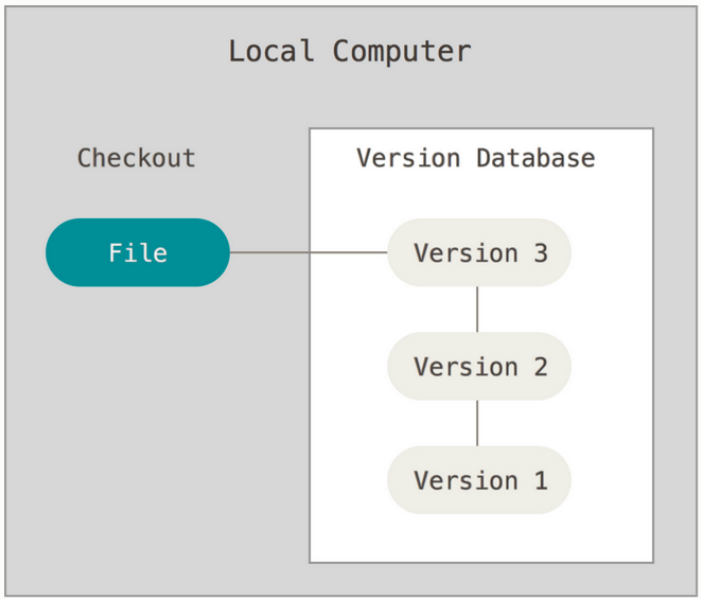
- easiest case : the “project” just consists of one file which experiences changes
- diffs : Are a precise description of the changes made to a file.
- The changes to your file are stored as diff : this does not use a lot of storage and is easier to look at (you don’t have to search where exactly in your file were the changes)






- here, the differences among the versions are still quite easy :
- first, you added milk
- then, you added Juice
- …
- version control does not store the file itself
  - but only the changes made to it
- the red rectangles show
  - the changes
- this is what version control actually saves
- they are called “diff”
- it is a special file format (= way of writing a file)
- a free format (you can open with any freely available software)

# what is version control?

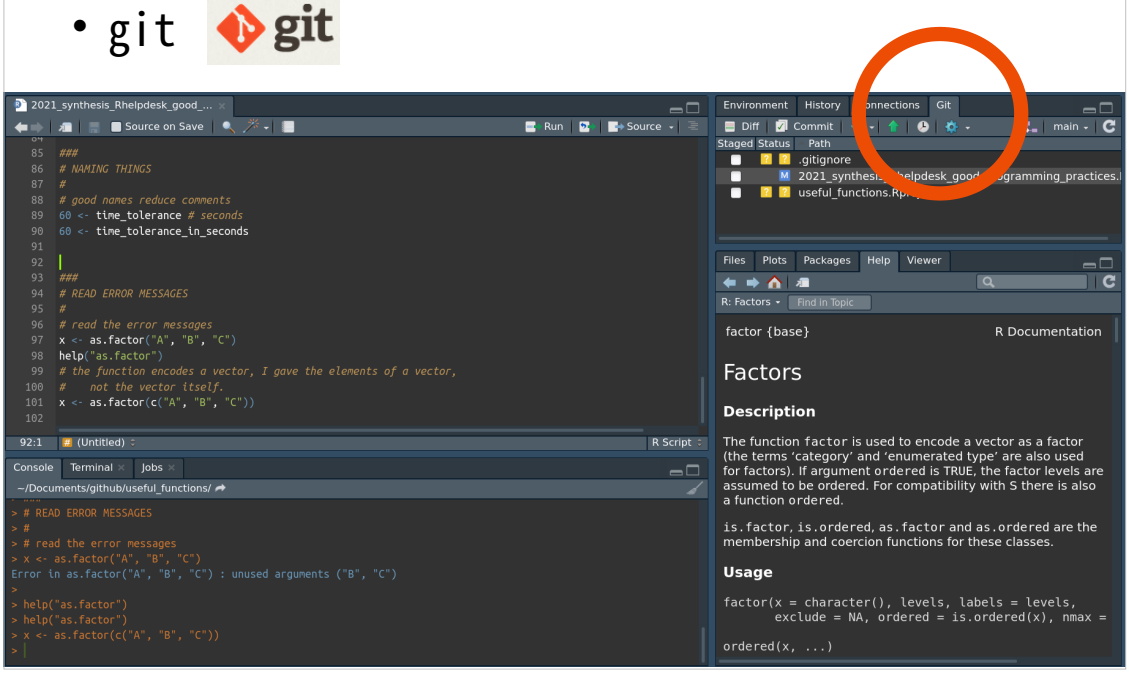





- So to show you another picture of the same principle :
  - the file is stored as a
  - sequence of changes
    - which represent the versions
  - all is stored associated to the same file
  - in your repository (=folder)
  - you still only see one file

# version control in RStudio

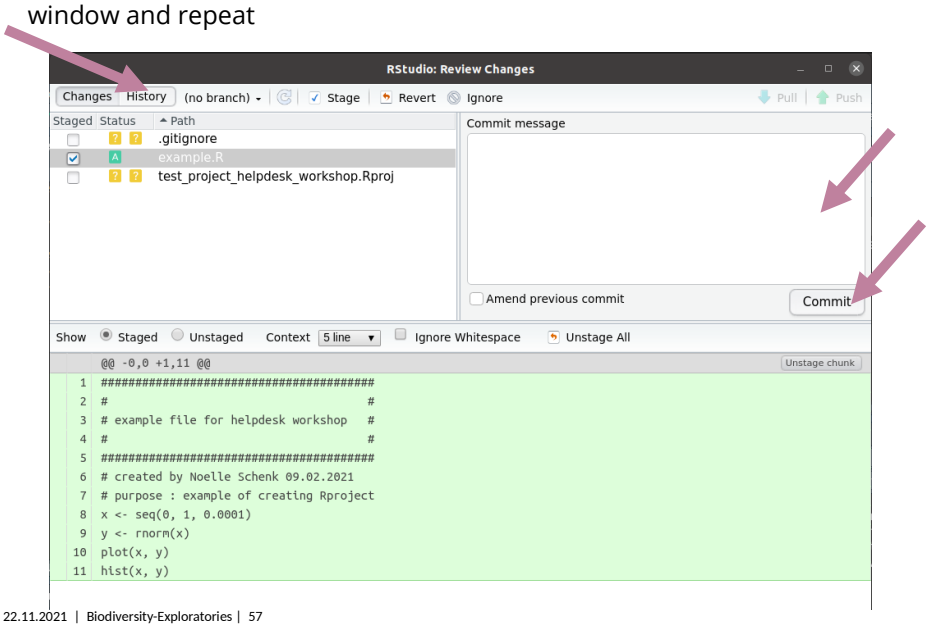
- git




- git works very well with RStudio
- it takes some time to set everything up
- I will show it to you on my computer

# guiding through version control

- make a change, check the box, click commit, write commit message, commit, leave window and repeat



# version control in RStudio

- git
- manual to set up git and RStudio
  https://happygitwithr.com/big-picture.html
- requires your folder to be an Rproject
- looking at history : see the commits you have done in the past

# GitHub

- GitHub is an external storage (owned by
- Windows) where repositories can be stored remotely
  - repository = the directory with your scripts in it
  - remotely = not in your local computer, but at a server in another place
- A place where you can share code and collaborate
  - but not for code publishing (e.g. zenodo)!
- If you want RStudio to work with GitHub, you also need “Push” and “Pull”.
- synthesis Github repository : https://github.com/biodiversity-exploratories-synthesis



- One thing that you will hear for sure is
- GitHub
  - it is often called a “version control”
  - but actually
  - it is a place where you can share your versions

- the actual version control happens locally on your computer
  - then you can decide to share your code
  - or to collaborate with others
- it is useful to know the difference between git and git HUB
- git is for version control, the hub is for sharing

- spent some time thinking an example :
- like if you make pictures with your smartphone
- the picture taking happens in your smartphone
- then you decide to share some of your pictures
- or all of them on a different platform
- like twitter
- the picture taking is on your smartphone, the sharing is another platform
- I am a big fan of GitHub, and I share like 90% of my code there
- but others prefere to use it differently,
- e.g. sharing the code on GitHub in an almost-finished state
- or keeping your code local
- you can decide

- If you want to use GitHub, I put some more information on the slide
- if you are interested in learning it, we can do during the exercises
- Is someone interested? (already?)

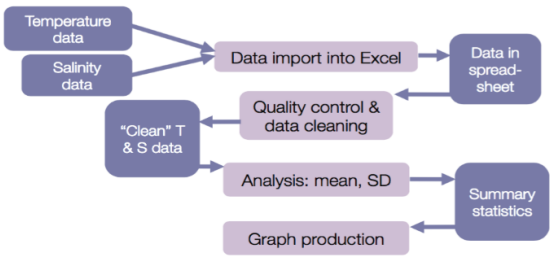
- around your desktop

# Document your workflow

- “all science is a workflow, but we rarely document it and rarely can we reconstruct all parts of it” – *unknown person in the internet*

*Workflow*: how you get from the raw data to the final products of your research

Simple workflow: flow chart






- all science is …

- not because we don’t know that it could be important
  - but  it’s really time-consuming
  - and tedious
  - I know
  - but it’s worth it
- in research, we have to go over our projects so often
  - mostly unexpectedly
  - and then you are happy to have the workflow documented
  - or when writing methods
  - to be reproducible

- There is no clear and one-for-all solution or  tool to do that
  - depending on your project, your collegues, …
  - some wirte it to the project README or document with Rmarkdown
    - ideally _ use a freely available, nonproprietary format like text file (see later)
  - you can do power point slide
  - anything

# keep raw data untouched

- make changes with R
- (document any changes)
- create derived data from raw data
- record all steps to process data – data manipulation is an integral part of your analysis!
- store together with metadata

- keep raw data untouched
  - it is sacred
  - with all its bumps and warts
- although it might be tempting to
  - correct mistakes
  - add some formatting
- you know better than me
  - there is so much work in data collection

- the risk is too high
  - that you make a change and later regret it
- but make changes with scripted language ®
  - → document all changes
  - record all steps used to process data
    - data manipulation is an integral part of your analysis
    - save intermediate data files
    - if you want to be able to re-run parts of your analysis
- change file permission to read-only
- if it helps

# create analysis-friendly data

- “tidy” data
- during ata collection : collection-friendly data
- prefer long-table formats over wide-tables
- each column : is a variable, units belong to variable  name
- each row : is an observaton (1 row per plot)
- header : 1 row with columnames
- additional info go to metadata
- names
- atomize

messy

| id | trt | work.T1 | home.T1 | work.T2 | home.T2 |
|---|---|---|---|---|---|
| 1 | treatment | 0.08513597 | 0.6158293 | 0.1135090 | 0.05190332 |
| 2 | control | 0.22543662 | 0.4296715 | 0.5959253 | 0.26417767 |
| 3 | treatment | 0.27453052 | 0.6516557 | 0.3580500 | 0.39879073 |
| 4 | control | 0.27230507 | 0.5677378 | 0.4288094 | 0.83613414 |

tidier

| id | trt | key | time |
|---|---|---|---|
| 1 | treatment | work.T1 | 0.08513597 |
| 2 | control | work.T1 | 0.22543662 |
| 3 | treatment | work.T1 | 0.27453052 |
| 4 | control | work.T1 | 0.27230507 |
| 1 | treatment | home.T1 | 0.61582931 |
| 2 | control | home.T1 | 0.42967153 |
| 3 | treatment | home.T1 | 0.65165567 |
| 4 | control | home.T1 | 0.56773775 |
| 1 | treatment | work.T2 | 0.11350898 |
| 2 | control | work.T2 | 0.59592531 |
| 3 | treatment | work.T2 | 0.35804998 |
| 4 | control | work.T2 | 0.42880942 |
| 1 | treatment | home.T2 | 0.05190332 |
| 2 | control | home.T2 | 0.26417767 |
| 3 | treatment | home.T2 | 0.39879073 |
| 4 | control | home.T2 | 0.83613414 |

| | Address | | Number | Street | City | State | Zip |
|---|---|---|---|---|---|---|---|
| | 415 20th St, Oakland CA 94612 | VS | 415 | 20th St | Oakland | CA | 94612 |

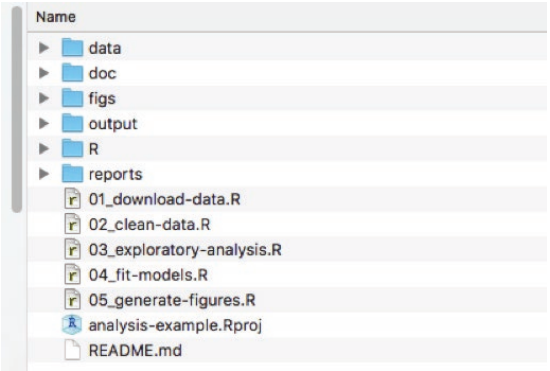
Hadley Wickham “R for Data Science” : https://r4ds.had.co.nz/tidy-data.html



- header : only one header row which contains column names.
  - The column names should be
    - unique within the table,
    - unambiguous and
    - easy to remember,
    - either lowercase OR only uppercase letters,
    - don’t contain any spaces or points or special characters except “_”, use english names
- atomize : one piece of information per cell (figure)

# folder structure

- each project to its own directory
- /data : raw data and metadata.
  - intermediate data : to separate subfolder in /data OR to /results
- /doc : manuscripts, documentations for code, lab notebook, …
- /figs, /output or /results : statistical tables, figures and publication-ready figures (subdirectories)
- /R : R scripts
- README
- /old

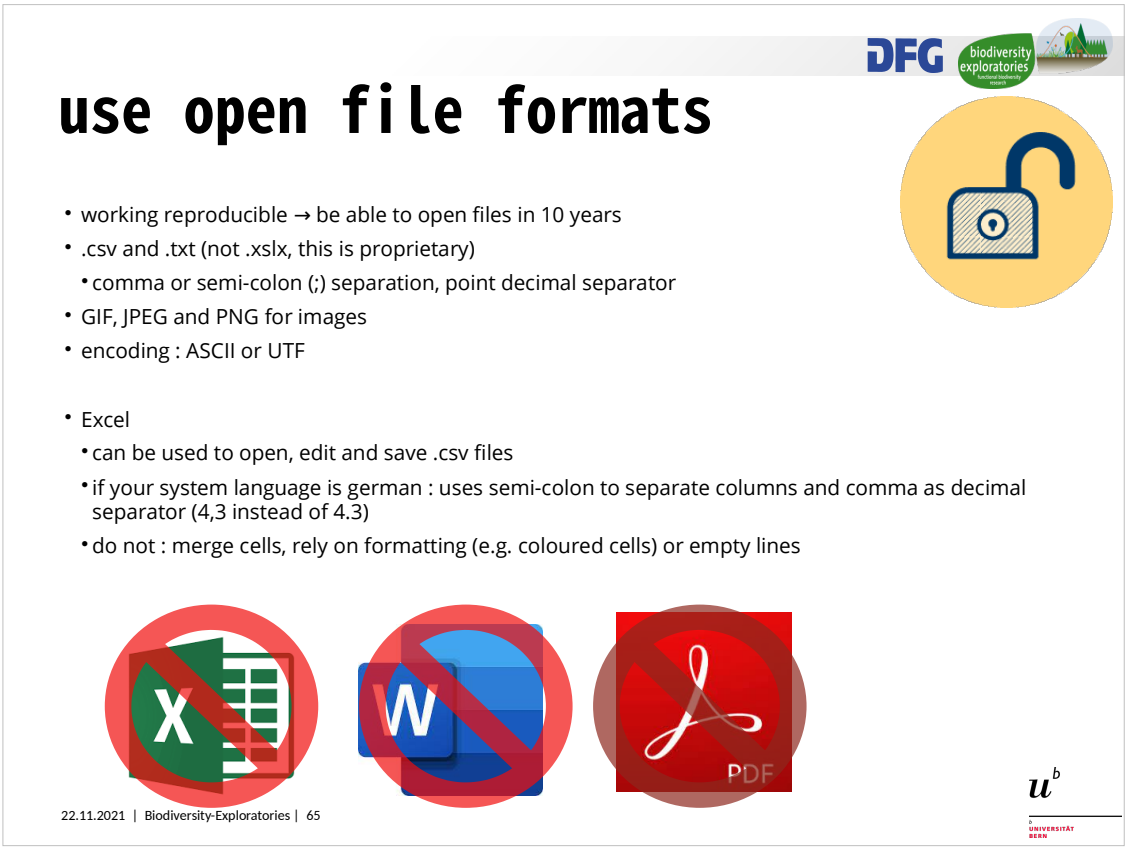



BES reproducible code manual :
https://www.britishecologicalsociety.org/wp-content/uploads/2019/06/BES-Guide-Reproducible-Code-2019.pdf

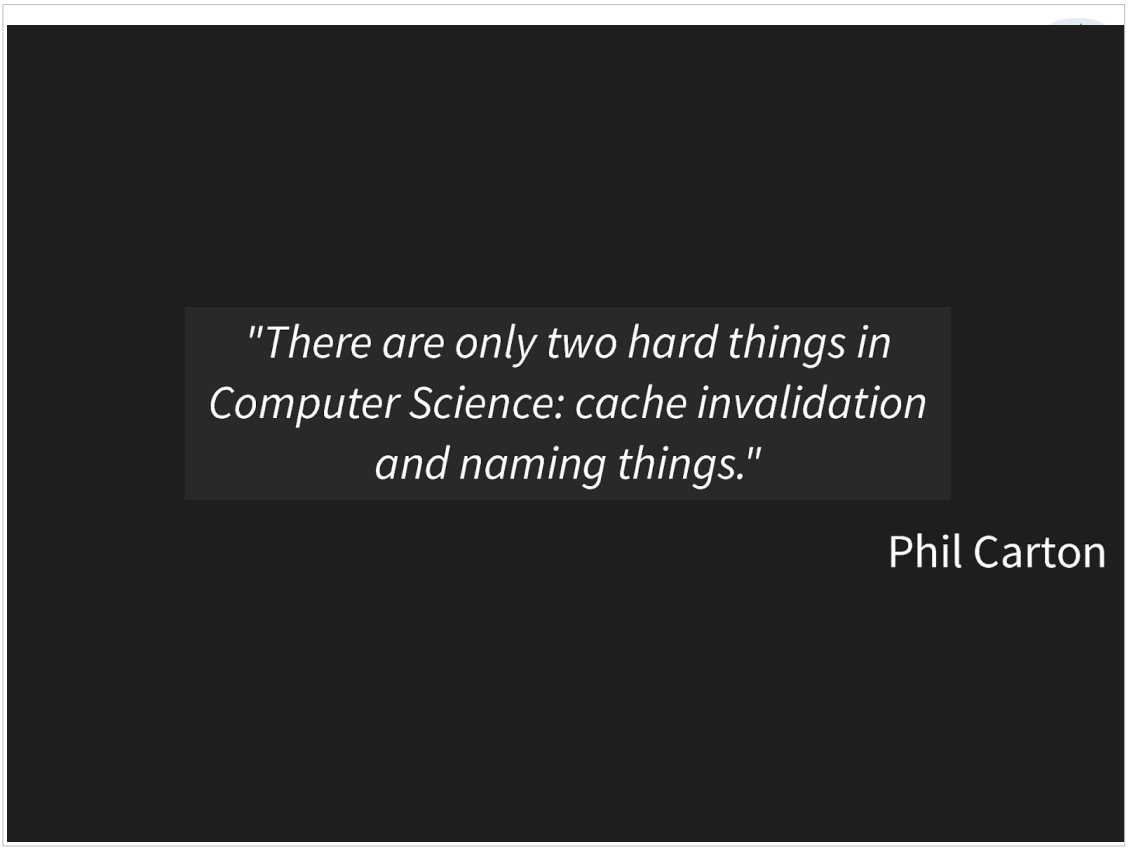


- It make sense to have a thought-through folder structure to organise files
- there is no perfect structure (no one-fits all  recommendation)
- Here is an example from BES Guide for reproducible code: should be adapted to the needs of ecologists

- each project in its own directory, which is named after the project
  - divide into projects based on the overlap in data and code files (if projects share >50% of files, put them together)
- Tools/ files to use in many project : put on their own (e.g. Literature)
- within the Project folder
  - /data : all data and metadata
    - raw data (never touch!)
    - cleaned data (intermediate data) can go to results or (in a separate folder) here.
  - /doc : text documents associated to the project.
  - manuscripts, documentation for code, lab notebook
  - maybe need a subfolder for manuscript?
  - /figs : figures and plots, publication-ready figures
  - /R : R scripts. naming with numbers can be useful, or a documentation of the order of scripts.
  - /output = /results : files generated by the scripts. statistical tables, raw figures
  - README
- additional folders : you might need additional folders for your project, depending on your needs
- /old : I personally find that useful : put all that is old, but that you don't really want to delete yet.




- working reproducible :
  - files should be openable in many years without having to purchase a specific software
  - proprietary software can
    - become unavailable.
    - when this software is no longer available, your data also will disappear
  - text files can always be read
    - .txt and .csv files can be read with many different programs
- use .csv and .txt
  - not .xlsx (the default Excel format)
  - Excel can open .csv
  - comma or semicolon ; separation, point decimal separator
- Excel : … <see above>




- This is a famous quote about programming.

- and it’s really true, naming things is something hard.
- naming things consistently
- is even harder

- Dozens of blogposts and recommendations about naming out there

# naming things

- file names :
  - meaningful
  - end in .R
  - no special characters but numbers, letters, - and _
- if files should be run in particular order, prefix them with numbers (and pad with zero).

```
# Good
fit_models.R
utility_functions.R

# Bad
fit models.R
foo.r
stuff.r
```

```
00_download.R
01_explore.R
...
09_model.R
10_visualize.R
```

# naming things

- use names to reveal intention
  - “area_rectangle” instead of “size”
- good names reduce comments

```
60 <- time_tolerance # seconds
60 <- time_tolerance_in_seconds
```

- capital letters for constants
  - BYTES_PER_GB = 1073741824
- use tab completion for names in RStudio
  - no excuse to not use long names
- use names that everyone can understand
  - avoid ambiguous names (e.g. “cat” = category or catalogue?)

# be consistent

- consistent naming
- either “index” or “idx”, not both
- consistent physical units (SI units)
- consistent style

# Take home

# Reflections

- Which of the practices will you apply?
- How much of the practices did you already know, how much were new?
- Which practices, you think, will help you the most in the future?
- Which practices seem less useful to you?

- Do you think the amount of information was too much, or too less, or just right?

# Take home

- Chose the practices that you think help you the most and focus on them
- practice version control
- keep the other practices in mind and address them after

# references and further reading

# references and resources

- the links provided in this presentation
- ETH Zürich Science IT course : "Best practices in Programming" https://sis.id.ethz.ch/services/consultingtraining/programming_best_practices.html
- Exploratories Synthesis page : https://github.com/biodiversity-exploratories-synthesis
- BES Guide for Reproducible Code : https://www.britishecologicalsociety.org/wp-content/uploads/2019/06/BES-Guide-Reproducible-Code-2019.pdf
- paper Wilson 2017 "Good enough practices in scientific computing" : https://doi.org/10.1371/journal.pcbi.1005510
- Florian Schneider resources , e.g. : basic R intro : http://fdschneider.de/r_basics/
- books of Hadley Wickham
  - R for Data Science : https://r4ds.had.co.nz/
  - Advanced R : https://adv-r.hadley.nz/
  - tidyverse : https://www.tidyverse.org/
- Introduction Course to Data Science by Brian Enquist : https://www.researchgate.net/publication/310798573_How_to_think_About_Your_Data_Introduction_to_Data_Science_Management_what_they_don%27t_-_but_should_-_teach_you_about_the_scientific_method

- **Exercises**

Mutating joins
Prices of 50,000 round cut diamonds by carat and cut
Quality of the cut
Ideal
Premium
Very Good
Good
Fair
Weight (carats)

# Exercises

try solving on your own and ask questions

- Git
- Unit testing
- generating a reproducible Example
- Divide coding problem into sub-problems
- Rmarkdown
- Renv
- **TODO**

- additional material

## why care? societal level

- Reproducibility
- FAIR principle

- Today : focus mostly on Reproducibility
- applied to code : Someone else should be able to reproduce your analysis within reasonable time and effort.
- Quick self-reflection:
- Your code, as it is today :
- Would someone else be able to reproduce it?
- (mine : hardly full analysis (not enough documentation on how to use which script). Understand and reproduce one script : yes.)

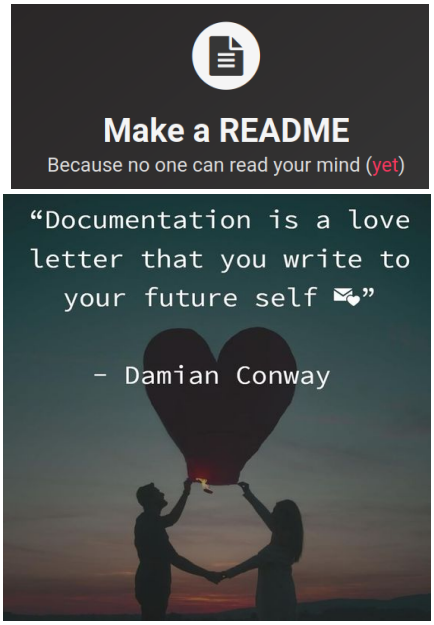
# How many R scripts should I write?

- Many!
- one script for one purpose
  - data cleaning
  - analysis
  - making figure
  - …
- → easier debugging, more flexible
- there are alternatives (debated)

- image : easier for debugging, changing things. if you want to edit a part of your analysis, just open the drawer of the step you want to edit. So you don’t have to deal with the chaos of the whole analysis, just with the chaos of the specific script.

# README

- describe the project (shortly) and your name and date
- provide basic orientation in the project folder
  - describe all column names and variables, define units
  - describe measurements
  - link relevant papers or data
- describe file naming scheme
- include answers to questions that a person trying to reproduce your project might have




Awesome README files: https://github.com/matiassingers/awesome-readme



provide a README or a METADATA file along with your data

Damian Conway : informatics professor in australia, perl programmer and wrote books about programming

# automated workflows

- what?
  - Code which can be run from input to output without you manually changing input or selecting subsections of the code.
- how ?
  - don’ t comment out parts of the script to adjust, but use if/else statements
  - use an executing script and a master script
  - if you need to do the same with several datasets : write code which is not dataset-specific, make a second script which runs the executing script for all datasets. Instead of having two scripts (D.R.Y.)



- return early : z.B. if number <1 return F, then do it in the beginning. Checking for wrong input in the beginning. → faster
- for those who don’t know yet :
  - keep in mind
  - if you want, we can look at it
  - start learning them a little bit before you feel ready

<DRY>
DRY
</DRY>
DON'T REPEAT YOURSELF

# Don’t repeat yourself

- try to write code only once
- re-use good pieces of your work
- reduce the risk of errors : you always have to update code in *all* places
- watch out for copy-pasting
- use functions

# functions

- should do what you mean
- no side effects
- describe parameters
- keep functions small
- few parameters
- return early
- for those who don’ t know how to use functions yet : keep in mind to learn it



- return early : z.B. if number <1 return F, then do it in the beginning. Checking for wrong input in the beginning. → faster
- for those who don’t know yet :
  - keep in mind
  - if you want, we can look at it
  - start learning them a little bit before you feel ready

# reduce mental overhead

- in german "mentaler Aufwand"
- Programming alone is already demanding, try to relieve your brain with a good overview.
- causes of mental overhead :
  - related code is not together
  - a lot of scrolling needed
  - little empty lines
  - no sections or visually distinguishable parts
  - very long lines (scrolling from left to right)
  - nested code (part of the code here, bt is dependent on manually running other part in different file)

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

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



you notice yourself, when you get impatient or feel overloaded when working on your code.

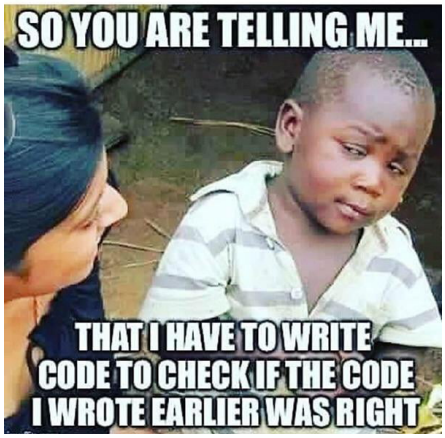
# unit testing (advanced)

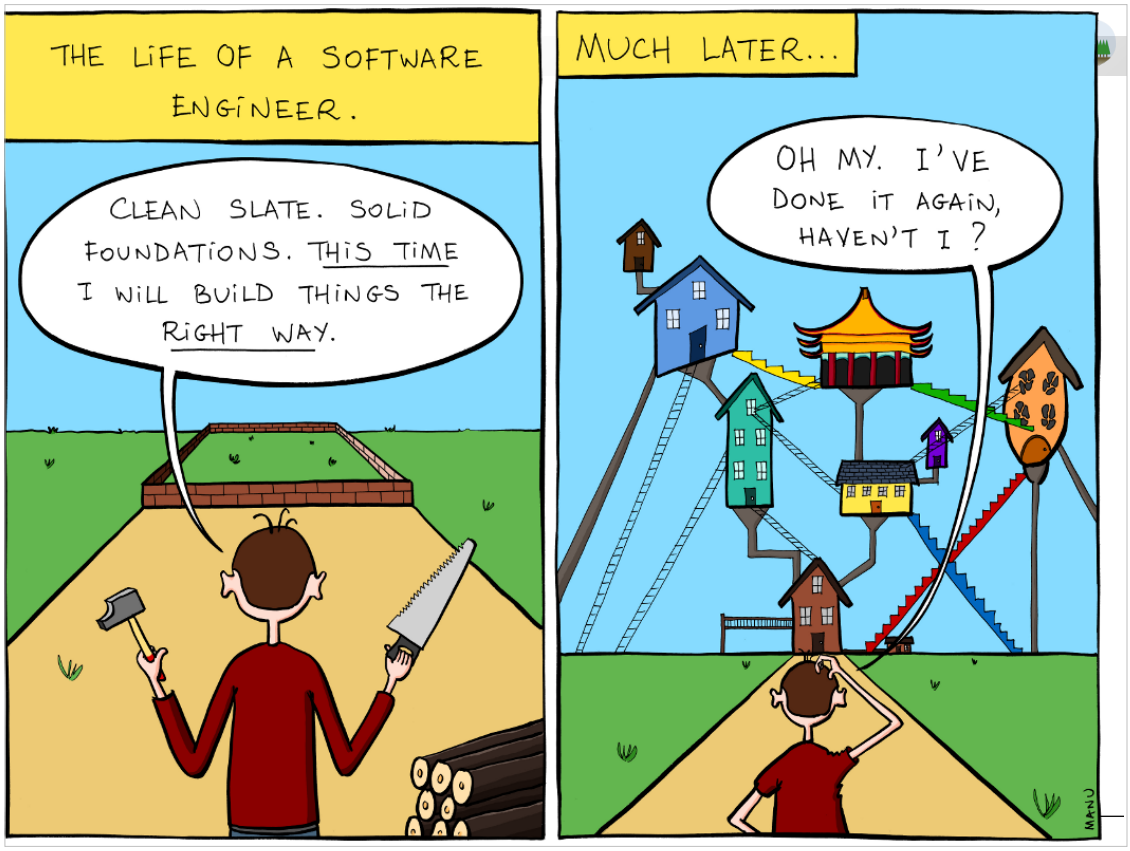
- entangled software **breaks upon tiny changes** (e.g. new R version, package updates, bug fixes, ...)
- write integrated test functions which check if your code works right
- unit testing in R :
- e.g. testthat

# unit testing (advanced)

## The Art of Software Testing, Third Edition

Editor(s): Glenford J. Myers, Tom Badgett, Corey Sandler
First published: 2 January 2012
Print ISBN: 9781118031964 | Online ISBN: 9781119202486
| DOI: 10.1002/9781119202486
Copyright © 2012 by Word Association, Inc. All rights reserved

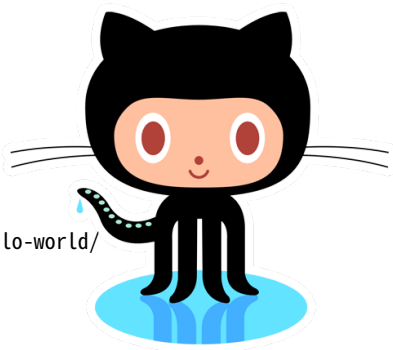



THE LIFE OF A SOFTWARE ENGINEER.
CLEAN SLATE. SOLID FOUNDATIONS. THIS TIME I WILL BUILD THINGS THE RIGHT WAY.
MUCH LATER...
OH MY. I'VE DONE IT AGAIN, HAVEN'T I ?
MANU

# code refractoring

- refractoring : going through your code and make it easier to understand without changing in- and output
  - difference to bug-fixing : there was no bug
- when should I refractor? all the time!
- things to address address :
  - cryptic names for variables: rename them
  - break long functions/ script bodies down to smaller units
  - replace algorithm if you learn a better one
  - combine duplicated pieces of code
  - add functions / loops after you learned them
  - TODO : make more advanced stuff slide, put this more basic
  - include new stuff, go over your code over again. once it works, its fine if it works, but if you want to improve you can do.

# GitHub and longtime storage



- GitHub : a possibility to store
- the versions of your file
- remotely (outside your computer)
- Guide to GitHub: https://guides.github.com/activities/hello-world/
- Connect with other programmers
- Share your code with the public
- GitHub doesn’ t offer longtime sotrage, if using it, publish finished code in other repository such as dryad or zenodo (has integration with github).