zeppelin universität

zwischen
Wirtschaft Kultur Politik

# Data Analytics and Visualisation

## I Basic R Programming

David Zimmermann

2017-04-21

Zeppelin University

# Outline

1. Math, Boolean, & Data Types
2. Control Statements
3. Basic Data Structures
4. Loops
5. Functions
6. Libraries
7. Style & Organization

# Math, Boolean, & Data Types

# R as a calculator

R can be used as a (simple) calculator, using: `+`, `-`, `*`, `/`, `^`, `sqrt()`, `log()`, `exp()`, `%%`, `%/%`

```
1 + 1 - 2
```

```
## [1] 0
```

```
42 / (2 * 3)
```

```
## [1] 7
```

```
sqrt(4 ^ 3)
```

```
## [1] 8
```

```
7 %% 3
```

```
## [1] 1
```

```
13 %/% 2
```

```
## [1] 6
```

See also: `?Arithmetic`

# Comparing Numbers

Using boolean operators: `==`, `!=`, `<`, `>`, `>=`, `<=`, `%in%`

```
4 == 2
```

```
## [1] FALSE
```

```
3 != 2
```

```
## [1] TRUE
```

```
2 < 2
```

```
## [1] FALSE
```

```
2 >= 2
```

```
## [1] TRUE
```

See also: `?Comparison`

# Comparing Numbers cont'd

Negation (reversing the outcome) can be done with !

```
!TRUE
```

```
## [1] FALSE
```

```
!FALSE
```

```
## [1] TRUE
```

```
!(2 < 4)
```

```
## [1] FALSE
```

```
"Alice" != "Bob"
```

```
## [1] TRUE
```

```
!("Alice" == "Bob")
```

```
## [1] TRUE
```

# Simple Math nesting

Evaluating multiple statements through nesting:

(using logical & AND, or | OR, also nesting using parenthesis ())

```
T & F
```

```
## [1] FALSE
```

```
T | F
```

```
## [1] TRUE
```

```
(T & F) | (T | F)
```

```
## [1] TRUE
```

```
(10 < 5) & (2 < 5)
```

```
## [1] FALSE
```

```
(10 < 5) | (2 < 5)
```

```
## [1] TRUE
```

See also: `?Logic`

# Using Variables

Assign values to a variable using the assignment-operator <- to use the data later

```
x <- 3
x
```

```
## [1] 3
```

```
x + 2
```

```
## [1] 5
```

```
x <- "Hello World!"
x
```

```
## [1] "Hello World!"
```

```
# Everything that comes after a pound-sign (#)
# is disregarded by the programm and is room for you
# to describe what and why you are doing something
```

# Data Types

R knows four* basic types:

1. `numeric`: numbers such as `1`, `-2`, `1.414`, or `3.141`
2. `character`: strings/texts such as `"Hello world!"` or `"1"`
3. `logical`: `TRUE` or `FALSE` (`T` or `F` in short)
4. `integer`: "whole" numbers `1L`, `3L`, `-12L`

Additional: `NA`s, `Inf` (?)

For the sake of completeness: `complex`, (`factor`)

# Data Types cont'd

We can use the `class`-function to check the type (class) of anything in R

```
class(3.141)
```

```
## [1] "numeric"
```

```
class("hello world")
```

```
## [1] "character"
```

```
class(TRUE)
```

```
## [1] "logical"
```

# Exercises: Math, Boolean, and Data Types

# Control Statements

# Control Statements: Idea

Controls the flow of the program/skript:

**IF** *SOMETHING* is true, then **DO** something.

Example: **If** it *rains*, then **take umbrella**.

In R:

```
if (rains == TRUE) {
  # take_umbrella
  ...
}
```

```
if (rains) {
  # take_umbrella
  ...
}
```

# IF-statements Blueprint

```
# IF-Statements
if (test) {
  # run this if test == TRUE
  ...
} else if (test2) {
  # run this if test != TRUE and test2 == TRUE
  ...
} else {
  # if neither test == TRUE, nor test2 == TRUE: run this
  ...
}
```

# IF-example

```
x <- 99
if (x < 0) {
  print("x < 0")
} else if (x < 10) {
  print("0 <= x < 10")
} else if (x < 100) {
  print("10 <= x < 100")
} else {
  print("x >= 100")
}
```

```
## [1] "10 <= x < 100"
```

# Cash Register

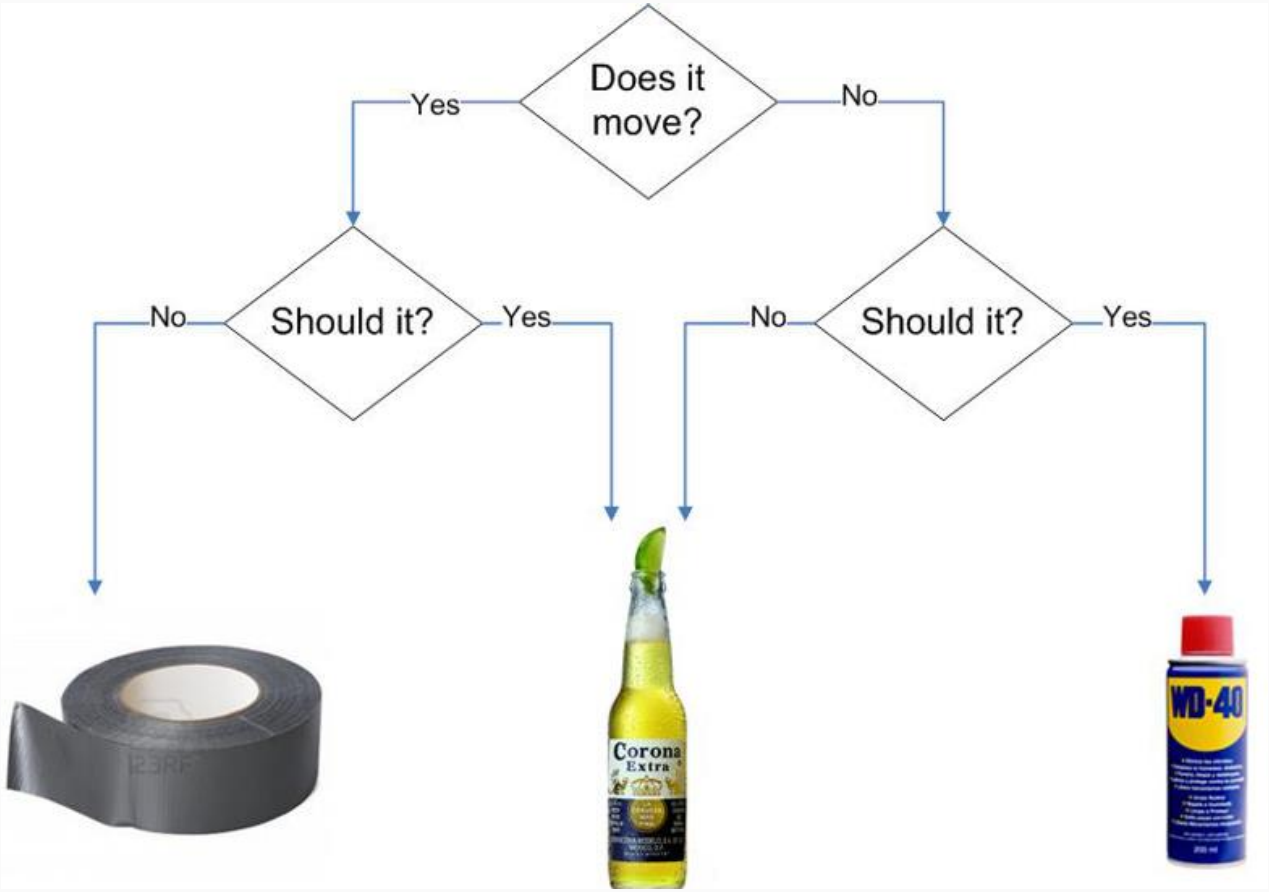
Say we have built a cash register for different products

Questions: What is the tax if we have a variable `product` that is either `book` (7% VAT) or `PC` (19% VAT).

```
product <- "book"

if (product == "book") {
  print("VAT is 7%")
} else if (product == "PC") {
  print("VAT is 19%")
} else {
  print("Unknown product")
}
```

# Engineering Flow-Chart



# Engineering Flow-Chart cont'd

```
if (does_it_move){
  if (should_it_move) {
    print("Beer")
  } else {
    print("Ducked-tape")
  }
} else {
  if (should_it_move) {
    print("WD-40")
  } else {
  print("Beer")
  }
}
```

# Engineering Flow-Chart cont'd

```
if (does_it_move && !should_it_move) {
    print("Ducked-tape")
} else if (!does_it_move && should_it_move) {
    print("WD-40")
} else {
  print("Beer")
}
```

# Exercises: If Statements

# Basic Data Structures

# Incomplete List of Data Structures

What structures does R provide us with, to organize data?

Very similar in most aspects (creation, access, etc.)

- Vector: 1D, one type
- Matrix: 2D, one type
- Array: nD, one type
- List: nD, can contain all other structures
- Data.frame: 2D, one type per column
- ...

Today: vectors, tomorrow data.frame

# Vectors

So far only single values... realistic? Solution (one-dimensional) vector

Concatenate values to vector with `c()`.

```
c(1, 3, 5, 7)
```

```
## [1] 1 3 5 7
```

Caveat: A vector has always the same class

```
c(1, 3, T, "cat")
```

```
## [1] "1"    "3"    "TRUE" "cat"
```

# Vector Length and Extending

`length()` returns the size of the vector

```
vec <- c(1, 2, 3, 52, NA, 123)
length(vec)
```

```
## [1] 6
```

Extending a vector using `c()` as well

```
vec <- c(vec, 7)
vec
```

```
## [1]   1   2   3  52  NA 123   7
```

# Accessing a Vector

How do we access only a single element of a vector? -> []

```
vec <- c(6, 4, 2, 5, 16, 7)
vec
```

```
## [1]  6  4  2  5 16  7
```

```
vec[1]
```

```
## [1] 6
```

```
vec[3]
```

```
## [1] 2
```

```
vec[7] # Out of bounds!
```

```
## [1] NA
```

# Comparing a Vector

We can compare a vector using the same operators

```
vec <- c(4, 7, 16)
```

```
vec == 7
```

```
## [1] FALSE  TRUE FALSE
```

```
vec < 5
```

```
## [1]  TRUE FALSE FALSE
```

```
vec %% 2 == 0
```

```
## [1]  TRUE FALSE  TRUE
```

```
vec %in% c(4, 16)
```

```
## [1]  TRUE FALSE  TRUE
```

# Accessing a Vector cont'd

More advanced access methods using `vec <- c(6, 4, 2, 5, 16, 7)`

```
vec[c(6, 4, 5)]
```

```
## [1]  7  5 16
```

```
vec[1:4]
```

```
## [1] 6 4 2 5
```

```
vec[c(T, T, F, F, F, T)]
```

```
## [1] 6 4 7
```

```
vec[vec > 5]
```

```
## [1]  6 16  7
```

# Vectors Sequences

Creating a sequence of numbers using : or using seq

```
1:5
```

```
## [1] 1 2 3 4 5
```

```
seq(from = 0, to = 2, by = 0.5)
```

```
## [1] 0.0 0.5 1.0 1.5 2.0
```

```
seq(0, 100, length.out = 5)
```

```
## [1]   0  25  50  75 100
```

# Comparing Values in a Vector

Checking if an element is in another vector using %in%

```
3 %in% 1:10
```

```
## [1] TRUE
```

```
0.5 %in% 0:10
```

```
## [1] FALSE
```

```
"Dave" %in% c("Alice", "Bob", "Charlie", "David")
```

```
## [1] FALSE
```

# Vector Repetitions

```
LETTERS[1:5]
```

```
## [1] "A" "B" "C" "D" "E"
```

```
rep(LETTERS[1:2], 3)
```

```
## [1] "A" "B" "A" "B" "A" "B"
```

```
rep(LETTERS[1:2], each = 3)
```

```
## [1] "A" "A" "A" "B" "B" "B"
```

# Vector Math Functions

```
# check ?Random
# creates 1000 normally-distributed random values
x <- rnorm(1000)
```

```
sum(x)
```

```
## [1] -3.846567
```

```
mean(x)
```

```
## [1] -0.003846567
```

```
sd(x)
```

```
## [1] 0.978444
```

```
summary(x)
```

```
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## -4.149000 -0.685700  0.004445 -0.003847  0.671100  3.410000
```

Also: `min()`, `max(x)`, `range(x)`, `median(x)`, `IQR(x)`, and `var(x)`

# Vector Math Functions cont'd

```r
# check ?Random
# creates 1000 uniformly-distributed random values
y <- runif(1000)
cor(x, y)
```

```
## [1] -0.01255338
```

```r
cov(x, y)
```

```
## [1] -0.00370108
```

# Vectorisation

```
vec <- 0:5
vec + 2
```

```
## [1] 2 3 4 5 6 7
```

```
vec == 2
```

```
## [1] FALSE FALSE  TRUE FALSE FALSE FALSE
```

```
vec ^ 2
```

```
## [1]  0  1  4  9 16 25
```

# Vectorisation - ifelse

```
vec <- -2:2
# vectorised if-else-statement
ifelse(vec < 0, -vec, vec)
```

```
## [1] 2 1 0 1 2
```

```
# Or nested ifelse statements
ifelse(vec < 0, -vec,
       ifelse(vec == 0, 100, vec))
```

```
## [1]   2   1 100   1   2
```

# Recap - Vector

```
x <- c(1, 2, 3)
x <- 1:3
x <- seq(from = 1, to = 3, by = 1)
x <- seq(1, 3, 1)
x <- seq(1, 3, length.out = 3)
x
```

```
## [1] 1 2 3
```

```
x[(x < 2) | (x / 2 == 1.5)]
```

```
## [1] 1 3
```

# Recap - Vector

```
LETTERS[1:10]
```

```
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J"
```

```
rep(LETTERS[1:5], each = 2)
```

```
##  [1] "A" "A" "B" "B" "C" "C" "D" "D" "E" "E"
```

```
vec_name <- c("Alice", "Bob")
paste("Hi", vec_name)
```

```
## [1] "Hi Alice" "Hi Bob"
```

# Exercises: Vector

# Loops

# For-loop

Blueprint of a for-loop, where ‘`var`’ is the name of a (not-yet used) variable and ‘`values`’ is a vector of values (all types valid)

```
for (var in values) {
  # do something
  ...
}
```

Example:

```
for (i in 1:3) {
  print(paste("i:", i))
}
```

```
## [1] "i: 1"
## [1] "i: 2"
## [1] "i: 3"
```

# For-loop cont'd

'var' and 'values' can take many forms

```
student_names <- c("Alice", "Bob", "Charlie")

for (name in student_names) {
  print(paste("Hello", name, "your name has",
              nchar(name), "letters!"))
}
```

```
## [1] "Hello Alice your name has 5 letters!"
## [1] "Hello Bob your name has 3 letters!"
## [1] "Hello Charlie your name has 7 letters!"
```

Use ?nchar

# Exercises: For-Loop

# Functions

# What is a Function?

Idea: write the code once in a function, call the function as often as needed

*Don't repeat yourself!*

Take (zero or more) inputs, do something, and produce (zero or more) outputs...

```
# define a function
foo <- function() {
  # do something
  ...
  return(result_var)
}
# call the function
foo()
```

```
# define a function
foo2 <- function(arg1) {
  # do something
  ...
  return(result_var)
}
# call the function
foo(123)
```

# On Functions

Objects are variabels for example (`x <- "hello World"`, `y = 1:10`), functions use objects (`print(x)`, `mean(y)`).

Try typing `var` into the console (without parenthesis), this is the code for the function.

**Understanding R**

To understand computations in R, two slogans are helpful:

- Everything that exists is an object.
- Everything that happens is a function call.

John Chambers

# Inputs

Define a function that takes one argument 'num', add 2 to it, and return the value:

```
add_2 <- function(num) {
  num <- num + 2
  return(num)
}
```

```
add_2(3)
```

```
## [1] 5
```

```
add_2(10)
```

```
## [1] 12
```

```
add_2(-2)
```

```
## [1] 0
```

# Multiple Inputs

Divide two inputs by each other

```
div_xy <- function(x, y) {
  result <- x/y
  return(result)
}
```

```
div_xy(10, 2)
```

```
## [1] 5
```

Also possible (naming the input and thus ordering):

```
div_xy(x = 100, y = 10)
```

```
## [1] 10
```

```
div_xy(y = 10, x = 100)
```

```
## [1] 10
```

# Default Inputs

```
add_n <- function(a, b = 10) {
  result <- a + b
  return(result)
}
```

No `b` provided, thus taking the default value of 10.

```
add_n(a = 100)
```

```
## [1] 110
```

```
add_n(100)
```

```
## [1] 110
```

# Use Case

Create a vector of random values and standardize their values (mean of zero, standard deviation of 1)

```
standardize <- function(vec){
  std_vec <- (vec - mean(vec)) / sd(vec)
  return(std_vec)
}
x <- rnorm(1000, mean = 10, sd = 5)
x_std <- standardize(x)
```

```
mean(x_std)
```

```
## [1] -1.533768e-16
```

```
sd(x_std)
```

```
## [1] 1
```

# Scoping

Each function has its own environment, thus variables are set only locally

```
x <- 42

set_x <- function(n) {
  x <- n
}
set_x(123)
x
```

```
## [1] 42
```

Although we would expect `x` to be set to 123, `x` was only changed in the function environment. Calling `x` from the main environment results in the unchanged value.

# Scoping 2

But values can (but should not) be taken from parent-environments

```
x <- 42

find_x <- function() {
  print(paste("x is", x))
  # no x in this env, go to parent env
}
find_x()
```

```
## [1] "x is 42"
```

Although there is no `x` in the functions environment, we can find `x` in the parent environment.

# Recap - Functions

```r
# Basic Function
foo <- function(){
  ...
  return(ret_values) # not necessarily needed
}
# Function Arguments
foo2 <- function(arg1, arg2){
  ...
}
# Default Parameters
foo3 <- function(arg1, argn = 3){
  ...
}
```

# Add-in - Debugging

```
## Error in if (my_test) {: missing value where TRUE/FALSE needed

## Error in eval(expr, envir, enclos): object 'sales_df' not found

## Error in foo(): argument "arg1" is missing, with no default

## Warning: You are trying to divide by zero.
```

What now?

0. Read the error message and RTFM!
1. Rubber Ducky-Method (-> Rubber Duck Debugging)!
2. Recreate the error with a Minimum Working Example (MWE)!
3. Google is your friend!!!!11
4. Ask friends/collegues/supervisors

MWE:

- https://stackoverflow.com/help/mcve
- https://stackoverflow.com/a/5963610/3048453

# Exercises: Functions

# Libraries

# Libraries

Libraries are R's way to distribute code, data, and functions.

Example: Calculate the value of an option; Make or buy(?) decision

Usage:

1. Install the package (only once) with `install.packages("myPackageName")`
2. Load the package/library (only at the top of the script or the beginning of each session) with `library(myPackageName)`

# Libraries cont'd

3. Use functions, data, and help (every(!) package has a website: i.e., https://cran.r-project.org/web/packages/tibble/index.html)

```
> tibble()
Error: could not find function "tibble"
```

Error? Usually package not loaded or misspelled the name of the function

# Libraries CRAN

Where do I find appropriate libraries?

- CRAN (Comprehensive R Archive Network): https://cran.r-project.org/
- CRAN Task Views: https://cran.r-project.org/web/views/
- Blogs: https://www.r-bloggers.com/
- Google. . . . (try "r option pricing")

# Exercises: Libraries

# Style & Organization

# Coding Style

*Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live. Code for readability. (John F. Woods)*

# Coding Style cont'd

- Use common sense and be consistent, i.e., use '`lower_snake_case`', '`camelCase`', '`period.case`', but '`Not.Like_thisWhichNoOne_can.Read`'
- Names have a reason '`some_var`' is not as useful as '`length_of_dt`'
- Clean code is readable! '`x=(3+2)/2`' is bad, instead you could use '`x <- (3 + 2) / 2`' (Thereisareasonweusewhitespacesintexts)
- Comment your code, frequently and extensively (it might save you in the future), see next slide
- More Info: http://adv-r.had.co.nz/Style.html
- Alt. Style: https://google.github.io/styleguide/Rguide.xml

# Coding Style cont'd

Code commenting using `roxygen2`-template (place the cursor in your function and hit: Cmd/Ctrl + Alt + Shift + R)

```
#' This is where the title of my function goes
#'
#' Here I describe in length (yet with the KISS-principle) what the
#' function does
#'
#' @param date a description of the argument needed
#' @param user another description of the second arg
#'
#' @return a description of what is returned
#' @export
#'
#' @examples
#' my_fun(date = as.Date("2001-01-01"), user = "David")
my_fun <- function(date, user){
  
}
```

# Project Organization

Organize yourself and be consistent (#1 goal!)

1. Folder structure: seperate folders for data, exports (pictures, tables, etc.), R-files, etc.

```
.
+-- data
|   +-- raw_data.csv
|   +-- tidy_data.csv (created by tidy_data.R)
+-- tex-files
|   +-- plots
|       +-- plot1.pdf (created by plot1.R)
|   +-- tables
|       +-- table1.tex (created by table1.R)
|   +-- main.tex
|   +-- ... (other tex-files)
|   +-- my_bib_file.bib
|   +-- main.pdf
+-- R
|   +-- functions.R (containing all functions used)
|   +-- tidy_data.R (clean and tidy the data)
|   +-- plot1.R (creates plot1.pdf)
|   +-- table1.R (creates table.tex)
+-- R_projects_data.RProj
+-- README.MD (this file)
+-- README.html (this file in HTML)
(+-- Makefile)
```

A typical file(folder)-structure for a reproducible research

Pro tip: Write a `README.txt` file, where you describe what you are doing, what file-system you have, what is needed for redoing the calculations etc.

# Project Organization cont'd

2. Use scripts: one script for functions, separate data wrangling from analysis/visualizations (`source()` is your friend -> `?source`). This allows you to easily find parts and redo calculations. I.e.,

`functions.R`

```
myFun <- function(){
  ...
}
```

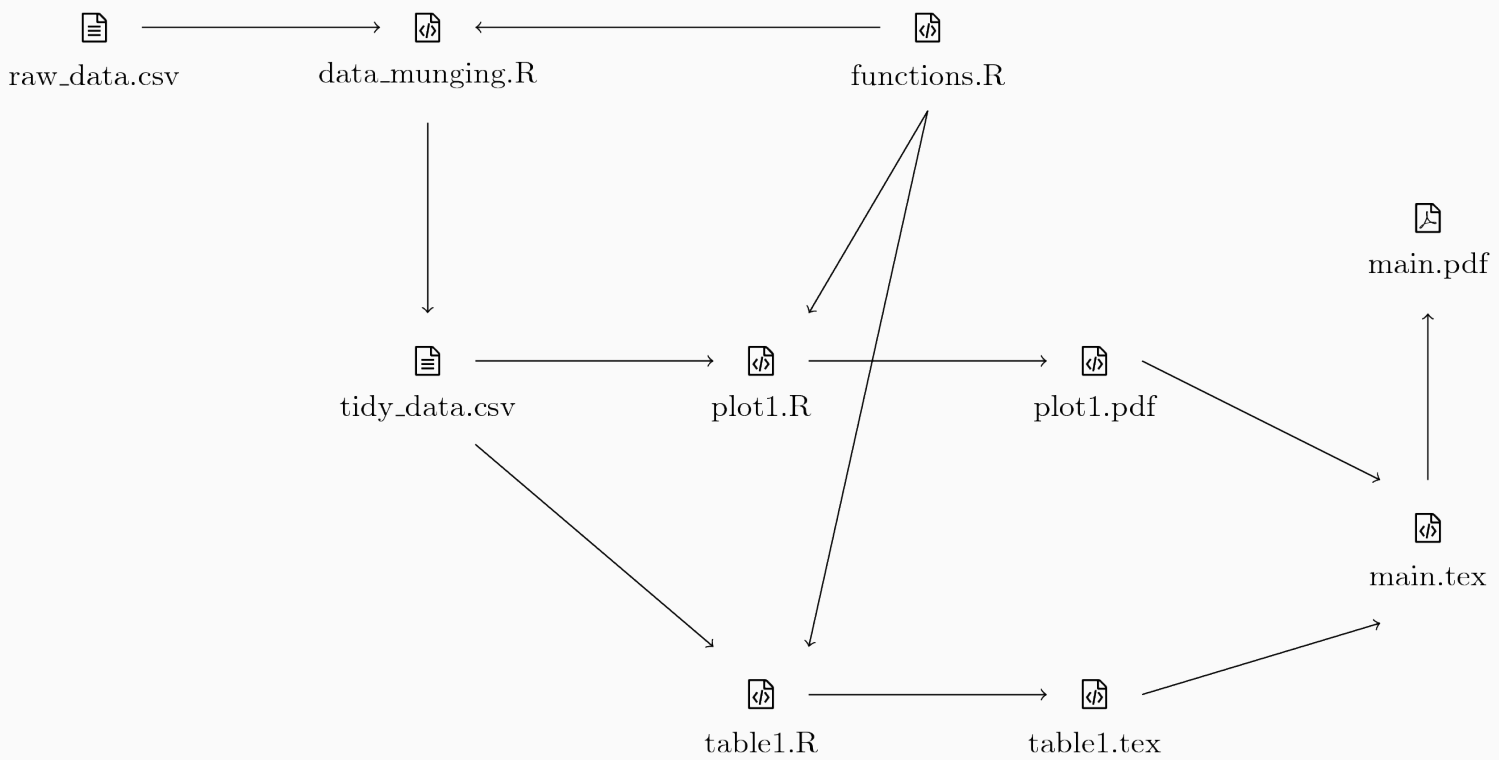
`myAnalysis.R`

```
source("functions.R")

myFun()
```

# Project Organization cont'd

3. Write for reproducibility (expect to do every step $>$ 10 times)




Organizational Workflow in a typical reproducible research project

# Working Directories and Projects

Working Directories (where the program looks for files and where it saves them)

```
getwd() # get working directory

setwd("C:/.../myProject/") # set working directory
setwd("~/.../myProject/")
```

Use R-projects in RStudio (topright)!!!

Working directory is always where the project is saved

# Exercises: Style & Organization

# Additional Information aka. Appendix

# Useful links for base-r

- https://www.rstudio.com/wp-content/uploads/2016/09/r-cheat-sheet-1.pdf
- http://www.statmethods.net/
- http://www.cookbook-r.com/
- http://www.urfie.net/read/mobile/index.html
- http://www2.warwick.ac.uk/fac/sci/statistics/staff/academic-research/reed/rexercises.pdf