

# Fundamentals of Programming for Digital Health - Part 1: R



Berry Boessenkool & Team  
[Berry.Boessenkool@hpi.de](mailto:Berry.Boessenkool@hpi.de)

use freely, cite



2022-12-04 19:07

- 1. Intro
- 2. Basics
- 3. Objects
- 4. Data
- 5. Graphics
- 6. Flow control

- 1.1 Welcome
- 1.2 Five-minutes showcase
- 1.3 Configuration: R & Rstudio
- 1.4 Exercises

We are here for you:

- ▶ **Berry**: Geoecology @ Potsdam university, lecturer @ HPI
- ▶ R fan since 2010
- ▶ R-packages  , training & consulting  , community 

---

- ▶ **Bert**: Professor for digital health - connected healthcare @ DHC
- ▶ Pervasive healthcare, wearable computing laboratory @ ETH Zurich
- ▶ Computer engineering department at Bosphorus university



# Fundamentals of Programming for Digital Health

## Part 1: **R** (Oct-Dec 2022)

- ▶ intro to programming with R
- ▶ basics in the first weeks, applications in later weeks
- ▶ weekly lectures + exercises
- ▶ tutorial sessions with exercises

## Part 2: **Python** (Jan-Feb 2023) -> similar structure

### **Exams** (each 50% of final grade)

- ▶ structurally like the (tutorial) exercises, just bigger (90 min)
- ▶ open book: static resources OK. Chats / asking on forums not OK.
- ▶ R midterm: **Thu 2022-12-15**
- ▶ Python final: **Thu 2023-02-09**

Ask **Questions** in the openHPI [course forum](#) [Discussions]



- 1. Intro
- 2. Basics
- 3. Objects
- 4. Data
- 5. Graphics
- 6. Flow control

- 1.1 Welcome
- 1.2 Five-minutes showcase
- 1.3 Configuration: R & Rstudio
- 1.4 Exercises

## Showcase - preview to the benefits of this course

- ▶ 5 minutes exemplary usage of R and Rstudio
  - ▶ 'live coding', i.e code development is shown in a script
  - ▶ the next slide contains the code for reference
- 
- ▶ the goal is not to understand all of it,
  - ▶ but to generate an 'appetite' for R :)
  - ▶ in 7 weeks, you can do all this too
- 
- ▶ sit back, relax, enjoy the show...



## Showcase Code

```
# R as calculator: -> "meaningful" result
7 * 6

# create objects to later work with them
age <- 33
age + 1          # next year, I am this old
2022 - age      # remind me when I was born, in case I forget

# read data into R:
weather <- read.table(file="weather.txt", header=TRUE)

# select a column from a data table:
weather$Rain

# Scatterplot für correlation (not necessarily causation):
plot(weather$Sun, weather$Temperature, col="orange", pch=16,
     main="lots of sun on warm days")

# Time series for development / seasonality / trend:
weather>Date <- as.Date(weather>Date, format="%Y-%m-%d")
plot(weather>Date, weather$Pressure, type="l", col="salmon", lwd=2, xaxt="n")

# Packages by other R users:
library(berryFunctions)
monthAxis()
```



- 1. Intro**
  - 2. Basics**
  - 3. Objects**
  - 4. Data**
  - 5. Graphics**
  - 6. Flow control**
- 
- 1.1 Welcome**
  - 1.2 Five-minutes showcase**
  - 1.3 Configuration: R & Rstudio**
  - 1.4 Exercises**

## Why R - and how to get it



- ▶ programming language for data analysis / visualization and statistics in research and business
- ▶ free, open source (comprehensible, reproducible, expandable)
- ▶ large user community (many methods)
- ▶ makes your work efficient, productive and replicable

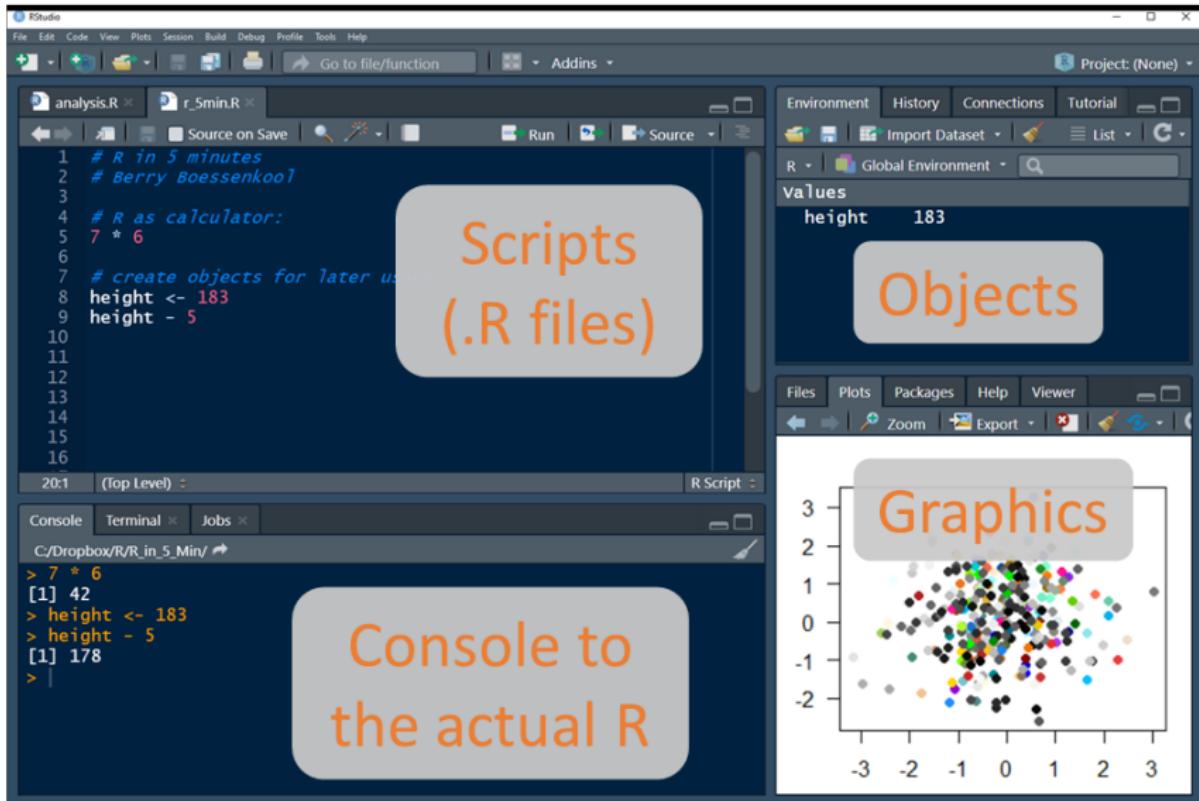


- ▶ Integrated Development Environment (IDE) for R: environment to develop code, has R integrated

## Installation

- ▶ [instructions](#)
- ▶ please use a current R [version](#) ( > 4.0, 2020)

# Overview Rstudio



# Script vs Console

The screenshot shows the RStudio interface. On the left, the 'Code' tab of the script editor is open, displaying the following R code:

```
1 # R in 5 minutes
2 # Berry Boessenkool
3
4 # R as calculator:
5 7 * 6
6
7 # create objects for later usage:
8 height <- 183
9 height - 5
10
11
12
13
14
15
16
```

The status bar at the bottom indicates '20:1 (Top Level)'. Below the editor is the 'Console' tab, which shows the execution of the code:

```
C:/Dropbox/R/R.in.5.Min/
> 7 * 6
[1] 42
> height <- 183
> height - 5
[1] 178
>
```

A large orange callout box labeled 'Script: write code' points to the script editor area. Another callout box labeled 'Code is executed here + results displayed' points to the console output.

In the top right corner, the 'Project' dropdown shows '(None)'. The 'Environment' tab in the top right pane displays the variable 'height' with the value '183'. In the bottom right pane, there is a scatter plot with numerous colored data points.

# Code can be executed directly in the console

The screenshot shows the RStudio interface. In the top-left pane, there are two open files: 'analysis.R' and 'r\_5min.R'. The 'analysis.R' file contains the following R code:

```
1 # R in 5 minutes
2 # Berry Boessenkool
3
4 # R as calculator:
5 7 * 6
6
7 # create objects for later usage:
8 height <- 183
9 height - 5
10
11
12
13
14
15
16
```

In the bottom-left pane, the 'Console' tab is active, showing the results of the code execution:

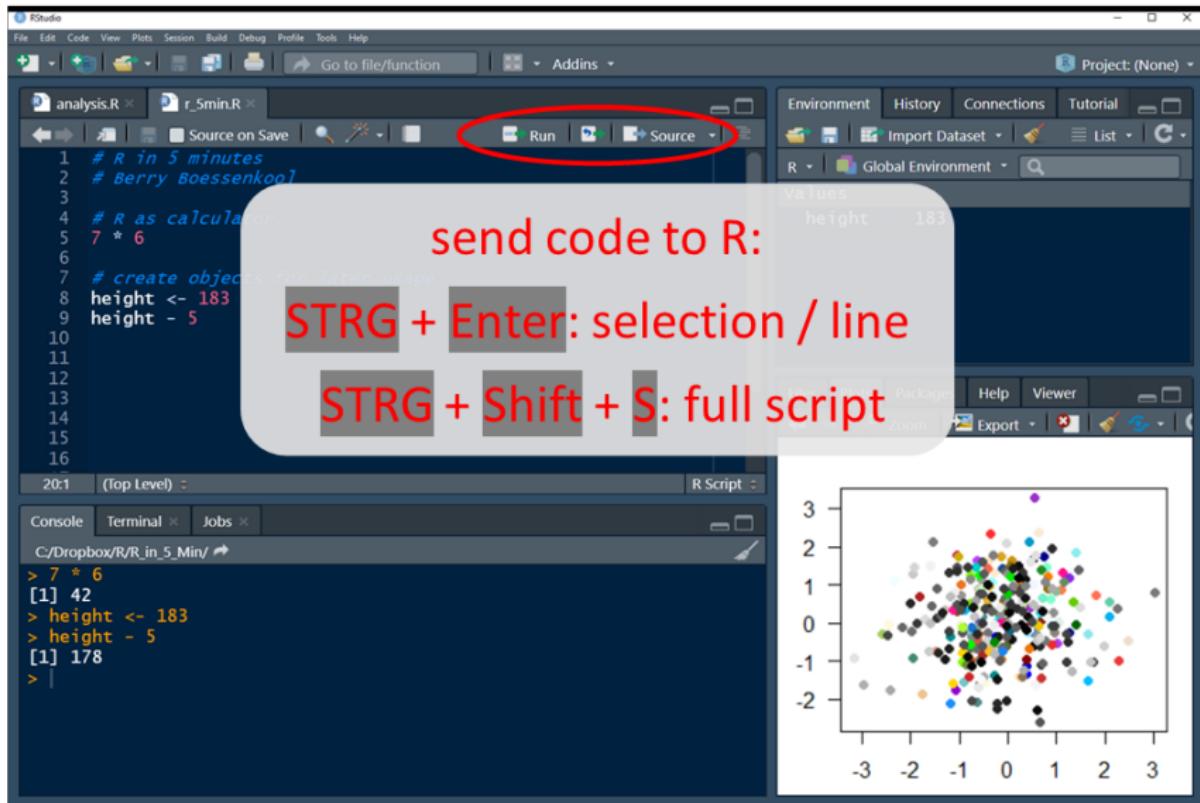
```
C:/Dropbox/R/R.in.5.Min/
> 7 * 6
[1] 42
> height <- 183
> height - 5
[1] 178
>
```

A large, semi-transparent callout bubble with rounded corners is overlaid on the console area. It contains the text 'keep longer' on the top line and 'quickly check' on the bottom line, separated by a horizontal dashed line.

The top-right pane shows the 'Environment' tab with a list of values. The 'height' variable is defined with the value 183.

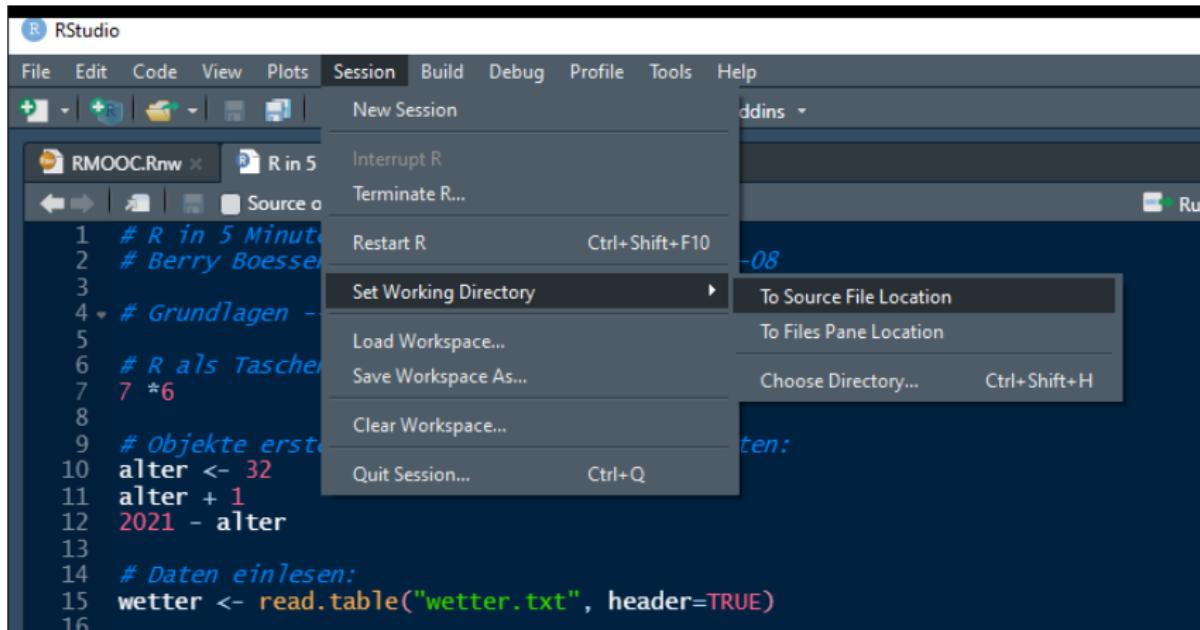
The bottom-right pane displays a scatter plot with data points colored by category, plotted against two axes ranging from -3 to 3.

## execute a line of code

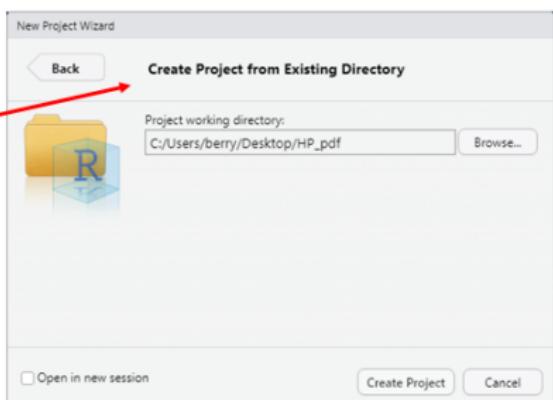
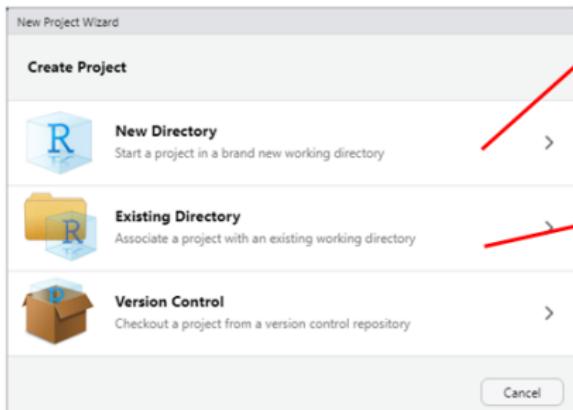
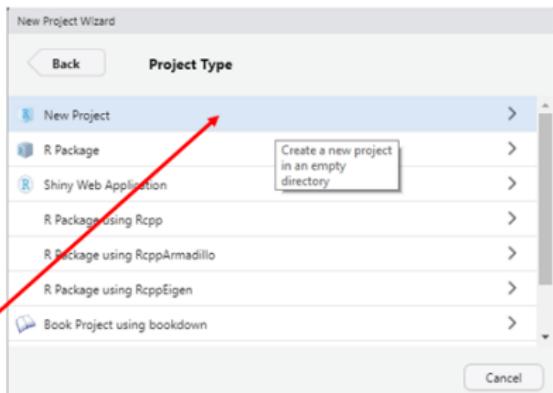


## working directory (where R reads and writes files)

for simple scripts:



# R-projects make everything easier!



## Summary for 1.3 Configuration: R & Rstudio

### **configure your R working environment:**

- ▶ install and use R & Rstudio
- ▶ scripts
- ▶ run lines of code
- ▶ use R-projects to avoid path hassle

## configure RStudio

If you want your work to be reproducible, set

**Rstudio - Tools - Global Options - General**

**OFF**: Restore .Rdata into workspace at startup

Save workspace to .RData on exit: **NEVER**

Further helpful [Rstudio settings](#)

The next slide contains a curation of keyboard shortcuts from

[rviews.rstudio.com/categories/tips-and-tricks](https://rviews.rstudio.com/categories/tips-and-tricks)

## Rstudio keyboard shortcuts ( **ALT + SHIFT + K** )

**CTRL** + **ENTER** (in source script) send line/selection to console

**UP / DOWN** (in console): previous commands

**CTRL** + **UP** (in console) command history

**CTRL** + **SHIFT** + **P** rerun previous code region (with changes)

**CTRL** + **SHIFT** + **S** / **ENTER** source entire document

**CTRL** + **SHIFT** + **N** create new R script

**CTRL** + **O** / **S** open file / save script

**CTRL** + **TAB** next Tab    **CTRL** + **SHIFT** + **TAB** previous Tab

**ALT** + mouse for multiline cursor. or: **CTRL** + **ALT** + **UP / DOWN**

turn off Windows screen rotation: Desktop rightclick - Graphics options - hotkeys - Disable

**ALT** + **UP / DOWN** move line of code up/down in script

**CTRL** + **1** cursor to panel (1:source, 2:console, 3:help, 4:history, 5:files, 6:plot...)

**CTRL** + **SHIFT** + **1 / 2 / 3 / ...** panel full view

**CTRL** + **SHIFT** + **C** (un)comment lines

**CTRL** + **SHIFT** + **O** Document outline ( **# section ----** )

**CTRL** + **SHIFT** + **M** insert pipe operator



- 1. Intro
- 2. Basics
- 3. Objects
- 4. Data
- 5. Graphics
- 6. Flow control

- 1.1 Welcome
- 1.2 Five-minutes showcase
- 1.3 Configuration: R & Rstudio
- 1.4 Exercises

## Exercises on CodeOcean - access

Each section in this course is accompanied by interactive coding exercises that consist of 5 - 15 tasks with increasing complexity. They are accessible through the openHPI platform (not per URL directly, because the login happens through openHPI).

### Exercise 1

[Edit item](#) [Statistics](#)

#### Instructions:

Click the button below to launch the exercise.

 This is a graded exercise.  10.0 points

 [Launch exercise tool](#)

They can be solved in the browser or (better): downloaded and solved in Rstudio.

## Exercises on CodeOcean - run

Once the task has opened, you can write in the scripts and execute them by clicking 'Run' (ALT + R)

The screenshot shows the CodeOcean exercise interface for an R exercise master template. At the top, there's a navigation bar with links for EXERCISES, R EXERCISE MASTER TEMPLATE, and IMPLEMENT. Below the navigation is a toolbar with a 'Run' button, a 'Score' button, and a 'Request Comments' button. To the right of the toolbar is a progress bar showing 0% completion and a '(Show)' link. On the left, there's a sidebar titled 'Collapse Action Sidebar' containing a 'Files' section with three files listed: examples\_1.R, examples\_2.R, and t\_dataset.txt. The main area contains the R script template:

```
1 # Structure of task files
2
3 # Task 1 -----
4 # Create an object named 'my_first_object' with the
5 # number 99.
6
7 # Make sure to save the script (CTRL + S) before running
8 # the following:
8 codeoceanR::rt_score()
9 # You can conveniently save + source (= run full script
# including previous line)
10 # by clicking on "Source" (topright if script window) or
# pressing CTRL + SHIFT + S.
11
12
13
14 # Task 2 -----
15 # Create another object with the integer values from 5
# till 15.
16 # The desired objectnames is already included for your
# convenience.
17 # Replace the zero with the intended code for the
# solution.
18 my_second_object <- 5:15
19 my_second_object
20
21
22 # Now continue in examples_2.R
23
```

To the right of the script, there's a 'Collapse Output Sidebar' button and a code output window showing the results of the 'rt\_score()' function:

```
> codeoceanR::rt_score()
NULL
> my_second_object <- 5:15
> my_second_object
[1] 5 6 7 8 9 10 11 12 13 14 15
```

## Exercises on CodeOcean - score

As often as you like, you can see if your solutions are correct by clicking 'Score' (ALT + S)

ster template

The screenshot shows the CodeOcean interface. At the top, there's a toolbar with 'Run', 'Score' (highlighted in orange), 'Request Comments', and a 'Keyboard shortcut: ALT + s'. To the right of the toolbar are '100%' and '(Show)' buttons. Below the toolbar is a code editor window containing R code. The code includes comments for task structures and a note about saving the script before running. It also includes a section for Task 2, creating another object with integer values from 5 till 15. A 'my\_second\_object' variable is defined with the value '5:15'. At the bottom of the code editor, a note says '# Now continue in examples\_2.R'. To the right of the code editor is a results panel titled 'Results'. It shows a message '1 test files have been executed.' Below this is a green box for 'Test File 1 (examples\_tests.R)'. Inside the box, under 'Passed Tests', it says '8 out of 8'. Under 'Score', it says '8 out of 8' and 'Feedback: Well done. All tests have been passed.' Under 'Error Messages', there is no content. At the bottom of the results panel, the text 'Score: 100%' is displayed above a green progress bar.

```
1 # Structure of task files
2
3 # Task 1 -----
4 # Create an object named 'my_first_object' with the
5 # number 99.
6 my_first_object <- 99
7
8 # Make sure to save the script (CTRL + S) before running
# the following:
9 codeoceanR::rt_score()
10 # You can conveniently save + source (= run full script
# including previous line)
11 # by clicking on "Source" (topright if script window) or
# pressing CTRL + SHIFT + S.
12
13
14 # Task 2 -----
15 # Create another object with the integer values from 5
# till 15.
16 # The desired objectnames is already included for your
# convenience.
17 # Replace the zero with the intended code for the
# solution.
18 my_second_object <- 5:15
19
20
21 # Now continue in examples_2.R
22
```

Score many times, as the messages get increasingly specific as you get closer to the intended solution. If an overview on openHPI is wanted:  
After 'Score', click 'Submit' to transfer the score to openHPI.

## Exercises in Rstudio - setup

Solving the exercises in Rstudio enables you to work in your regular R work environment, run a line / selection of code, enjoy autocompletion, partially work offline (except for scoring), use keyboard shortcuts, appreciate debugging tools, get integrated graphics output, help, package management, version control and more.

To solve exercises in Rstudio, install the package `codeoceanR` (once only): [instructions](#)



## Exercises in Rstudio - download

for each exercise:

- ▶ 1. through OpenHPI, go to the CodeOcean exercise
- ▶ 2. download it to a good location on your PC (unzip optional)

➤ R exercise master template

The screenshot shows the CodeOcean interface. On the left, there's a sidebar with a 'Collapse Action Sidebar' button and a 'Download' button. Below that is a 'Files' section containing three files: 'examples\_1.R', 'examples\_2.R', and 't\_dataset.txt'. A small lock icon is next to 't\_dataset.txt'. On the right, there's a 'Run' button and a 'Score' button. The main area displays a block of R code with line numbers from 1 to 11. Lines 1 through 4 are visible, followed by a large grayed-out area from line 6 to line 10, which contains instructions about saving the file. Line 11 is also partially visible.

```
1 # Structure of task
2
3 # Task 1 -----
4 # Create an object i
number 99.
5
6
7 # Make sure to save
the following:
8 codeoceanR::irt_scor
9 # You can conveniently
including previous
10 # by clicking on "S"
pressing CTRL + SH
11
```

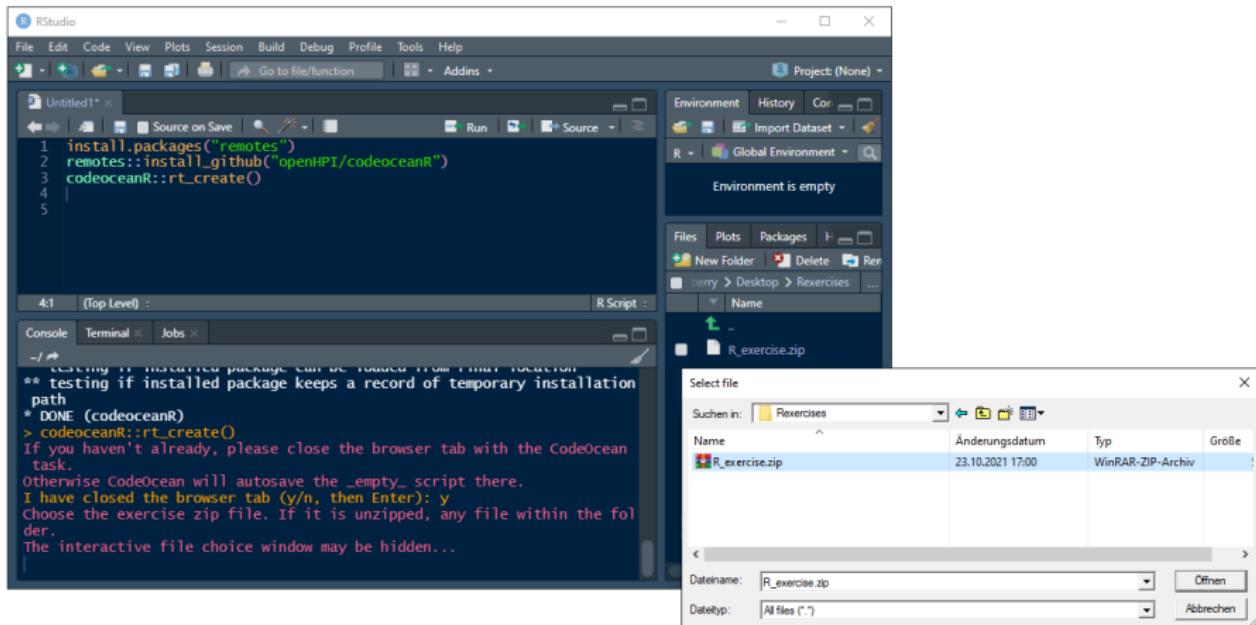
Occasionally, CodeOcean signals the error "Sorry, something went wrong". If the download starts, ignore it. Otherwise refresh the page.

- ▶ 3. close the CodeOcean browser tab (so CodeOcean doesn't autosave the unsolved script there)



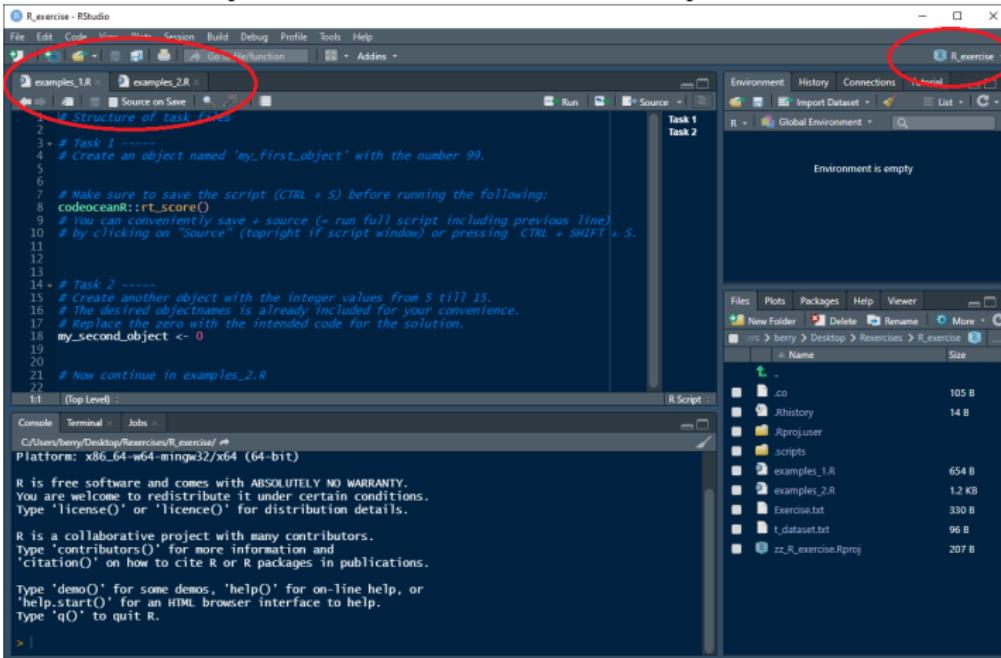
## Exercises in Rstudio - start

- ▶ 4. in R / Rstudio, run  
`codeoceanR::rt_create()`
- ▶ - confirm to have closed the tab
- ▶ - select the exercise file (if unzipped, any file within the folder)



## Exercises in Rstudio - solve tasks

`rt_create` should open a new Rproject in a separate Rstudio instance with the exercise script files already opened.  
Contact Berry if this does not work for you.



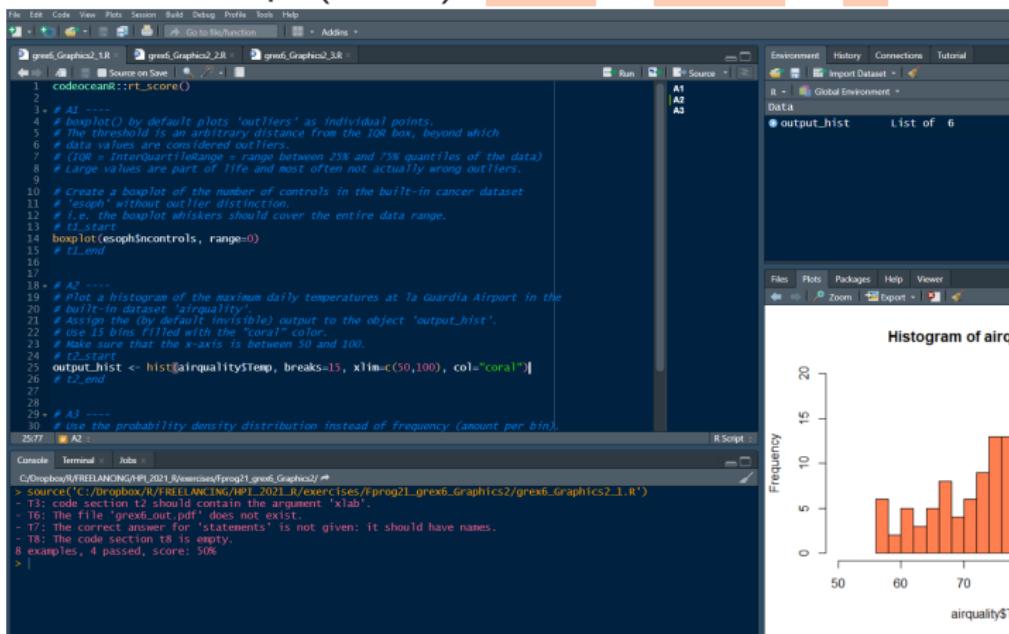
You can reopen the exercise anytime by opening the `zz_*.Rproj` file.

## Exercises in Rstudio - score

codeoceanR::rt\_score()

sends your code to CodeOcean (it will be visible if you reopen an exercise in the browser), runs the test script and shows you the feedback messages in Rstudio.

Easiest: run entire script (source): **CTRL** + **SHIFT** + **S**



The screenshot shows the RStudio interface with several panes:

- Code Editor:** Displays the R code for exercises A1, A2, and A3. The A2 section contains the following code:

```
1 # codeoceanR::rt_score()
2
3 # A1 ----
4 # boxplot() by default plots 'outliers' as individual points,
5 # The threshold is an arbitrary distance from the IQR box, beyond which
6 # data values are considered outliers.
7 # IQR = Interquartile range = range between 25% and 75% quantiles of the data
8 # Large values are part of life and most often not actually wrong outliers.
9
10 # Create a boxplot of the number of controls in the built-in cancer dataset
11 # 'esoph' without outlier distinction,
12 # i.e., the boxplot whiskers should cover the entire data range.
13
14 boxplot(esoph$controls, range=0)
15 # End
16
17
18 # A2 ----
19 # Plot histogram of the maximum daily temperatures at La Guardia Airport in the
20 # built-in dataset 'airquality'.
21 # Assign the (by default invisible) output to the object 'output_hist'.
22 # Use 15 bins filled with the "coral" color.
23 # Make sure that the x-axis is between 50 and 100.
24 # t2_start
25 output_hist <- hist(airquality$Temp, breaks=15, xlim=c(50,100), col="coral")
26 # t2_end
27
28
29 # A3 ----
30 # use the probability density distribution instead of frequency (amount per bin).
```
- Console:** Shows the command `codeoceanR::rt_score()` being run and its output, including feedback messages for each exercise.
- Environment:** Shows the global environment with an object `output_hist`.
- Plots:** Displays a histogram titled "Histogram of airq" showing the frequency of maximum daily temperatures (airquality\$Temp) from 50 to 100. The x-axis is labeled "airquality\$Temp" and the y-axis is labeled "Frequency".

### interactive exercises:

- ▶ on CodeOcean, with hidden score script testing your solutions
- ▶ in Rstudio: download exercise, close browser tab, run  
`codeoceanR::rt_create()`
- ▶ score many times, submit only once
- ▶ scoring messages get increasingly specific



- 1. Intro
- 2. Basics
- 3. Objects
- 4. Data
- 5. Graphics
- 6. Flow control

- 2.1 Syntax
- 2.2 Help
- 2.3 Vectors
- 2.4 Statistics
- 2.5 Functions
- 2.6 Logic
- 2.7 Character strings
- 2.8 Categories
- 2.9 Packages



## Operators I: arithmetic

```
4*9  
## [1] 36
```

Code in these slides is in grey boxes, with syntax-highlighting.  
R output is prefixed with `##`.

`[1]` will be explained in lesson 2.3 (vectors).

Comments (following a hashtag) are ignored by R.  
They improve readability of source code for humans:

```
2 + 4.8 # Period as decimal separator  
## [1] 6.8
```

```
2+4.8 # R doesn't mind spaces, they help humans to read
```

```
3^2  
## [1] 9
```



## Operators II: root, magnitude, logarithm, exponential function

```
sqrt(81) # square root  
## [1] 9
```

```
abs(-12) # absolute value  
## [1] 12
```

```
log(100) # natural logarithm (ln) with base e (2.72)  
## [1] 4.60517
```

```
log10(100) # logarithm with base 10  
## [1] 2
```

```
exp(3) # exponential function e^3  
## [1] 20.08554
```



Assignment: create an object with data in the workspace

```
age <- 15.4
```

Rstudio: press keys **ALT** + **-** for **<-**

```
age # now exists in the workspace (quasi R storage)
```

```
## [1] 15.4
```

```
age + 5
```

```
## [1] 20.4
```

`age` itself has not changed (is still 15.4). To change it, overwrite it:

```
age <- 37.1
```

```
age # always the current version, no history attached
```

```
## [1] 37.1
```

note lower/UPPER case:

```
Age # is not an existing object
```

```
## Error: object 'Age' not found
```

"objects" are called "variables" in many programming languages, but can be constants too, hence "object" is the better word.

## Objects

```
ls() # list custom objects in the workspace
## [1] "age"

rm(age) # remove (delete) an object
```

```
pi # built-in constant
```

```
## [1] 3.141593
```

```
pi <- 3 # you can have your own copy,
```

```
sin(pi/2) # but then you don't get 1 as intended...
```

```
## [1] 0.997495
```

Recommendation: do not use existing objects/functions like `pi`, `sin`

as names. If a personal object `pi` exists, the built-in `pi` is not used.

Good object names are **short but meaningful**, e.g. `tempMaxBerlin`

The usual conventions are `lowerCamelStandard` and

`underscore_standard`. Do not use the old `point.standard`

anymore. It has a special meaning in other programming languages.



## Overview: syntax of R functions

Functions are called (executed) with round brackets:

```
log(7.4) # function call  
## [1] 2.00148
```

```
log(x=7.4) # explicitly named argument  
## [1] 2.00148
```

Arguments have names. These can be omitted as long as they are in the right order.

`log` has another argument `base`. If this is not specified (as before), 2.718 will be used. This is the default value for `base`. For a user-defined base:

```
log(x=200, base=12) #  $12^{\wedge}2.1322 = 200$   
## [1] 2.1322
```

Argument names can be abbreviated, as long as they are unique:

```
log(200, b=12)  
## [1] 2.1322
```



## Summary for 2.1 Syntax

### syntax, objects, operators, functions:

- ▶ `+ , - , * , / , ^`
- ▶ spaces and comments (`#`) make code easier to read for humans
- ▶ `pi , sqrt , abs , log , log10 , exp`
- ▶ objects: `ALT + -` for assignment arrow (`<-`)
- ▶ object names are case sensitive
- ▶ `ob_ject` or `obJect`, not: `pi , Data , ....`
- ▶ `ls , rm`
- ▶ `function(1, argument=2, arg=3)`

Report unclear tasks in the forum

Highlight the topics from this lesson in your RefCard



## Operators 2.0 ; exponential representation

```
sin(15 * pi/180) # degrees to decimal
## [1] 0.247404

factorial(5) # n! = 1*2*3*4*...*n
## [1] 120

exp(1) # Euler's number e
## [1] 2.718282

e^3 # not possible
## Error: object 'e' not found

3.91 * 10^-3 # scientific notation: 3.91 E-3
## [1] 0.00391

3.91e-3 # no spaces allowed around 'e'
## [1] 0.00391

1e6 # quickly write a million (without 6 zeros)
## [1] 1e+06

options(scipen=9) # up to 1e9 from now on written in full
1e6 ; 1e14
## [1] 1000000
## [1] 1e+14
```



## create objects

Technically, `=` can be used instead of `<-` for assignments.

According to [style guides](#), `=` should only be used for

```
function(arg="value") .
```

`median(x <- 1:10)` also creates `x` in the `globalenv()` workspace,

`median(x=1:10)` does not.

[blog.revolutionanalytics.com](http://blog.revolutionanalytics.com), [csgillespie.wordpress.com](http://csgillespie.wordpress.com)

- 1. Intro
- 2. Basics
- 3. Objects
- 4. Data
- 5. Graphics
- 6. Flow control

- 2.1 Syntax
- 2.2 Help
- 2.3 Vectors
- 2.4 Statistics
- 2.5 Functions
- 2.6 Logic
- 2.7 Character strings
- 2.8 Categories
- 2.9 Packages



## Documentation of built-in functions

```
help("append") # open the docs of the function 'append'
```

```
?append # quick variant with less typing
```

fastest: press **F1** while the cursor is on the command (**Fn** on laptops)

package containing the function

title & description,  
sometimes additional info  
in the section **Details** or **Note**

arguments, sometimes with defaults

description of the arguments

output of the function (returned value)

sources, often also under **See Also**

examples, often very helpful!

further info about the package



## Further help

```
help.search("append") # all available matching docs
```

```
??append # again a shorter variant
```

```
help.start() # offline manuals and materials
```

- ▶ course forum
- ▶ StackOverflow for programming questions <- main resource
- ▶ (online) book: Grolemund & Wickham (2017) - R for Data Science
- ▶ German book: Uwe Ligges (2005) - Programmieren mit R
- ▶ Reference Card: Tom Short & Jonas Stein (2013)
- ▶ base and advanced cheatsheets from Rstudio
- ▶ more at [bookdown.org/brry/course/resources.html](http://bookdown.org/brry/course/resources.html)
- ▶ Online R instances: [rdrr.io/snippets](https://rdrr.io/snippets), [cocalc.com](https://cocalc.com), [colab.to/r](https://colab.to/r)



### find answers to R questions:

- ▶ open the documentation (manual) of a function ( `?`  / `F1` )
- ▶ Stackoverflow
- ▶ course forum
- ▶ RefCard & books

Report unclear tasks in the forum

Highlight the topics from this lesson in your RefCard



- 1. Intro
  - 2. Basics
  - 3. Objects
  - 4. Data
  - 5. Graphics
  - 6. Flow control
- 2.1 Syntax
  - 2.2 Help
  - 2.3 Vectors**
  - 2.4 Statistics
  - 2.5 Functions
  - 2.6 Logic
  - 2.7 Character strings
  - 2.8 Categories
  - 2.9 Packages



## Vectors I

Vectors in R are not geometric constructs, but an ordered set of values. Vectors are created with `c` (Combine / Concatenate). Entries are separated with a comma.

```
numbers <- c(3, 7, -2.7654321, 11, 3.8, 9)
```

call (display, print) object:

```
numbers
```

```
## [1] 3.000000 7.000000 -2.765432 11.000000 3.800000  
## [6] 9.000000
```

```
print(numbers, digits=3) # explicit printing with options  
## [1] 3.00 7.00 -2.77 11.00 3.80 9.00
```



## Vectors II: sequences

```
1:5 # whole numbers (integers) from : to
## [1] 1 2 3 4 5

rep(1:4, times=3) # repeat numbers several times
## [1] 1 2 3 4 1 2 3 4 1 2 3 4

rep(1:3, each=3, times=2)
## [1] 1 1 1 2 2 2 3 3 3 1 1 1 2 2 2 3 3 3

seq(from=3, to=-1, by=-0.5) # sequence
# for descending sequences, 'by' must be negative
## [1] 3.0 2.5 2.0 1.5 1.0 0.5 0.0 -0.5 -1.0

seq(1.32, 6.1, length.out=9) # 9 elements
## [1] 1.3200 1.9175 2.5150 3.1125 3.7100 4.3075 4.9050
## [8] 5.5025 6.1000

seq(1.32, 6.1, len=15) # argument names abbreviable
```



Indexing: select subsets -> square brackets AltGr + 8 / 9, Option + 5 / 6

```
vec <- c(3, 7, -2, 11, 4, 9)
```

```
vec[1] # return and print first element  
## [1] 3
```

```
vec[2:4] # select several elements  
## [1] 7 -2 11
```

```
vec[ c(2,5,1,6,1) ] # flexible order  
## [1] 7 4 3 9 3
```

```
vec[-2] # all elements except the second  
## [1] 3 -2 11 4 9
```

```
vec[-(1:3)] # all elements except the first three  
## [1] 11 4 9
```

```
vec[-1:3] # does not work  
## Error in vec[-1:3]: only 0's may be mixed with negative subscripts  
-1:3      # because -1 and 1 cannot both be fulfilled  
## [1] -1 0 1 2 3
```



## head/tail, str, class, length

```
a <- seq(from=1, to=100, by=0.1)
head(a) # select and display first 6 elements
## [1] 1.0 1.1 1.2 1.3 1.4 1.5

tail(a, 8) # the last 8 elements
## [1] 99.3 99.4 99.5 99.6 99.7 99.8 99.9 100.0

a[2] <- 87 # change a single element of an object
head(a) # the object 'a' is now different
## [1] 1.0 87.0 1.2 1.3 1.4 1.5

str(a) # structure: data type, [Dimension], first values
## num [1:991] 1 87 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 ...
class(a) # common: numeric, logical, factor, character
## [1] "numeric"

length(a) # vector length (number of elements)
## [1] 991
```



## Recycling: extend vectors as needed

```
2 * 7
## [1] 14

2:9 * 7          # 7 is repeated as often as necessary
## [1] 14 21 28 35 42 49 56 63

2:9 * c(7,1)    # this concept is called "recycling"
## [1] 14 3 28 5 42 7 56 9
##       7 1 7 1 7 1 7 1

2:9 * c(7,1,2) # Result with warning if it does not fit
## Warning in 2:9 * c(7, 1, 2): longer object length is
## not a multiple of shorter object length
## [1] 14 3 8 35 6 14 56 9
```



## Summary for 2.3 Vectors

### create and index vectors:

- ▶ `c` , `:` , `rep` , `seq`
- ▶ `v[n]` , `v[-n]` , `v[m:n]` , `v[-(m:n)]`
- ▶ `head` , `tail` , `str` , `class` , `length`
- ▶ Recycling
- ▶ Warning messages can sometimes be ignored

Report unclear tasks in the forum

Highlight the topics from this lesson in your RefCard



## Vectors with names

```
score <- c(Christoph=19, Berry=17, "Anna Lena"=22)
```

Better avoid spaces in names

```
score[2] # index: by position
```

```
## Berry
```

```
## 17
```

```
score["Berry"] # index: by name
```

```
## Berry
```

```
## 17
```

```
names(score) # 'score' is a "named vector"
```

```
## [1] "Christoph" "Berry"      "Anna Lena"
```

```
names(score) <- LETTERS[1:3]
```

```
names(score)[2] <- "NewName"
```

```
score
```

```
##      A NewName      C
```

```
##      19      17      22
```



- 1. Intro
  - 2. Basics
  - 3. Objects
  - 4. Data
  - 5. Graphics
  - 6. Flow control
- 
- 2.1 Syntax
  - 2.2 Help
  - 2.3 Vectors
  - 2.4 Statistics**
  - 2.5 Functions
  - 2.6 Logic
  - 2.7 Character strings
  - 2.8 Categories
  - 2.9 Packages



## statistical measures I: average, dispersion, range of values

```
# vector with body sizes:
```

```
size <- c(149.3, 173.6, 172.2, 172.9, 161.6, 179.2,  
        164.8, 162.8, 180.5, 165.1, 181.7, 171.4,  
        172.1, 148.1, 161.1, 171.9, 186.9) # cm
```

```
mean(size)    # arithmetic mean
```

```
## [1] 169.1294
```

```
var(size)      # variance: cm^2
```

```
## [1] 112.591
```

```
sd(size)       # standard deviation: cm
```

```
## [1] 10.61089
```

```
min(size)      # minimum
```

```
## [1] 148.1
```

```
max(size)      # maximum
```

```
## [1] 186.9
```

```
range(size)    # range of values (domain)
```

```
## [1] 148.1 186.9
```



## statistical measures II: even without normal distribution

```
median(size) # outlier-independent (unlike mean)
```

```
## [1] 171.9
```

```
mad(size) # median absolute deviation
```

```
## [1] 10.82298
```

```
quantile(size) # proportion < specific values
```

```
##    0%    25%    50%    75%   100%
```

```
## 148.1 162.8 171.9 173.6 186.9
```

```
quantile(size, probs=0.80) # 80% is below this
```

```
##    80%
```

```
## 178.08
```

```
summary(size)
```

```
##    Min.  1st Qu.  Median      Mean  3rd Qu.  Max.
```

```
##    148.1    162.8    171.9    169.1    173.6    186.9
```

Also shows number of NAs (if any), can also be used for data frames, see corresponding section 3.1



## Sorting

```
size <- round(size[1:8])      ;      size
## [1] 149 174 172 173 162 179 165 163
```

```
sort(size) # sort in ascending order
## [1] 149 162 163 165 172 173 174 179
```

```
sort(size, decreasing=TRUE)
## [1] 179 174 173 172 165 163 162 149
```

```
order(size)
## [1] 1 5 8 7 3 4 2 6
```

The smallest is in position 1, the second smallest in `size[5]`, etc.

```
weight <- c(49, 77, 66, 91, 69, 72, 73, 74)
```

```
weight[order(size)] # sort weight by order of size
## [1] 49 69 74 73 66 91 77 72
```



## Random numbers

```
sample(0:9, size=7)          # randomly pull values from vector
## [1] 2 7 1 4 9 8 6

sample(0:9, size=7, replace=TRUE)      # draw with replacement
## [1] 5 6 9 0 7 6 5
```

## Continuous distributions:

```
rnorm(n=5, mean=20, sd=3.5)      # from normal distribution
## [1] 21.9 19.0 20.4 19.9 11.2

rexp(n=5, rate=1/20)            # exponential distribution
## [1] 7.27 23.47 22.79 8.56 29.17

runif(n=5, min=15, max=25)      # uniform distribution
## [1] 22.8 19.3 24.3 22.7 17.6

rbeta(n=5, shape1=3, shape2=9)    # beta distribution
## [1] 0.1879 0.0687 0.2998 0.4172 0.1498
```

## Discrete distributions:

```
rpois(n=5, lambda=20)          # Poisson distribution
## [1] 19 13 23 29 29

rbinom(n=5, size=100, prob=1/5)  # binomial distribution
## [1] 27 16 21 27 22
```



### statistical measures, sorting and random numbers:

- ▶ `mean` , `var` , `sd`
- ▶ `min` , `max` , `range` , `median` , `quantile` , `summary`
- ▶ `round` , `sort` , `order` (decreasing)
- ▶ `sample` , `rnorm` etc

Report unclear tasks in the forum

Highlight the topics from this lesson in your RefCard



## unique and duplicated

```
val <- c(1, 7, 3, 3, 8, 5, 6, 6, 6, 7)
unique(val) # keeps original order
## [1] 1 7 3 8 5 6

duplicated(val)
## [1] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE
## [9]  TRUE  TRUE
duplicated(val, fromLast=TRUE)
## [1] FALSE  TRUE  TRUE FALSE FALSE FALSE  TRUE  TRUE
## [9] FALSE FALSE
```

## Rounding

```
values <- c(149.3, 173.6, 172.2, 172.9, 161.6, 179.2, 164.8, 142.8)

round(values, digits=-1) # round to 10s
## [1] 150 170 170 170 160 180 160 140
round(values, -2) # round to 100s
## [1] 100 200 200 200 200 200 200 100
round(values/5)*5 # round to 5s
## [1] 150 175 170 175 160 180 165 145
```



## Display options

```
pi  
## [1] 3.141593
```

The behavior of R can be customized in many options, e.g. for handling warning messages or printed output. `digits` controls how many relevant decimal places are displayed (relative to the magnitude of the number):

```
oo <- options(digits=3) # display ca 2 decimal places  
oo # previous value now in oo (old options)  
## $digits  
## [1] 7  
  
pi  
## [1] 3.14  
  
options(oo) ; rm(oo) # Reset custom settings
```



```
sample(1:50, 3)
## [1] 18 45 5
sample(1:50, 3)
## [1] 37 29 22
```

Generate the same "random numbers" over and over again for the slides.

-> Set starting point for the RNG (Random Number Generator):

```
set.seed(12345)
```

```
sample(1:50, 3)
## [1] 14 16 26
```

```
set.seed(12345)
sample(1:50, 3)
## [1] 14 16 26
```

- 1. Intro
  - 2. Basics
  - 3. Objects
  - 4. Data
  - 5. Graphics
  - 6. Flow control
- 
- 2.1 Syntax
  - 2.2 Help
  - 2.3 Vectors
  - 2.4 Statistics
  - 2.5 Functions**
  - 2.6 Logic
  - 2.7 Character strings
  - 2.8 Categories
  - 2.9 Packages



## Functions: make code systematically reusable

```
divide <- function(number, divisor=5)
{
  output <- number / divisor
  output <- round(output, digits=4)
  return(output)
}
```

Execute custom function (like other commands):

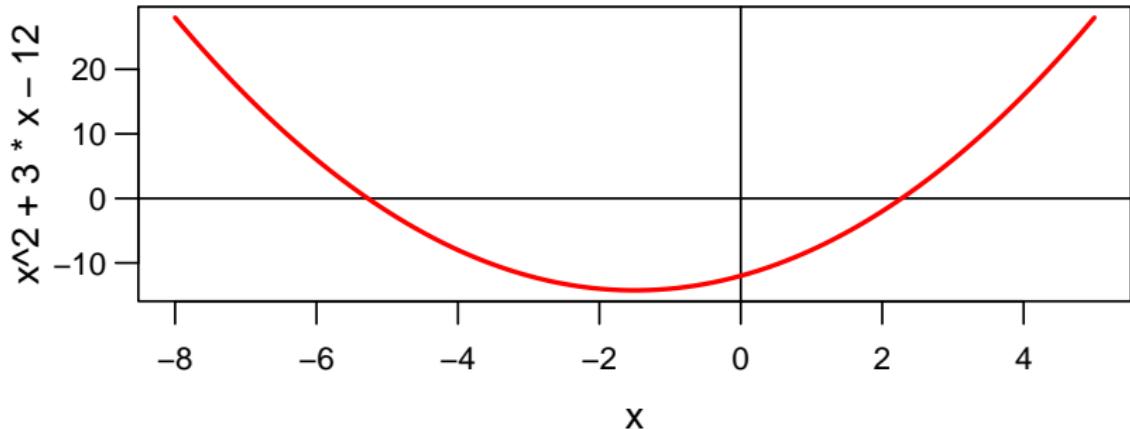
```
divide(number=23, divisor=7) # to call your function
## [1] 3.2857

divide(23) # uses default value (5) for 'divisor'
## [1] 4.6
```

With functions, the exercise check routine can easily execute and examine your code for different inputs.



## Real example: PQ formula for zeros of a quadratic function



```
function(p,q) # y = x^2 + px + q
{
  w <- sqrt( p^2 / 4 - q )
  c(-p/2-w, -p/2+w)
}
```

```
## [1] -5.274917 2.274917
```



## Object environment

```
divide <- function(number, divisor=5) # example from before
{
  output <- number / divisor
  output <- round(output, digits=4)
  return(output)
}
```

The code inside the function body is executed by R in a separate environment.

Objects within a function (`number`, `divisor` & `output` in the example above) are local / temporary and not created in the normal workspace (global environment).

They are not available in the `globalenv()` afterwards.



## Return / Curly brackets

`return()` terminates the execution of the function. Code after it will not be called.

`return()` can be omitted, then result of the last calculation (statement, "expression") is returned:

```
divide <- function(number, divisor=5){  
  output <- number / divisor  
  round(output, digits=4)  
}
```

For functions that contain only a single statement (expression), the curly braces are optional:

```
normalize <- function(x) (x-min(x)) / (max(x)-min(x))
```

```
# scale -7 to 13 unto 0 to 1
```

```
normalize( c(8,-7,13,2,3) )
```

```
## [1] 0.75 0.00 1.00 0.45 0.50
```



### functions to reuse code:

- ▶ `myFun <- function(x) {x+7}`
- ▶ workspace: normally, objects are in the "global environment"
- ▶ objects inside a function are in a temporary environment
- ▶ `return` at the end of functions is optional
- ▶ `{}` are optional, if the function contains only one command

Report unclear tasks in the forum

Highlight the topics from this lesson in your RefCard



## Application example 1: Statistics

Function with number-vector as input, adds 1000 to the values and computes mean and standard deviation (location and spread of the values)

```
computeStats <- function(numbers)
{
  numbers <- numbers + 1000
  intermediate_mu <- mean(numbers)
  intermediate_sd <- sd(numbers)
  output <- c(intermediate_mu, intermediate_sd)
  return(output)
}
```

Vector with numbers in the usual Global Environment:

```
someValues <- c(284,890,522,498,744,39,534)
```

```
computeStats(someValues)
## [1] 1501.5714 280.6064
```



## Application example 1 (continued)

```
otherValues <- c(1490,2876,4242,9007)
computeStats(otherValues)
## [1] 5403.750 3268.029

computeStats(1:50)
## [1] 1025.50000 14.57738
computeStats(-1500 + 0:7)
## [1] -496.50000 2.44949
```

For this simple example, the function can be a lot shorter:

```
computeMS <- function(z) c(mean(z+1000), sd(z+1000))
computeMS(-1500 + 0:7) # same result
## [1] -496.50000 2.44949
```

## Application example 2: Workspace

```
an_object <- 7
a_function <- function(input) input * 33 + an_object
a_function(2) # 66 + 7 = 73
## [1] 73
```

The function uses `input` as passed in when it is called.

It searches `an_object`, does not find it inside the function, continues searching and finds it in the Global Environment and uses its value 7.

```
rm(an_object)
a_function(2)
## Error in a_function(2): object 'an_object' not found
```

it is better to use:

```
a_function <- function(input, summand)
           input * 33 + summand
a_function(2, 7)
## [1] 73
```



- 1. Intro
- 2. Basics
- 3. Objects
- 4. Data
- 5. Graphics
- 6. Flow control

- 2.1 Syntax
- 2.2 Help
- 2.3 Vectors
- 2.4 Statistics
- 2.5 Functions
- 2.6 Logic
- 2.7 Character strings
- 2.8 Categories
- 2.9 Packages



Boolean = logical = truth values

```
7 > 4  
## [1] TRUE      SHIFT + <
```

```
7 > 42  
## [1] FALSE
```

T # = TRUE. T can be overwritten. Don't use it.

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

```
! 7 > 4          # NOT-operator (negation, opposite)  
## [1] FALSE
```

```
TRUE & TRUE      SHIFT + 6          # AND-operator  
## [1] TRUE  
TRUE & FALSE  
## [1] FALSE
```

```
TRUE | FALSE      AltGr + <, Option + 7  # OR-operator  
## [1] TRUE
```



## logic operators I: vectors

```
x <- c(1, 2, 3, 4, 5)
y <- c(4, 5, 6, 7, 1)
```

Many operators are vectorized, so they work for a whole vector:

```
x > 3
## [1] FALSE FALSE FALSE TRUE TRUE
y < 6
## [1] TRUE TRUE FALSE FALSE TRUE
x>3 & y<6          # for vector with several truth values
## [1] FALSE FALSE FALSE FALSE TRUE
x>3 | y<6
## [1] TRUE TRUE FALSE TRUE TRUE
```

`&&` and `||` evaluate only the first value:

```
x>3 && y<6          # for a single truth value
## Warning in x > 3 && y < 6: 'length(x) = 5 > 1' in
coercion to 'logical(1)'
## [1] FALSE
```

## logic operators II: size comparison

```
values <- c(28, 29, 30, 32, 31, 32)
```

```
values < 30
```

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

```
values <= 30
```

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

```
values > 30
```

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

```
values >= 30
```

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

```
values == 30
```

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

```
values != 30
```

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



## logic operators III: which , any , all

```
values <- c(28, 29, 30, 32, 31, 32)
```

```
values < 30
```

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

```
which(values < 30) # -> Index: places with TRUE in vector
```

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

```
which(values == max(values))
```

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

```
which.max(values) # only the index of first occurrence!
```

```
## [1] 4
```

```
any(values < 30) # is at least one of the T/F values TRUE?
```

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

```
all(values < 30) # are all TRUE?
```

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



```
values < 30
## [1] TRUE TRUE FALSE FALSE FALSE FALSE
as.numeric(values < 30) # internally: 0=FALSE, 1=TRUE
## [1] 1 1 0 0 0 0
sum(values < 30) # number of TRUES in the vector
## [1] 2
mean(values < 30) # proportion of TRUE values
## [1] 0.3333333
```



## Logical selection (subsetting, indexing) = Filtering

```
values <- c( 28, 29, 30, 32, 31, 32)
names <- c("a", "b", "c", "d", "e", "f")
```

```
names[4]
```

```
## [1] "d"
```

```
values < 32
```

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

values	names	bigger
28	a	TRUE
29	b	TRUE
30	c	TRUE
32	d	FALSE
31	e	TRUE
32	f	FALSE

```
names[values < 32]
```

```
## [1] "a" "b" "c" "e"
```

To select with logical values ("filtering"), both vectors should have the same length.



### logical values, size comparison:

- ▶ TRUE , FALSE (don't use T , F )
- ▶ ! , & , | , && , ||
- ▶ < , > , <= , >= , == , !=
- ▶ which , which.max , any , all , sum , mean
- ▶ vec[logical]

Report unclear tasks in the forum

Highlight the topics from this lesson in your RefCard



## Caution when checking equality

Check equality only for integer or rounded numbers!

```
0.4 - 0.1 == 0.3           # not the expected result!  
## [1] FALSE
```

```
print(0.4-0.1 , digits=22)  
## [1] 0.3000000000000000444089
```

```
round(0.4 - 0.1, digits=5) == round(0.3, digits=5)    # OK  
## [1] TRUE
```

```
all.equal(0.4 - 0.1, 0.3) # has an error tolerance  
## [1] TRUE
```

Variant for vectors:

```
berryFunctions::almost.equal(c(6.34, 9.69, 3.77), 9.69)  
## [1] FALSE TRUE FALSE
```

More such pitfalls in the [R inferno](#).

See also (the image in) the [floating point format standard](#) wikipedia description.

## More details about logic

```
T          # can be abbreviated, however:  
## [1] TRUE  
T <- 99    # T can be overwritten  
T <- FALSE # Prank: secretly type at colleague's PC ;)  
TRUE <- 77 # is protected  
## Error in TRUE <- 77: invalid (do_set) left-hand side  
to assignment
```

```
xor(TRUE, FALSE) # EXCLUSIVE OR (exactly 1 of 2 true?)  
## [1] TRUE
```

```
isTRUE(T); isTRUE(F); isTRUE(NA) # T, F, F (for NAs)
```

## More details about &amp;

A & B and A && B differ in another thing.

If A is false, in the second variant B is not evaluated at all, because the output cannot be a vector where a TRUE could still occur.

```
compute <- function(out){cat("computing...\\n") ; out}
```

# Output of both 'compute' calls:

```
compute(FALSE) & compute(TRUE)  
## computing...  
## computing...  
## [1] FALSE
```

# Only the left instance is executed:

```
compute(FALSE) && compute(TRUE)  
## computing...  
## [1] FALSE
```

## Truth tables

```
berryFunctions::TFtest(a|b, na=FALSE)
##      a      b  __ a | b
## 1  TRUE  TRUE      TRUE
## 2  TRUE FALSE      TRUE
## 3 FALSE TRUE      TRUE
## 4 FALSE FALSE     FALSE
```

& has operator precedence over | (like \* over +):

```
berryFunctions::TFtest(a|b&c, a|(b&c), (a|b)&c, na=FALSE)
##      a      b      c  __ a | b & c  __ a | (b & c)  __ (a | b) & c
## 1  TRUE  TRUE  TRUE      TRUE      TRUE      TRUE
## 2  TRUE  TRUE FALSE     TRUE      TRUE      FALSE
## 3  TRUE FALSE  TRUE     TRUE      TRUE      TRUE
## 4  TRUE FALSE FALSE     TRUE      TRUE      FALSE
## 5 FALSE  TRUE  TRUE     TRUE      TRUE      TRUE
## 6 FALSE  TRUE FALSE    FALSE      FALSE      FALSE
## 7 FALSE FALSE  TRUE    FALSE      FALSE      FALSE
## 8 FALSE FALSE FALSE   FALSE      FALSE      FALSE
```

see also ?Syntax

- 1. Intro
  - 2. Basics
  - 3. Objects
  - 4. Data
  - 5. Graphics
  - 6. Flow control
- 2.1 Syntax
  - 2.2 Help
  - 2.3 Vectors
  - 2.4 Statistics
  - 2.5 Functions
  - 2.6 Logic
  - 2.7 Character strings**
  - 2.8 Categories
  - 2.9 Packages



## Character string basics

```
"Hey hey"          # quotation marks:  
'Good morning'   # both kinds possible  
  
sentence <- "Note: nono knots are not tied"  
class(sentence)  
## [1] "character"  
  
nchar(sentence) # number of characters != length  
## [1] 29  
  
tolower(sentence)  
## [1] "note: nono knots are not tied"  
toupper(sentence)  
## [1] "NOTE: NONO KNOTS ARE NOT TIED"  
  
substr(sentence, start=7, stop=16) # spaces count  
## [1] "nono knots"  
  
cat(sentence) # concatenate and type  
## Note: nono knots are not tied
```

## Character string misc

```
trimws(" Text with      spaces. ")
## [1] "Text with      spaces."  
  
as.character(pi) # convert numbers to strings
## [1] "3.14159265358979"  
  
x <- c("Some", "blue", "flowers", "bloom", "in", "BLUE")
startsWith(x, "bl") # per entry: does it start with bl?
## [1] FALSE  TRUE FALSE  TRUE FALSE FALSE
endsWith(x, "e") # for each element of x. case sensitive
## [1]  TRUE  TRUE FALSE FALSE FALSE FALSE
```



## Merge character strings

```
paste("word", 1:4) # vectorized: 'word' recycled 4 times
## [1] "word 1" "word 2" "word 3" "word 4"
```

```
paste("word", 1:4, sep="_/") # Custom separator string
## [1] "word_1" "word_2" "word_3" "word_4"
```

```
paste0("word", 1:4) # empty charstring '' as separator
## [1] "word1" "word2" "word3" "word4"
```

Merge multiple elements into a single string:

```
paste0("word", 1:4, collapse="-")
## [1] "word1-word2-word3-word4"
```

```
toString(c("these", 1:5, "words")) # comma-separated
## [1] "these, 1, 2, 3, 4, 5, words"
```



## Separate (split) character strings

```
sentence
## [1] "Note: nono knots are not tied"

words <- strsplit(sentence, split=" ")[[1]]
```

`strsplit` returns a list. To select the first element (which has a vector with 5 entries), we use double square brackets. More about lists in the corresponding section [3.3](#).

```
words
## [1] "Note:" "nono"   "knots"  "are"    "not"    "tied"
```



## Search character strings

words

```
## [1] "Note:" "nono"  "knots" "are"    "not"    "tied"
```

```
match("not", words)      # index of the first matching entry
```

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

```
match("no", words)       # only complete matches
```

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

```
"not" %in% words      # logical value, whether entry occurs
```

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

```
grep("no", words)       # in which elements does 'no' occur?
```

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

```
grep("no", words, value=TRUE)      # words containing 'no'
```

```
## [1] "nono"  "knots" "not"
```

```
grepl("no", words)      # for each word: is 'no' included?
```

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



## Replace character strings

```
words
## [1] "Note:" "nono"   "knots"  "are"    "not"    "tied"

# replace the first find in each case:
sub(pattern="no", replacement="NO", x=words)
## [1] "Note:" "NOno"   "kN0ts"  "are"    "N0t"    "tied"

# replace all occurrences of 'no'
gsub(pattern="no", replacement="NO", x=words)
## [1] "Note:" "NONO"   "kN0ts"  "are"    "N0t"    "tied"
```



Regular expression (regex): capitalization, begin / end

grep: global search for a regular expression, print out matched lines

```
x <- c("abz", "Abz", "yzab", "abyz", "nichts")
```

```
grep("ab", x, v=T) # value=TRUE abbreviated for slides only
## [1] "abz"   "yzab"  "abyz"
```

```
grep("ab", x, v=T, ignore.case=TRUE) # ign. capitalization
## [1] "abz"   "Abz"   "yzab"  "abyz"
```

```
grep("^ab", x, v=T) # caret: must begin with
## [1] "abz"   "abyz"
```

```
grep("yz", x, v=T)
## [1] "yzab"  "abyz"
```

```
grep("yz$", x, v=T) # dollar: must end with
## [1] "abyz"
```

See also: `startsWith` and `endsWith`



## regex: wildcards

```
x <- c("cfu", "cfgu", "cfghu", "cnu", "cmu")
```

```
grep("c.u", x, v=T) # . : any arbitrary character  
## [1] "cfu" "cnu" "cmu"
```

```
grep("c.*u", x, v=T) # .*: no matter how many characters  
## [1] "cfu"    "cfgu"   "cfghu"  "cnu"    "cmu"
```

```
grep("c.{2}u", x, v=T) # .{2}: exactly 2 arbitrary chars  
## [1] "cfgu"
```

```
grep("c(f|n)u", x, v=T) # (x/y): x or y  
## [1] "cfu" "cnu"
```

```
grep("c[kmf]u", x, v=T) # [xyz]: any of these chars  
## [1] "cfu" "cmu"
```

```
grep("c[^km]u", x, v=T) # [^xyz]: not these chars (normally ^startswith)  
## [1] "cfu" "cnu"
```

```
grep("c[k-o]u", x, v=T) # [a-zA-Z]: between a and X  
## [1] "cnu" "cmu"
```



## Special character backslash

A backslash signals that something special comes after it.

```
cat("sentence with\nline break") # \newline  
## sentence with  
## line break  
AltGr + ⚁,  
cat("1\t9","1234\t9","12345678\t9", sep="\n") # \tabstop  
## 1      9  
## 1234   9  
## 12345678       9
```

```
cat("sentence with \" symbol") # quotation mark  
## sentence with " symbol  
cat('sentence with " symbol') # less typing :)  
## sentence with " symbol
```

```
cat("sentence with \\ literal") # backslash itself  
## sentence with \ literal
```

```
cat("sentence with \u{0B00} degree symbol") # \Unicode  
## sentence with ° degree symbol
```

## Summary for 2.7 Character strings

### character strings:

- ▶ `"char"`, `'string'`, `as.character`
- ▶ `nchar`, `tolower`, `toupper`, `trimws`
- ▶ `paste`(`sep`,`collapse`), `paste0`, `toString`
- ▶ `substr`, `strsplit`
- ▶ `startsWith`, `endsWith`
- ▶ `match`, `%in%`, `grep`(`ignore.case`), `grepl`
- ▶ `sub`, `gsub`
- ▶ Regex: `"^begin"`, `"end$"`, `".+"`, `".*"`
- ▶ `\n`, `\t`, `\\"`, `\U{}`, `\\`

Report unclear tasks in the forum

Highlight the topics from this lesson in your RefCard



Why `strsplit` returns a list

```
v <- c("ab-cdefg-hij-k-lmn", "opqrstuvwxyz")
v
## [1] "ab-cdefg-hij-k-lmn" "opqrstuvwxyz"

w <- strsplit(v, split="-")
w # one list element for each original vector element
## [[1]]
## [1] "ab"      "cdefg"   "hij"     "k"       "lmn"
##
## [[2]]
## [1] "opqrstuvwxyz" "wxyz"

w[[1]]
## [1] "ab"      "cdefg"   "hij"     "k"       "lmn"
```



## Search character strings II

```
words
## [1] "Note:" "nono"  "knots" "are"    "not"    "tied"
# for each word: at which position 'no' begins
c(regex("no", words)) # -1 when not included
## [1] -1  1  2 -1  1 -1
gregexpr("no", words) # ditto: all locations -> list
## -1      1, 3        2          -1         1          -1
```

## regex: repetitions

repetition quantifiers for frequency of the preceding item:

```
x <- c("cd", "cxd", "cxxd", "cxxxd", "cxxxxd")
```

```
grep("cxd", x, v=T)
```

```
## [1] "cxd"
```

```
grep("cx?d" , x, v=T) # ?: 0 or 1 times
```

```
## [1] "cd"   "cxd"
```

```
grep("cx*d" , x, v=T) # *: 0 or more times
```

```
## [1] "cd"   "cxd"   "cxxd"   "cxxxd"   "cxxxxd"
```

```
grep("cx+d" , x, v=T) # +: once or more often
```

```
## [1] "cxd"   "cxxd"   "cxxxd"   "cxxxxd"
```

```
grep("cx{2}d" , x, v=T) # {n}: n times
```

```
## [1] "cxxd"
```

```
grep("cx{2,}d" , x, v=T) # {n,}: n times or more
```

```
## [1] "cxxd"   "cxxxd"   "cxxxxd"
```

```
grep("cx{2,3}d" , x, v=T) # {n,m}: n to m times
```

```
## [1] "cxxd"   "cxxxd"
```

## Ignore regex operators

```
". \ | ( ) [ { ^ $ * + ?" # regex metacharacters
```

```
x <- c("ab.de", "abde", "a^bcde", "bcde")
```

```
grep("^bc", x, value=TRUE)  
## [1] "bcde"
```

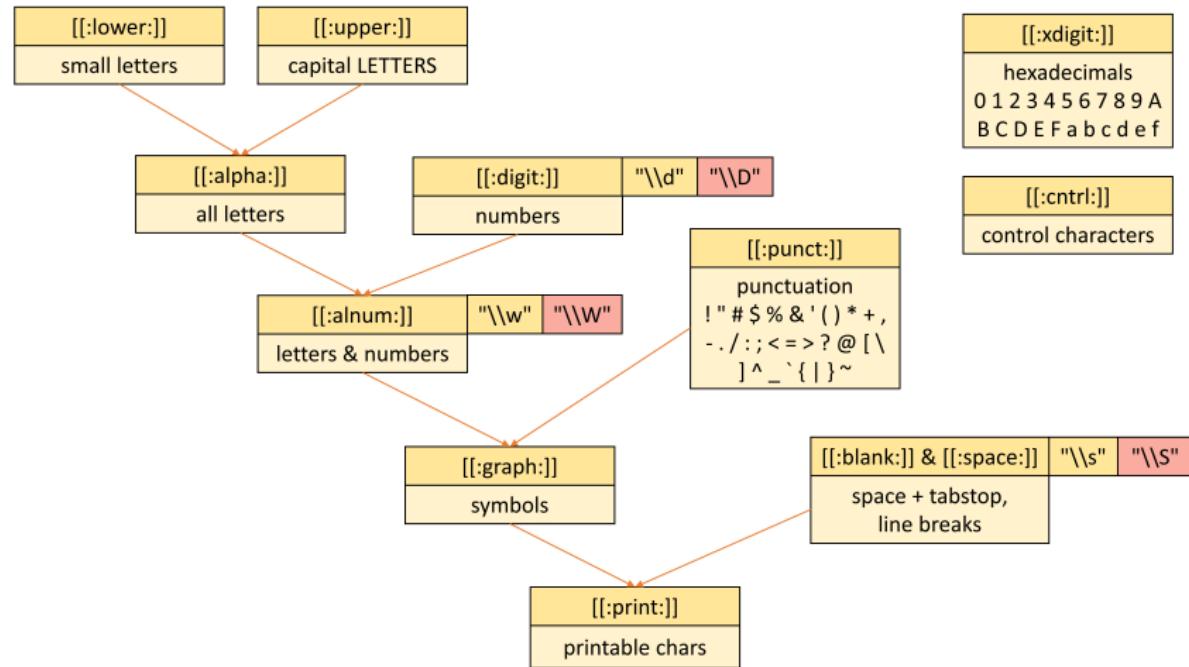
```
grep("^bc", x, value=TRUE, fixed=TRUE) # without regex  
## [1] "a^bcde"
```

```
grep(".de", x, value=TRUE)  
## [1] "ab.de" "abde" "a^bcde" "bcde"
```

```
grep("\\.de", x, value=TRUE) # real period (with regex)  
## [1] "ab.de"
```

## Prepared collections of strings

```
grep("[UV[:digit:]WX]", c("ab3d", "abUd", "abcd"), v=T)  
## [1] "ab3d" "abUd"
```



See also IBM on [backslash sequences](#)

- 1. Intro
  - 2. Basics
  - 3. Objects
  - 4. Data
  - 5. Graphics
  - 6. Flow control
- 2.1 Syntax
  - 2.2 Help
  - 2.3 Vectors
  - 2.4 Statistics
  - 2.5 Functions
  - 2.6 Logic
  - 2.7 Character strings
  - 2.8 Categories
  - 2.9 Packages



## Factors = categorial variables

```
factor(c("boat", "car", "train", "car", "boat"))
## [1] boat car train car boat
## Levels: boat car train
```

Sorted alphabetically by default. For ordinal variables:

```
factor(c("boat", "car", "train", "car", "boat"),
       levels=c("boat", "train", "car"))
## [1] boat car train car boat
## Levels: boat train car
```

```
state.region[37] # built-in dataset with factors
## [1] West
## Levels: Northeast South North Central West
```

```
class(state.region)
```

```
## [1] "factor"
```

```
levels(state.region)
## [1] "Northeast"      "South"          "North Central"
## [4] "West"
```



## Frequency tables

```
table(state.region) # Number of occurrences per value
## state.region
##      Northeast          South       North      Central        West
##                 9             16            12            13
```

```
grades <- strsplit("CEBCABBCEBBBACDDBDCFCA", "")[[1]]
table(grades, dnn=NULL) # dnn: omit dimension names
## A B C D E F
## 3 7 6 3 2 1
```

```
names(table(grades))
## [1] "A" "B" "C" "D" "E" "F"
```

```
sex <- c("m", "m", "m", "f", "d", "f", "f", "m", "m", "d", "f",
       "d", "f", "m", "f", "f", "f", "f", "f", "d", "m", "f")
```

```
table(sex, grades) # contingency table / crosstab
```

```
##      grades
## sex A B C D E F
##   d 1 2 0 0 0 1
##   f 2 4 2 3 0 0
##   m 0 1 4 0 2 0
```



tagged (grouped) apply: call a function group-wise

```
head(state.name) # charstring
## [1] "Alabama"     "Alaska"       "Arizona"      "Arkansas"
## [5] "California"   "Colorado"

head(state.region) # category (factor)
## [1] South West  West  South West  West
## Levels: Northeast South North Central West

nchar(state.name[1:6])
## [1] 7 6 7 8 10 8
```

```
tapply(X=state.name, INDEX=state.region, FUN=nchar)
## $Northeast
## [1] 11 5 13 13 10 8 12 12 7
## $South
## [1] 7 8 8 7 7 8 9 8 11 14 8 14 9 5 8 13
## $'North Central'
## [1] 8 7 4 6 8 9 8 8 12 4 12 9
## $West
## [1] 6 7 10 8 6 5 7 6 10 6 4 10 7
```



## tagged (grouped) `apply`: aggregate

```
mean_charlen <- function(x) mean(nchar(x))

mean_charlen(state.name)
## [1] 8.44

tapply(X=state.name, INDEX=state.region, FUN=mean_charlen)
##      Northeast          South       North    Central           West
##      10.111111      9.000000     7.916667     7.076923
```

If the function always returns exactly one single value, `tapply` simplifies the result. Here, the dimension is reduced to a vector\* with 4 values.

\*: technically, an array

For small temporary things, the function does not have to be created separately and can be used nameless (anonymous function):

```
tapply(X=state.name, INDEX=state.region,
       FUN=function(x) mean(nchar(x)) )
```



## Internals and reading tip

Internally, categories are stored as numbers:

```
as.numeric(state.region) [c(37, 7, 27)]  
## [1] 4 1 3
```

If numbers are read in as factors, `as.numeric(x)` would yield the levels, only `as.numeric(as.character(x))` the actual numbers:

```
as.numeric(as.factor(19:17))  
## [1] 3 2 1
```

*# NOT 19,18,17, but their levels*

More details about factors in [Advanced R](#).



## Summary for 2.8 Categories

### categorical variables:

- ▶ factor, `levels`
- ▶ `table` for frequency + contingency tables
- ▶ `tapply` (`values`, `categories`, `function`)
- ▶ Useful for grouping data, e.g. with colors in graphics
- ▶ internally stored as integers, be careful with `as.numeric`

Report unclear tasks in the forum

Highlight the topics from this lesson in your RefCard



- 1. Intro
  - 2. Basics
  - 3. Objects
  - 4. Data
  - 5. Graphics
  - 6. Flow control
- 2.1 Syntax
  - 2.2 Help
  - 2.3 Vectors
  - 2.4 Statistics
  - 2.5 Functions
  - 2.6 Logic
  - 2.7 Character strings
  - 2.8 Categories
  - 2.9 Packages



- ▶ Many R users write code for specific tasks.
- ▶ If this could be useful for others, it is often packaged in an R package. This is a regulated form for source code including documentation, instructions and examples.
- ▶ If extensive requirements are met, it can be published on CRAN (Comprehensive R Archive Network).
- ▶ There are >18'000 [packages](#) available, see [CRAN Task Views](#)

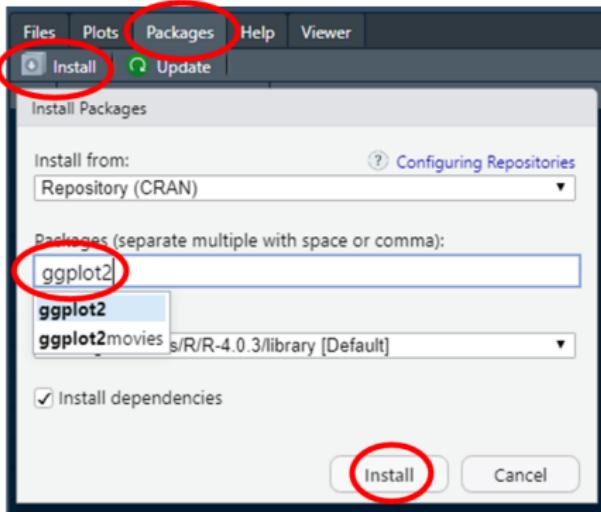
## Installing R packages

Download and install:

```
install.packages("ggplot2")
```

Needs to be executed only once, does not need admin rights.

Can also be done manually in Rstudio:



Rarely needed: undo with `remove.packages("packagename")`

## Using R packages

Load from local library:

```
library("ggplot2")
```

Needed in every new R session.

Should be in the script for it to be reproducible.

After that the functions from the package are directly usable.

For functions with the same name in multiple packages, the one that was loaded last is used.

To make it clear from which package a function is used, the

```
package::function()
```

```
rdwd::findID("Potsdam")
```

is clear + unambiguous, and hence safer than

```
library(rdw)
```

```
findID("Potsdam")
```



## Package tips

update regularly - with the Rstudio button or  
`update.packages()`

For packages that are not yet installed, the error message is misleading:

```
library("xx")
## Error in library("xx"): there is no package called
'xx'
```

In that case, simply install it with:

```
install.packages("xx")
```

```
## package 'xx' is not available for this version of R
```

This warning usually indicates a typo.

With the Rstudio Packages - Install menu, these can mostly be avoided.



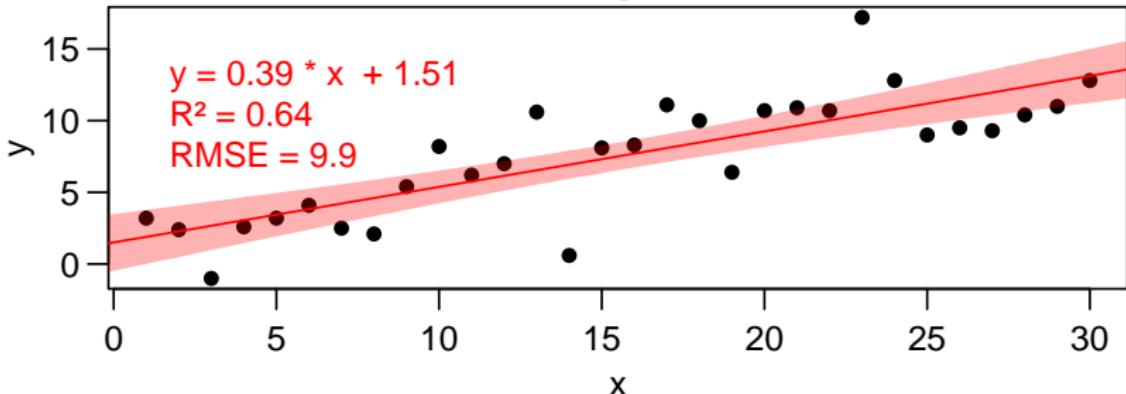
Install packages automatically when needed

```
# If a package is not available, then install it:  
if(!requireNamespace("berryFunctions", quietly=TRUE))  
  install.packages("berryFunctions")
```

More at [bookdown.org/brry/course/packages](http://bookdown.org/brry/course/packages)

```
x <- 1:30  
y <- c(3.2, 2.4, -1, 2.6, 3.2, 4.1, 2.5, 2.1, 5.4, 8.2,  
      6.2, 7, 10.6, 0.6, 8.1, 8.3, 11.1, 10, 6.4, 10.7,  
      10.9, 10.7, 17.2, 12.8, 9, 9.5, 9.3, 10.4, 11, 12.8)  
berryFunctions::linReg(x, y, pos1="topleft")
```

linear regression



## preinstalled packages / source code of R commands

Always provided (base R):

- ▶ loaded: `base`, `datasets`, `utils`, `grDevices`, `graphics`, `stats`, `methods`
- ▶ load manually: `compiler`, `grid`, `parallel`, `splines`, `tcltk`, `tools`

Examine definition (source code) of a function:

- ▶ call without brackets: `append` instead of `append()`
- ▶ open in github: `berryFunctions::funSource` (`F7` with `rskey`)
  - ▶ base R: [github.com/wch/r-source/src/library/base](https://github.com/wch/r-source/src/library/base) (stats, parallel, ...)
  - ▶ CRAN: [github.com/cran](https://github.com/cran)
- ▶ `head` - `UseMethod` - `methods(head)` - zB `head.matrix` ([more](#))
- ▶ `abs` - `.Primitive` - `do_abs` in `src/main/names.c` and `/arithmetic.c`, see  
`pryr::show_c_source(.Primitive("abs"))`, also for `.Internal`

Online help for CRAN package functions (without installing):

[www.rdocumentation.org](http://www.rdocumentation.org)



**packages with additional code, checked packages on CRAN:**

- ▶ `install.packages` (once, OK to "point and click")
- ▶ `library` (in each script)
- ▶ `Package::function()` clearly reveals the origin of a function
- ▶ `requireNamespace` checks if a package can be loaded
- ▶ `berryFunctions::funSource` for source code on github

Report unclear tasks in the forum

Highlight the topics from this lesson in your RefCard



- 1. Intro
  - 2. Basics
  - 3. Objects
  - 4. Data
  - 5. Graphics
  - 6. Flow control
- 
- 3.1 Data frames
  - 3.2 Matrices
  - 3.3 Lists
  - 3.4 Arrays
  - 3.5 Data & Object types

## Create data.frames

Tabular data in R is almost always created as a `data.frame`. Each column can have its own data type (numeric, character, factor, logical, etc).

```
?data.frame
```

```
exdf <- data.frame(numbers=11:14, chars=letters[1:4],  
                    booleans=(1:4)>2)  
exdf # exdf: exampleDataFrame  
##   numbers  chars  booleans  
## 1       11     a    FALSE  
## 2       12     b    FALSE  
## 3       13     c     TRUE  
## 4       14     d     TRUE
```



## Examine data.frames

Do not display large dataframes completely (clutters the console)

-> use `str` + `summary` (and `head` + `tail`):

```
str(exdf)
## 'data.frame': 4 obs. of 3 variables:
## $ numbers : int 11 12 13 14
## $ chars   : chr "a" "b" "c" "d"
## $ booleans: logi FALSE FALSE TRUE TRUE
```

`summary(exdf) # helpful info per column`

```
##      numbers          chars          booleans
## Min.   :11.00   Length:4           Mode :logical
## 1st Qu.:11.75   Class :character   FALSE:2
## Median :12.50   Mode   :character   TRUE :2
## Mean   :12.50
## 3rd Qu.:13.25
## Max.   :14.00
```



## Subset: data.frame-selection by position (index)

Use square brackets for indexing (as with vectors), but specify two dimensions (comma separated):

```
exdf[ 3 , 1 ] # value in the third row, first column  
## [1] 13
```

```
exdf[ , 2 ] # all rows, second column (-> vector)  
## [1] "a" "b" "c" "d"
```

```
exdf[2, ] # all columns = complete row (-> data.frame)  
##   numbers chars booleans  
## 2       12      b     FALSE
```

```
?"[ " # to read the documentation about subsetting
```

```
nrow(exdf)  
## [1] 4
```

```
ncol(exdf) # see also dim(exdf)  
## [1] 3
```



## data.frame selection with column names

```
exdf[, "booleans"]  
## [1] FALSE FALSE TRUE TRUE
```

With Rstudio autocompletion (`TAB`-key): `$` (`Shift` + `4`)

```
exdf$booleans  
## [1] FALSE FALSE TRUE TRUE
```

```
colnames(exdf)  
## [1] "numbers" "chars"    "booleans"
```

```
colnames(exdf)[2] <- "char"      # changes the object exdf  
  
exdf  
##   numbers char booleans  
## 1       11    a     FALSE  
## 2       12    b     FALSE  
## 3       13    c      TRUE  
## 4       14    d      TRUE
```



## Selecting multiple rows / columns

```
exdf[2:3, ] # rows 2 to 3
##   numbers char booleans
## 2      12    b    FALSE
## 3      13    c     TRUE
```

```
exdf[c(4,1), ] # vectors for indexing
##   numbers char booleans
## 4      14    d     TRUE
## 1      11    a    FALSE
```

```
exdf[exdf$char=="c", ] # logical values: 'filtering'
##   numbers char booleans
## 3      13    c     TRUE
```

```
exdf[-2, 2:1] # negative selection as with vectors
##   char numbers
## 1     a      11
## 3     c      13
## 4     d      14
```



## Modify / add / remove columns

```
# overwrite an existing column:
```

```
exdf$numbers <- 45:48
```

```
# add a new column at the end:
```

```
exdf$time3000m <- c(12.08, 10.27, 11.79, 13.50)
```

```
exdf
```

```
##   numbers char booleans time3000m
```

```
## 1      45     a    FALSE    12.08
```

```
## 2      46     b    FALSE    10.27
```

```
## 3      47     c     TRUE    11.79
```

```
## 4      48     d     TRUE    13.50
```

```
# delete column:
```

```
exdf$char <- NULL
```

```
exdf
```

```
##   numbers booleans time3000m
```

```
## 1      45    FALSE    12.08
```

```
## 2      46    FALSE    10.27
```

```
## 3      47     TRUE    11.79
```

```
## 4      48     TRUE    13.50
```

## data.frame tips and tricks

### Column selection

- ▶ `df[, 2]` always selects the second column
- ▶ `df[, "name"]` is independent of the column order and better understandable in terms of readability (unless you know exactly what the second column contains)
- ▶ `df$name` is less typing and can be entered without typos with autocompletion

### General tips

- ▶ If `nrow(df)` returns `NULL`, `df` could be a vector.
- ▶ `NROW(df)` shows `length(df)`, if `df` is a vector.
- ▶ Objects can be converted with `as.data.frame(theMatrix)`
- ▶ If column names start with a number, the prefix "X" or "V" (variable) is prepended. This can happen especially when reading data.



## Handling data.frames in Rstudio

The built-in DataFrame `BOD` (provided in the R package `datasets`) can be viewed with `str(BOD)`.

The documentation can be accessed with `F1` or `?BOD`.

`data(BOD)` loads it into the GlobalEnv workspace (initially as an unevaluated `promise`). Once e.g. `head(BOD)` is executed, the dataset is listed regularly in Rstudio.

By clicking on the object name, `View(object)` is run, opening the possibility for temporary manual sorting and selection.

Clicking on the blue arrow opens `str(object)`.

The screenshot shows the RStudio interface with the 'airquality' dataset loaded into a data frame view. The top navigation bar includes tabs for 'Vorlagen.Rnw', 'skript.R', and 'airquality'. A red arrow points from the 'airquality' tab to the 'View' button in the toolbar. Another red arrow points from the 'airquality' tab to the 'str' button in the toolbar. A large red callout box labeled 'Selection + Sorting' covers the top portion of the data frame view, which displays several rows of air quality data with columns for Ozone, Solar.R, Wind, Temp, Month, and Day. To the right of the data frame, the 'Global Environment' pane is open, showing the 'airquality' dataset as a 153 obs. of 6 variables object, along with other datasets like 'iris'. A red arrow also points from the 'airquality' entry in the environment pane to the 'str' button. The bottom of the screenshot shows the 'Values' section of the environment pane, where 'BOD' is listed as a <Promise> object.

### create and index tabular data:

- ▶ `data.frame` , `str` , `summary`
- ▶ `nrow` , `ncol` , `colnames`
- ▶ `df[r,c]` , `df[r,]` , `df[,c]` , `df[,"cname"]` ,  
`df[,c("cn1","cn2")]`
- ▶ `df$cname` , `df$newCol <- "newValues"`

Report unclear tasks in the forum

Highlight the topics from this lesson in your RefCard



## Output a column as a data.frame

```
exdf[ , "booleans"]                      # -> vector
## [1] FALSE FALSE  TRUE  TRUE

exdf[ , "booleans", drop=FALSE ]    # -> data.frame
##   booleans
## 1 FALSE
## 2 FALSE
## 3  TRUE
## 4  TRUE

exdf["booleans"]                  # no comma -> data.frame  CAUTION
```

Don't use this! `object[index]` is for vectors!

R can handle that, but human readers of your code can't.

Use `dataframe[, column, drop=FALSE]` if you want a data.frame instead of a vector

```
rownames(exdf) <- c("Alex", "Berry", "Chris", "Daniel")
exdf
##           numbers booleans time3000m
## Alex          45     FALSE    12.08
## Berry         46     FALSE    10.27
## Chris         47      TRUE    11.79
## Daniel        48      TRUE    13.50

exdf["Berry", ]
##           numbers booleans time3000m
## Berry         46     FALSE    10.27
```



- 1. Intro
  - 2. Basics
  - 3. Objects**
  - 4. Data
  - 5. Graphics
  - 6. Flow control
- 3.1 Data frames
  - 3.2 Matrices**
  - 3.3 Lists
  - 3.4 Arrays
  - 3.5 Data & Object types

## Create a matrix (table with one single data type)

```
matrix(data=1:6 , nrow=2, ncol=3)
##      [,1] [,2] [,3]
## [1,]     1     3     5
## [2,]     2     4     6

m <- matrix(1:6 , nrow=2, ncol=3, byrow=TRUE)
m
##      [,1] [,2] [,3]
## [1,]     1     2     3
## [2,]     4     5     6

dim(m) ; nrow(m) ; ncol(m) ; length(m)
## [1] 2 3
## [1] 2
## [1] 3
## [1] 6

class(m) # two classes: array is superclass of matrix
## [1] "matrix" "array"
```



## Element-wise multiplication of two matrices

```
m
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6

m * 2    # multiplication per element
##      [,1] [,2] [,3]
## [1,]    2    4    6
## [2,]    8   10   12

n <- matrix(rep(0:1,each=3), ncol=3) ; n
##      [,1] [,2] [,3]
## [1,]    0    0    1
## [2,]    0    1    1

m * n                      # per element, also for +, -, etc
##      [,1] [,2] [,3]
## [1,]    0    0    3
## [2,]    0    5    6
```



## Row and column names, manipulation, transposing

```
colnames(m) <- c("A", "B", "C")    # as with data.frames  
rownames(m) <- c("row1", "row2")  
  
m[1,1] <- 989 # change (manipulate) matrix
```

```
m  
##          A B C  
## row1  989 2 3  
## row2    4 5 6
```

transpose: rotate / mirror across the diagonal (swap rows and columns)

```
t(m) # transpose  
##   row1  row2  
## A    989    4  
## B      2    5  
## C      3    6
```



Matrices have a single data type for the entire object

If one element is changed to a string, all are converted:

```
m[1,1] <- "a"  
m  
##      A    B    C  
## row1 "a"  "2"  "3"  
## row2 "4"  "5"  "6"
```

Columns are output as vectors (as with data.frames), now also rows:

```
m["row1", ]  
##      A    B    C  
## "a"  "2"  "3"  
  
class(m["row1", ])                                # -> vector  
## [1] "character"  
is.vector(m["row1", ])      # only 1 data type -> downcast  
## [1] TRUE
```

See order of coercion in 3.5.

```
m[ c(1,2,3,5) ] # select with 1D vector -> see next slide  
## [1] "a"  "4"  "2"  "3"
```



Internally, a matrix is a vector - but with dimension info

```
v <- 1:20      ;      is.matrix(v)
## [1] FALSE

dim(v) <- c(2, 10)
v
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]     1     3     5     7     9    11    13    15    17    19
## [2,]     2     4     6     8    10    12    14    16    18    20
is.matrix(v)
## [1] TRUE

dim(v) <- NULL
v
## [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20

dim(v) <- c(4, 5)      ;      v
##      [,1] [,2] [,3] [,4] [,5]
## [1,]     1     5     9    13    17
## [2,]     2     6    10    14    18
## [3,]     3     7    11    15    19
## [4,]     4     8    12    16    20
```

is.atomic(obj) shows TRUE for vectors and matrices (and arrays).



## Apply a function to rows / columns of a matrix |

```
m <- matrix(1:12, ncol=4) ; m
## [,1] [,2] [,3] [,4]
## [1,] 1 4 7 10
## [2,] 2 5 8 11
## [3,] 3 6 9 12
```

```
rowSums(m) # +colSums. Caution: rowsum() is something else
## [1] 22 26 30
```

```
colMeans(m) # + rowMeans
## [1] 2 5 8 11
```

```
# apply the function 'median' to each column:
apply(m, MARGIN=2, median)
## [1] 2 5 8 11
```

## Apply a function to rows / columns of a matrix II

```
m
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

# *MARGIN=1: keep the first (x) dimension:*

```
apply(m, MARGIN=1, FUN=median)
## [1] 5.5 6.5 7.5
```

# *Further arguments to be passed on to 'median':*

```
apply(m, MARGIN=1, FUN=median, na.rm=TRUE)
## [1] 5.5 6.5 7.5
```

```
apply(m, 1, function(x) sum(x==2)) # anonymous function
## [1] 0 1 0
```

```
apply(m, 1, FUN=function(x) cat(toString(x), " - "))
## 1, 4, 7, 10 - 2, 5, 8, 11 - 3, 6, 9, 12 -
## NULL
```



### tabular data with one single data type:

- ▶ `matrix` (`nrow`, `ncol`, `byrow`), `ncol`, `nrow`, `dim`, `length`
- ▶ Arithmetic operations are performed by element
- ▶ `colnames`, `rownames`, `t`
- ▶ `rowMeans`, `colSums`, `apply` (`X=mat`, `MARGIN`, `FUN`)

Report unclear tasks in the forum

Highlight the topics from this lesson in your RefCard

## Matrix multiplication

```
m <- matrix(1:6 , nrow=2, ncol=3, byrow=TRUE)
m
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
```

```
n <- matrix(rep(1:5,each=3), ncol=5)
```

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

---

```
m %*% n
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    6   12   18   24   30
## [2,]   15   30   45   60   75
```

?"%\*%" # use quotation marks to open the documentation

Data.frames can be converted:

```
d <- data.frame(AA=5:8, BB=6:9)
class(d)
## [1] "data.frame"

dm <- as.matrix(d)      ;  dm
##      AA  BB
## [1,]  5  6
## [2,]  6  7
## [3,]  7  8
## [4,]  8  9

class(dm)
## [1] "matrix" "array"
```



- 1. Intro
  - 2. Basics
  - 3. Objects**
  - 4. Data
  - 5. Graphics
  - 6. Flow control
- 3.1 Data frames
  - 3.2 Matrices
  - 3.3 Lists**
  - 3.4 Arrays
  - 3.5 Data & Object types

## Create a list

Different R objects (vectors, data.frames, ...) can all be stored in a single `list` object:

```
# Create list: an object containing several objects in it
exlist <- list(
  creator="Result of me calling 'list'",
  sequence=345:287, normal=rnorm(10),
  table=data.frame(a=6:3, b=rexp(4)),
  upper=LETTERS, lower=letters,
  numbers=sample(300), farewell="goodbye")

# Show complete list:
exlist

head(exlist,4) # only the first 4 elements
tail(exlist) # the last 6
```



## Structure of a list

```
str(exlist)
## List of 8
## $ creator : chr "Result of me calling 'list'"
## $ sequence: int [1:59] 345 344 343 342 341 340 339 338 ...
## $ normal   : num [1:10] 0.586 0.709 -0.109 -0.453 ...
## $ table    :'data.frame': 4 obs. of  2 variables:
##   ..$ a: int [1:4] 6 5 4 3
##   ..$ b: num [1:4] 0.92 0.415 0.289 1.252
## $ upper    : chr [1:26] "A" "B" "C" ...
## $ lower    : chr [1:26] "a" "b" "c" ...
## $ numbers  : int [1:300] 212 12 259 14 141 192 148 106 ...
## $ farewell: chr "goodbye"
```



## Structure of top level only

```
str(exlist, max.level=1) # less info on 'table'  
## List of 8  
## $ creator : chr "Result of me calling 'list'"  
## $ sequence: int [1:59] 345 344 343 342 341 340 339 338 ...  
## $ normal   : num [1:10] 0.586 0.709 -0.109 -0.453 ...  
## $ table    :'data.frame': 4 obs. of 2 variables:  
## $ upper    : chr [1:26] "A" "B" "C" ...  
## $ lower    : chr [1:26] "a" "b" "c" ...  
## $ numbers  : int [1:300] 212 12 259 14 141 192 148 106 ...  
## $ farewell: chr "goodbye"
```



## Indexing I: single / double square brackets

```
exlist[3]
## $normal
## [1] 0.5855288 0.7094660 -0.1093033 -0.4534972
## [5] 0.6058875 -1.8179560 0.6300986 -0.2761841
## [9] -0.2841597 -0.9193220

exlist[[3]]
## [1] 0.5855288 0.7094660 -0.1093033 -0.4534972
## [5] 0.6058875 -1.8179560 0.6300986 -0.2761841
## [9] -0.2841597 -0.9193220

class(exlist[3])    # list
class(exlist[[3]])  # numeric

exlist[3] -> list (with 1 element)
exlist[[3]] -> the element as such (here: vector with 10 numbers).
```



## Indexing II: several elements

```
exlist[4:5]      # -> list
## $table
##   a          b
## 1 6 0.9196347
## 2 5 0.4149638
## 3 4 0.2890853
## 4 3 1.2524611
##
## $upper
## [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L"
## [13] "M" "N" "O" "P" "Q" "R" "S" "T" "U" "V" "W" "X"
## [25] "Y" "Z"
```



## Indexing III: by names

```
exlist$farewell # use Rstudio autocompletion :)  
## [1] "goodbye"
```

```
exlist$Farewell  
## NULL
```

No error message!! -> Observe spelling

```
names(exlist) # like names(vector) / rownames(dataframe)  
## [1] "creator"    "sequence"   "normal"     "table"  
## [5] "upper"       "lower"       "numbers"    "farewell"
```

```
names(exlist)[2] <- "newName"
```



## Indexing IV: change / remove elements

```
exlist[c(3,6)] <- c(77777777, 3333333)  
  
head(exlist, 3)  
## $creator  
## [1] "Result of me calling 'list'"  
##  
## $newName  
## [1] 345 344 343 342 341 340 339 338 337 336 335 334  
## [13] 333 332 331 330 329 328 327 326 325 324 323 322  
## [25] 321 320 319 318 317 316 315 314 313 312 311 310  
## [37] 309 308 307 306 305 304 303 302 301 300 299 298  
## [49] 297 296 295 294 293 292 291 290 289 288 287  
##  
## $normal  
## [1] 77777777
```

```
exlist$anElement <- NULL # like with data.frames
```



## lapply : apply functions to lists

```
class(exlist[[1]])  
## [1] "character"  
class(exlist[[2]])  
## [1] "integer"
```

Execute the function 'class' for each element of the list:

```
lapply(X=exlist, FUN=class) # -> list  
  
## $creator  
## [1] "character"  
##  
## $newName  
## [1] "integer"  
##  
## $normal  
## [1] "numeric"  
##  
## $table  
## [1] "data.frame"  
  
## $upper  
## [1] "character"  
##  
## $lower  
## [1] "numeric"  
##  
## $numbers  
## [1] "integer"  
##  
## $farewell  
## [1] "character"
```



## sapply : apply functions, simplify the result

```
sapply(X=exlist, FUN=class) # -> vector
##      creator      newName      normal       table
## "character"    "integer"    "numeric"   "data.frame"
##      upper       lower      numbers     farewell
## "character"    "numeric"    "integer"   "character"
```

Mixed data types are converted (as with matrices):

```
sapply(X=unname(exlist)[-1], FUN=max) # num -> char
## [1] "345"        "77777777"  "6"          "Z"
## [5] "3333333"   "300"        "goodbye"
```

If `FUN(a_list[[1]])` and `FUN(a_list[[2]])` provide differently dimensioned results, the result is not simplified and remains a list.

A function in `tapply` (2.7) must return an output of length 1 for the result to be simplified. `sapply` will also simplify the result if each output is a longer vector or matrix - as long as all lengths are the same.



## Further arguments passed to function

```
L3 <- list(AB=c(6,9,2,6), BC=1:8, CD=c(-3,2) )  
quantile(L3$AB)  
## 0% 25% 50% 75% 100%  
## 2.00 5.00 6.00 6.75 9.00  
quantile(L3$AB, probs=0.3) # 30% of values below 5.6  
## 30%  
## 5.6  
sapply(L3, quantile)  
## AB BC CD  
## 0% 2.00 1.00 -3.00  
## 25% 5.00 2.75 -1.75  
## 50% 6.00 4.50 -0.50  
## 75% 6.75 6.25 0.75  
## 100% 9.00 8.00 2.00  
  
sapply(L3, quantile, probs=0.3)  
## AB.30% BC.30% CD.30%  
## 5.6 3.1 -1.5  
  
sapply(L3, quantile, probs=0.3, names=FALSE)  
## AB BC CD  
## 5.6 3.1 -1.5
```



## sapply / lapply with vectors

```
sample(x=1:100, size=5)
## [1] 31 9 59 53 73
```

The input does not have to be a list, a vector is also passed to 'FUN' element by element:

```
lapply(X=c(6,2,13), FUN=sample, x=1:100)
## [[1]]
## [1] 87 20 59 51 57 10
##
## [[2]]
## [1] 6 41
##
## [[3]]
## [1] 32 82 56 49 23 33 10 99 72 96 63 62 8
```

## lapply/sapply example from practice



## sapply / lapply with dataframes

Since a `data.frame` internally is a `list`, instead of

```
apply(rock, MARGIN=2, median)
```

```
##           area         peri        shape        perm
## 7487.000000 2536.195000    0.198862 130.500000
```

you may use:

```
sapply(rock, median) # nice and short
##           area         peri        shape        perm
## 7487.000000 2536.195000    0.198862 130.500000
```

For mixed data types, this is mandatory:

```
str(ToothGrowth)
## 'data.frame': 60 obs. of 3 variables:
##   $ len : num  4.2 11.5 7.3 5.8 6.4 10 11.2 11.2 5.2 7 ...
##   $ supp: Factor w/ 2 levels "OJ","VC": 2 2 2 2 2 2 2 2 2 2 ...
##   $ dose: num  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 ...
apply(ToothGrowth, MARGIN=2, class) # first coerces all!
##           len         supp        dose
## "character" "character" "character"
sapply(ToothGrowth, class)
##           len         supp        dose
## "numeric"  "factor"  "numeric"
```



### create and analyze lists, loop over inputs:

- ▶ `list`, `[[`, `[`, `$`, `names`
- ▶ `str`, `head` / `tail`,
- ▶ `lapply` (`X`, `FUN`, ...)
- ▶ `sapply` (mixed data types as output are converted when simplifying, but dimension of output must be the same for each function call)

Report unclear tasks in the forum

Highlight the topics from this lesson in your RefCard



Simplify lists to matrices: `sapply` and `I`

List with vectors of equal length:

```
L2 <- list(3:7, 8:4, 5:9)
```

```
L2
```

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

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

```
##
```

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

```
## [1] 8 7 6 5 4
```

```
##
```

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

```
## [1] 5 6 7 8 9
```

`I` returns the input 'as is':

```
I(7:3)
```

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

```
sapply(L2, I)
```

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

```
## [1,] 3 8 5
```

```
## [2,] 4 7 6
```

```
## [3,] 5 6 7
```

```
## [4,] 6 5 8
```

```
## [5,] 7 4 9
```

Result of `I` for each element is a vector with 5 numbers, so these are bundled into a matrix (column by column).

## Simplify lists to dataframes: 12df

Vectors of different length cannot be combined by `sapply` :

```
L3 <- list(AB=c(6,9,2,6), BC=1:8, CD=c(-3,2) )  
sapply(L3, I)  
## $AB  
## [1] 6 9 2 6  
##  
## $BC  
## [1] 1 2 3 4 5 6 7 8  
##  
## $CD  
## [1] -3  2
```

```
berryFunctions::12df(L3)  
##      V1 V2 V3 V4 V5 V6 V7 V8  
## AB  6   9   2   6 NA NA NA NA  
## BC  1   2   3   4   5   6   7   8  
## CD -3   2 NA NA NA NA NA NA
```

Vectors are padded with NAs, names are kept

```
L3
## $AB
## [1] 6 9 2 6
##
## $BC
## [1] 1 2 3 4 5 6 7 8
##
## $CD
## [1] -3 2
```

```
unlist(L3)
```

```
## AB1 AB2 AB3 AB4 BC1 BC2 BC3 BC4 BC5 BC6 BC7 BC8
##   6    9    2    6    1    2    3    4    5    6    7    8
## CD1 CD2
##   -3    2
```

```
unlist(L3, use.names=FALSE)
```

```
## [1] 6 9 2 6 1 2 3 4 5 6 7 8 -3 2
```

Argument `recursive` for nested lists (where individual elements are in turn a list).

## sapply : nested subsetting

```
L3
## $AB
## [1] 6 9 2 6
##
## $BC
## [1] 1 2 3 4 5 6 7 8
##
## $CD
## [1] -3 2
```

```
L3$AB[4]          # classic use of square brackets
## [1] 6
"["(L3$AB, 4)    # Calling brackets in function syntax
## [1] 6

sapply(L3, "[", 4) # -> Vector with each fourth element
## AB BC CD
## 6 4 NA
```



## sapply : check results

For results of different dimensions, the sapply output is not simplified and remains a list.

This happens without warning. In 'production code', the output should be checked:

```
if(is.list(      sapply(exlist, dim)      ))  
  warning("sapply could not bundle the results.")
```

- 1. Intro
  - 2. Basics
  - 3. Objects**
  - 4. Data
  - 5. Graphics
  - 6. Flow control
- 3.1 Data frames
  - 3.2 Matrices
  - 3.3 Lists
  - 3.4 Arrays**
  - 3.5 Data & Object types

## Create an array (world population by age, UN data)

```
names <- list(region=c("Africa", "Asia", "Europe", "Oceania"),
              age=c("U25", "25-69", "70+"),
              year=c(1950, 1980, 2020, 2050, 2100))
# U25          25-69          70+
# AF,AS,EU,OC, AF,AS,EU,OC, AF,AS,EU,OC
values <- c(60,55,44,46, 38,43,51,50,  2, 2, 5, 4, # 1950
           64,57,39,48, 35,40,54,47,  2, 3, 8, 5, # 1980
           60,39,26,38, 38,56,60,53,  2, 5,13, 9, # 2020
           50,30,25,34, 46,57,54,53,  4,13,22,13, # 2050
           36,25,25,29, 54,53,51,53, 10,22,25,19) # 2100
humans <- array(values, dim=c(4,3,5), dimnames=names)

str(humans)
##  num [1:4, 1:3, 1:5] 60 55 44 46 38 43 51 50 ...
##  - attr(*, "dimnames")=List of 3
##    ..$ region: chr [1:4] "Africa" "Asia" "Europe" ...
##    ..$ age   : chr [1:3] "U25" "25-69" "70+"
##    ..$ year  : chr [1:5] "1950" "1980" "2020" ...

dim(humans)
## [1] 4 3 5
```



## Array is like a matrix with several levels (layers)

```
humans[, , 1:2]
## , , year = 1950
##
##          age
## region    U25 25-69 70+
## Africa    60   38   2
## Asia      55   43   2
## Europe    44   51   5
## Oceania   46   50   4
##
## , , year = 1980
##
##          age
## region    U25 25-69 70+
## Africa    64   35   2
## Asia      57   40   3
## Europe    39   54   8
## Oceania   48   47   5
```



## Array indexing

```
some_array[x,y,z,t, ...] # generic syntax
```

```
humans["Europe",,] # All values for EU, new matrix arrangement
```

```
## year
```

```
## age 1950 1980 2020 2050 2100
```

```
## U25 44 39 26 25 25
```

```
## 25-69 51 54 60 54 51
```

```
## 70+ 5 8 13 22 25
```

```
humans[,2,] # second age group by position (index)
```

```
## year
```

```
## region 1950 1980 2020 2050 2100
```

```
## Africa 38 35 38 46 54
```

```
## Asia 43 40 56 57 53
```

```
## Europe 51 54 60 54 51
```

```
## Oceania 50 47 53 53 53
```

```
humans["Asia","70+",] # time series of seniors in Asia
```

```
## 1950 1980 2020 2050 2100
```

```
## 2 3 5 13 22
```



## Array Viewing

Display individual slices in the Rstudio Data Viewer panel:

```
View(humans[, , 1]) ; View(humans[, , 2])
```

# View all layers in Rstudio:

```
sapply(dimnames(humans)[[3]], function(i)
  View(humans[, , i], title=paste0("humans[ , , ", i, "]")))
```

# Same content as a for loop with less brackets:

```
for(i in dimnames(humans)[[3]])
  View(humans[, , i], title=paste0("humans[ , , ", i, "]"))
```



## Aggregate arrays I

```
names(dimnames(humans))
```

```
## [1] "region" "age"     "year"
```

MARGIN determines the dimension(s) to be kept.

Over the other dimensions is to be aggregated.

```
# median per year
```

```
apply(humans, MARGIN=3, FUN=median)
```

```
## 1950 1980 2020 2050 2100
```

```
## 43.5 39.5 38.0 32.0 27.0
```

```
# median per region and age (aggregated over time):
```

```
apply(humans, 1:2, median)
```

```
##           age
```

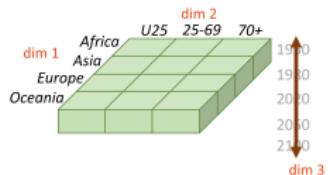
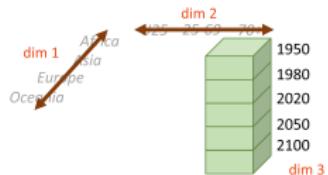
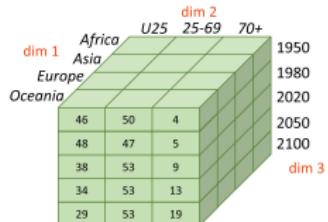
```
## region    U25 25-69 70+
```

```
##   Africa    60   38   2
```

```
##   Asia      39   53   5
```

```
##   Europe    26   54  13
```

```
##   Oceania   38   53   9
```



## Aggregate arrays II

```
names(dimnames(humans))
## [1] "region" "age"     "year"

apply(humans, c(1,3), max) # max aggregated over age
##                 year
## region      1950 1980 2020 2050 2100
##   Africa      60   64   60   50   54
##   Asia        55   57   56   57   53
##   Europe      51   54   60   54   51
##   Oceania     50   48   53   53   53
```

# *MARGIN* can also be given as names:

```
apply(humans, c("region", "year" ), max)
##                 year
## region      1950 1980 2020 2050 2100
##   Africa      60   64   60   50   54
##   Asia        55   57   56   57   53
##   Europe      51   54   60   54   51
##   Oceania     50   48   53   53   53
```



## Aggregate arrays III

```
dimnames(humans)
## $region
## [1] "Africa"   "Asia"      "Europe"    "Oceania"
##
## $age
## [1] "U25"      "25-69"     "70+"
##
## $year
## [1] "1950"     "1980"     "2020"     "2050"     "2100"
```

*# maximum per year without age group 25-69:*

```
apply(humans[,-2,], 3, max)
## 1950 1980 2020 2050 2100
##   60   64   60   50   36
```

*# To find out the 2 if there were many groups:*

```
which(dimnames(humans)$age=="25-69")
## [1] 2
```



### **multidimensional matrices:**

- ▶ `array` , `ar[row, column, layer, slice, level, ...]`
- ▶ `dim` , `dimnames`
- ▶ `apply` (`MARGIN=1:2, FUN`)

Report unclear tasks in the forum

Highlight the topics from this lesson in your RefCard



- 1. Intro
  - 2. Basics
  - 3. Objects
  - 4. Data
  - 5. Graphics
  - 6. Flow control
- 
- 3.1 Data frames
  - 3.2 Matrices
  - 3.3 Lists
  - 3.4 Arrays
  - 3.5 Data & Object types

## Data types I

R can handle data of a variety of types: Numbers like `42.6`, characters like `"R rocks"`, complex numbers like `8+4i`, categories with `as.factor(c("blue", "red"))`, logical or boolean values `c(TRUE, FALSE)` and a few others.

Functions and operators may not be able to handle all:

```
number <- "1.3"  
2 * number      # error, because 'number' is a character string  
## Error in 2 * number: non-numeric argument to binary  
operator
```

```
2 * as.numeric(number)  
## [1] 2.6
```

```
number           # 'number' itself is still a charstring  
## [1] "1.3"  
is.numeric(number)  
## [1] FALSE  
mode(number) # internally more precise: typeof  
## [1] "character"
```



## Data types II

When combining several data types, they are transformed (without warning message):

```
c(42, "67")  
## [1] "42" "67"
```

```
c(42, TRUE)  
## [1] 42  1
```

because a vector in R always consists of a single data type (for all entries).



## Overview: data types (in order of coercion)

`c(TRUE, 7) -> 1,7 ; c(7, "z") -> "7" "z"`

[adv-r.had.co.nz/Data-structures](http://adv-r.had.co.nz/Data-structures)

Description	example	<code>typeof</code>	<code>class</code>
empty set	<code>NULL</code>	<code>NULL</code>	<code>NULL</code>
not available	<code>NA</code>	<code>logical</code>	<code>logical</code>
logical	<code>c(T, F, FALSE, TRUE)</code>	<code>logical</code>	<code>logical</code>
category	<code>factor("left")</code>	<code>integer</code>	<b>factor</b>
integer number	<code>4:6 ; 7L</code>	<code>integer</code>	<code>integer</code>
decimal	<code>8.7</code>	<code>double</code>	<b>numeric</b>
complex	<code>5+3i</code>	<code>complex</code>	<code>complex</code>
character string	<code>"R rocks"</code>	<code>character</code>	<code>character</code>
date	<code>as.Date("2020-06-26")</code>	<code>double</code>	<b>Date</b>
time	<code>Sys.time()</code>	<code>double</code>	<b>POSIXct</b>
function	<code>ncol</code>	<code>closure</code>	<b>function</b>

`as.character(3.14)` converts a data type ; `is.integer(4:6)` checks.

`str` shows an abbreviation of `class`. Other rare typeofs: raw, environment, promise, ...

`mode` ≈ `typeof`, but integer/double -> numeric ; closure/special/builtin -> function.

When mixing date/time with others, the order of appearance determines the output class.



## Determine the `class` of an object

R is an object-oriented language, i.e. everything is an object (even a function is an object).

```
v <- c(6.3,2); df <- data.frame(1:4,4:1); m <- matrix(1:6)
class(v)
## [1] "numeric"
class(df)
## [1] "data.frame"
class(m)
## [1] "matrix" "array"
```

For rather internal object properties there are also `typeof` and `mode`:

```
typeof(v)
## [1] "double"
typeof(df)
## [1] "list"
typeof(m)
## [1] "integer"
```

```
mode(v)
## [1] "numeric"
mode(df)
## [1] "list"
mode(m)
## [1] "numeric"
```



## Examine objects

Objects have classes and methods exist for those.

`str` e.g. shows different outputs, depending on the `class`:

```
str(v)
## num [1:2] 6.3 2
str(df)
## 'data.frame': 4 obs. of 2 variables:
## $ X1.4: int 1 2 3 4
## $ X4.1: int 4 3 2 1
str(m)
## int [1:6, 1] 1 2 3 4 5 6
```

Other methods are specific to individual object types:

```
length(v) # for vectors
## [1] 2
colnames(df) # for DFs
## [1] "X1.4" "X4.1"
```

```
dim(m)
## [1] 6 1
which(v > 4) # for logicals
## [1] 1
```



## Overview: object types

Object	example	typeof	class
vector	<code>c(pi, 2)</code> see <b>data types</b>	...	...
matrix	<code>matrix(9:15, ncol=2)</code>	...	matrix
array	<code>array(letters, dim=c(2,6,4))</code>	...	array
dataframe	<code>data.frame(4:5,B=c("a","b"))</code>	list	data.frame
list	<code>list(el1=7:15, el2="big")</code>	list	list
function	<code>function(x) 12+0.5*x</code>	closure	function
...	<code>lm(b ~ a)</code>	list	lm

A `matrix` consists of one single data type. If one element is changed, all are converted (in `order of coercion`).

A `data.frame` can have multiple data types, one type per column.

A `list` can combine anything, even other lists.

`is.atomic(Object)` returns TRUE (`vector, matrix, array`) or FALSE

`is.vector(Object)` returns TRUE only for certain data types

`as.matrix(Object)` converts the class of an object by force.



### overview of data and object types:

- ▶ Data types: numeric (integer/double), character, logical, complex
- ▶ `class`, `mode`, `typeof`
- ▶ Objekt types: vector, matrix, array, data.frame, list
- ▶ Overviews

Report unclear tasks in the forum

Highlight the topics from this lesson in your RefCard



Be careful with `is.vector`:

- FALSE for `factor` / `date` / `time` - TRUE for `list` - see SO thread

Use `is.atomic(x) && is.null(dim(x))`

## Data types

	<code>is.vector</code>	<code>is.atomic</code>
<code>null</code>	FALSE	TRUE
<code>na</code>	TRUE	TRUE
<code>logi</code>	TRUE	TRUE
<code>factor</code>	FALSE	TRUE
<code>int</code>	TRUE	TRUE
<code>double</code>	TRUE	TRUE
<code>complex</code>	TRUE	TRUE
<code>char</code>	TRUE	TRUE
<code>date</code>	FALSE	TRUE
<code>time</code>	FALSE	TRUE
<code>func</code>	FALSE	FALSE

## Object types

	<code>is.vector</code>	<code>is.atomic</code>
<code>vector</code>	T/F	TRUE
<code>matrix</code>	FALSE	TRUE
<code>array</code>	FALSE	TRUE
<code>data.frame</code>	FALSE	FALSE
<code>list</code>	TRUE	FALSE
<code>function</code>	FALSE	FALSE
<code>Im</code>	FALSE	FALSE

```
class(m)
## [1] "matrix" "array"
```

```
# Do not use:
if( class(m) == "matrix" ) message("m is a matrix")
## Error in if (class(m) == "matrix") message("m is a
matrix"): the condition has length > 1
```

```
# Do use:
if( inherits(m, "matrix") ) message("m is a matrix")

## m is a matrix
```

- 1. Intro
  - 2. Basics
  - 3. Objects
  - 4. Data**
  - 5. Graphics
  - 6. Flow control
- 4.1 Read
  - 4.2 Merge
  - 4.3 Missing data
  - 4.4 Data sources

## Read in files (import data)

General syntax:

```
tabular_data <- read.table(file="filename.txt")
```

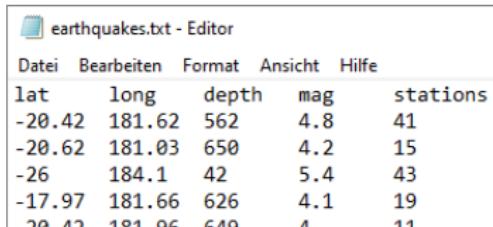
Always check imported data! (90% of later errors by skipping this)

```
data_object_name <- read.table("File.txt")
str(data_object_name)
```

### Example

# Generate a tabstop separated textfile to be read:

```
berryFunctions::write.tab(quakes[1:15,], file="earthquakes.txt", open=FALSE)
```



lat	long	depth	mag	stations
-20.42	181.62	562	4.8	41
-20.62	181.03	650	4.2	15
-26	184.1	42	5.4	43
-17.97	181.66	626	4.1	19
20.12	181.06	610	4	11

```
eq <- read.table("earthquakes.txt", header=TRUE)
```

# Frequently needed arguments:

```
read.table("File.txt", header=TRUE, dec=",", sep="\t")
```



## Common arguments for `read.table`

```
read.table(file="FileName.txt", # can also be a URL
header=TRUE,                 # read first row as column names
dec=",",                      # comma as decimal mark
sep="_",                      # _ as column separator ("\t" -> tabstop)
fill=TRUE,                     # fill incomplete rows with NAs at the end
skip=12,                       # ignore first 12 lines (e.g. with metadata)
comment.char="%",             # ignore content after % (default: # like R)
na.strings=c(-999, "NN"),      # identify NA entries (missing values)
stringsAsFactors=FALSE,        # do not convert charstrings to factors
text="1,2,3",                  # small sample dataset in script
... )                          # other arguments, see ?read.table
```

The `stringsAsFactors` default is `FALSE` since R version 4.0.0 (2020-04-24).

```
read.table(header=TRUE, sep=",", text="
Example, Column
meaning, 42
bond, 007")
##   Example Column
## 1 meaning      42
## 2     bond      7
```



## Alternatives to `read.table`

- ▶ `read.csv()` : comma separated values (`read.table` with different default values)
- ▶ `read.fwf()` : constant column widths (fixed width formatted data)
- ▶ `readLines()` : Read rows as charstring vector
- ▶ `scan()` : rarely needed (also used at the core of `read.table`)

## Complex files:

- ▶ `readxl::read_excel()` : Excel files, see [github.com/tidyverse/readxl](https://github.com/tidyverse/readxl)
- ▶ `readBin()` : for binary files
- ▶ `raster::raster` / `rgdal::readGDAL` : geodata (grd, asc, tif)
- ▶ `ncdf4::nc_open()` + `ncdf4::ncvar_get` : NetCDF files



## Common error messages from `read.table`

Error in scan(file, what, nmax, sep, dec, quote, skip, nlines, na.strings, : line _ did not have _ elements	Correctly specify <code>header</code> , <code>sep</code> or <code>fill</code> .
Warning message: <code>In read.table("_.txt", ...)</code> : incomplete final line found by readTableHeader on 'C:\_.txt'	Manually add line break ( <code>ENTER</code> ) at the end of the file). line break = line feed (LF) = newline = carriage return (CR).
<code>str(importedDataframe)</code> shows "factor" in some numeric columns	Check file. If needed, set <code>dec</code> correctly

Severity of messages:

**error**: Function aborts. Problem must be fixed.

**warning**: Function continues, potentially erroneous. Ignore only if reason known but not solvable.

```
# Read data with thousands separator "3,590.18":  
df$column <- as.numeric( gsub(",",".", df$column) )
```

```
# Risky: replace commas with periods as decimal marker:  
df$column <- as.numeric( gsub(",",".", df$column) )
```



## Use folder-specific files

R reads and writes files in the working directory.

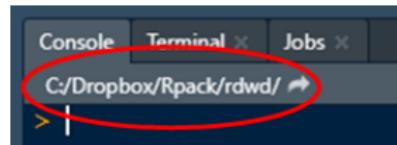
Quick and dirty:

```
setwd("C:/Users/berry/Project_XY") # set working directory
```

Windows users: \ is a nuisance of Microsoft. When copying a path from the file browser (Explorer), replace all with / .

Generally, **work with Rstudio projects**, see [slide](#) in section 1.2.

Show current working directory with `getwd()`



Always use relative file names, i.e. files that are present in the WD:

```
table <- read.table("subfolder/file.txt")
```

```
dir() # Show available file names in the WD
```

## Manage folders

helpful functions:

```
getwd() # Show current working directory (WD)
```

```
setwd("../") # Set WD one level up
```

```
dir() # List files / folders in WD
```

```
dir(recursive=TRUE) # Also show items in subfolders
```

```
dir("../other_subfolder") # In different folder
```

```
file.create() # Create files
```

```
file.rename() # Rename files
```

```
file.remove() # Delete files, see also: unlink()
```

```
file.copy() # Copy files
```

```
file.exists() # Check files for existence
```

```
dir.create() # create folder
```

```
dir.exists() # Check folder presence
```



## Writing data to disc

```
write.table(x=mynewdata, file="output.txt",
             quote=FALSE, row.names=FALSE,
             fileEncoding="UTF-8")
```

Side note: Never change data manually:

```
newtable <- edit(olddata)
fix(mytable) # Does not even keep the old data
```

This would not be reproducible. Better write a script:

```
mytable[265, "temperature"] <- 17.53 # original 175.3, probably typo
mytable[1:24, "readings"] <- NA # wrong measuring method on first day
```



### read file content into R:

- ▶ `setwd`, `dir`, Rstudio projects
- ▶ `file.rename`, `file.exists`, `dir.create`, ...
- ▶ `read.table`(`file`, `header`, `dec`, `sep`, `skip`), `write.table`
- ▶ `read.csv`, `read.fwf`, `readxl::read_excel`, `scan`
- ▶ `str`

Report unclear tasks in the forum

Highlight the topics from this lesson in your RefCard



- 1. Intro
  - 2. Basics
  - 3. Objects
  - 4. Data**
  - 5. Graphics
  - 6. Flow control
- 4.1 Read
  - 4.2 Merge
  - 4.3 Missing data
  - 4.4 Data sources

`cbind / rbind` : combine tabular data unchanged and unchecked

`cbind` : bind columns beside each other

`rbind` : merge rows below each other

dimensions and names must be correct

```
exdf <- data.frame(number=11:13, group=letters[1:3])  
  
cbind(data.frame(Poisson=rpois(3,80)), exdf) # column-bind  
##   Poisson number group  
## 1      92     11     a  
## 2      74     12     b  
## 3      80     13     c  
  
row <- data.frame(number=2, group="new") ; row  
##   number group  
## 1      2     new  
  
rbind(exdf, row) # row-bind: column names must be equal  
##   number group  
## 1      11     a  
## 2      12     b  
## 3      13     c  
## 4      2     new
```

## Combine matrices I: rbind

```
p <- matrix(11:16, ncol=3) # 2 x 3
q <- matrix(21:32, ncol=4) # 3 x 4
r <- matrix(31:39, ncol=3) # 3 x 3
```

p	q	r
11 13 15	21 24 27 30	31 34 37
12 14 16	22 25 28 31	32 35 38
23 26 29 32	33 36 39	

```
rbind(p,r) # row-bind: number of columns must be equal
```

```
##      [,1] [,2] [,3]
## [1,]    11   13   15
## [2,]    12   14   16
## [3,]    31   34   37
## [4,]    32   35   38
## [5,]    33   36   39
```

```
rbind(p,q) # Error: 3 and 4 columns
```

```
## Error in rbind(p, q): number of columns of matrices
must match (see arg 2)
```



## Combine matrices II: cbind

```
p <- matrix(11:16, ncol=3) # 2 x 3
q <- matrix(21:32, ncol=4) # 3 x 4
r <- matrix(31:39, ncol=3) # 3 x 3
```

p	q	r
11 13 15	21 24 27 30	31 34 37
12 14 16	22 25 28 31	32 35 38
23 26 29 32	33 36 39	

```
cbind(q,r) # column-bind: nrow must both be the same
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
## [1,]    21   24   27   30   31   34   37
## [2,]    22   25   28   31   32   35   38
## [3,]    23   26   29   32   33   36   39
```

```
cbind(p,r) # Error: 2 and 3 rows
## Error in cbind(p, r): number of rows of matrices must
match (see arg 2)
```



Join / connect / link datasets: `merge` |

```
participants <- read.table(  
  header=TRUE, text="  
name  age  
Alice 27  
Berry 32  
Chris 14  
David 45")
```

```
subjects <- read.table(  
  header=TRUE, text="  
person size weight  
Berry 1.83 82  
Chris 1.43 51  
David 1.75 72  
Elise 1.67 57")
```

Merge information from both datasets (for names that appear in both):

```
merge(participants, subjects, by.x="name", by.y="person")  
##   name age size weight  
## 1 Berry 32 1.83 82  
## 2 Chris 14 1.43 51  
## 3 David 45 1.75 72
```



Join / connect / link datasets: `merge` II

If the names are the same, the columns are selected automatically:

```
colnames(subjects)[1] <- "name"
```

```
merge(participants, subjects)
##   name age size weight
## 1 Berry  32  1.83     82
## 2 Chris  14  1.43     51
## 3 David  45  1.75     72
```

Keep all rows, fill missing data with NAs:

```
merge(participants, subjects, all=TRUE) # all.x / all.y
##   name age size weight
## 1 Alice 27    NA     NA
## 2 Berry 32  1.83     82
## 3 Chris 14  1.43     51
## 4 David 45  1.75     72
## 5 Elise NA  1.67     57
```



## Summary for 4.2 Merge

**combine and merge datasets:**

- ▶ `cbind` , `rbind`
- ▶ `merge` (`by`, `all`)

Report unclear tasks in the forum

Highlight the topics from this lesson in your RefCard



## Combine multiple datasets I: create exemplary data

```
LIST_WITH_DFS <- list(  
  data.frame(date=1:4, AA=11:14),  
  data.frame(date=2:6, BB=22:26),  
  data.frame(date=3:7, CC=33:37)  
)  
LIST_WITH_DFS  
## [[1]]  
##   date AA  
## 1    1 11  
## 2    2 12  
## 3    3 13  
## 4    4 14  
##  
## [[2]]  
##   date BB  
## 1    2 22  
## 2    3 23  
## 3    4 24  
## 4    5 25  
## 5    6 26  
##  
## [[3]]  
##   date CC  
## 1    3 33  
## 2    4 34  
## 3    5 35  
## 4    6 36  
## 5    7 37
```



## Combine multiple datasets II: Reduce &amp; merge

LIST\_WITH\_DFS

```
## [[1]]  
##   date AA  
## 1 1 11  
## 2 2 12  
## 3 3 13  
## 4 4 14
```

## [[2]]  
## date BB  
## 1 2 22  
## 2 3 23  
## 3 4 24  
## 4 5 25  
## 5 6 26

## [[3]]  
## date CC  
## 1 3 33  
## 2 4 34  
## 3 5 35  
## 4 6 36  
## 5 7 37

Reduce(merge, LIST\_WITH\_DFS)

```
##   date AA BB CC  
## 1 3 13 23 33  
## 2 4 14 24 34
```

Reduce(function(...) merge(..., all=TRUE), LIST\_WITH\_DFS)

```
##   date AA BB CC  
## 1 1 11 NA NA  
## 2 2 12 22 NA  
## 3 3 13 23 33  
## 4 4 14 24 34  
## 5 5 NA 25 35  
## 6 6 NA 26 36  
## 7 7 NA NA 37
```

## Combine multiple datasets III: do.call & rbind

```
LIST_WITH_DFS <- lapply(LIST_WITH_DFS, function(x)
                           {colnames(x) <- c("date", "XX"); x})
```

LIST\_WITH\_DFS

```
## [[1]]
##   date XX
## 1   1 11
## 2   2 12
## 3   3 13
## 4   4 14
```

```
## [[2]]
##   date XX
## 1   2 22
## 2   3 23
## 3   4 24
## 4   5 25
## 5   6 26
```

```
## [[3]]
##   date XX
## 1   3 33
## 2   4 34
## 3   5 35
## 4   6 36
## 5   7 37
```

```
do.call(rbind, LIST_WITH_DFS)
```

```
##   date XX
## 1   1 11
## 2   2 12
## 3   3 13
## 4   4 14
## 5   2 22
## 6   3 23
## 7   4 24
## 8   5 25
## 9   6 26
## 10  3 33
## 11  4 34
## 12  5 35
## 13  6 36
## 14  7 37
```

- 1. Intro
  - 2. Basics
  - 3. Objects
  - 4. Data**
  - 5. Graphics
  - 6. Flow control
- 4.1 Read
  - 4.2 Merge
  - 4.3 Missing data**
  - 4.4 Data sources

## Missing values (NA = not available)

```
df <- data.frame(A=11:20, B=21:30)
df[3,2] <- NA
```

```
df
##      A   B
## 1    11  21
## 2    12  22
## 3    13  NA
## 4    14  24
## 5    15  25
## 6    16  26
## 7    17  27
## 8    18  28
## 9    19  29
## 10   20  30
```

```
is.na(df)
##           A     B
## [1,] FALSE FALSE
## [2,] FALSE FALSE
## [3,] FALSE  TRUE
## [4,] FALSE FALSE
## [5,] FALSE FALSE
## [6,] FALSE FALSE
## [7,] FALSE FALSE
## [8,] FALSE FALSE
## [9,] FALSE FALSE
## [10,] FALSE FALSE
```

```
na.omit(df)
##      A   B
## 1    11  21
## 2    12  22
## 4    14  24
## 5    15  25
## 6    16  26
## 7    17  27
## 8    18  28
## 9    19  29
## 10   20  30
```



## na.rm-argument

```
df
##      A   B
## 1  11 21
## 2  12 22
## 3  13 NA
## 4  14 24
## 5  15 25
## 6  16 26
## 7  17 27
## 8  18 28
## 9  19 29
## 10 20 30
```

```
mean(df$A)
## [1] 15.5
```

```
mean(df$B)
## [1] NA
```

Average of non-NA entries:

```
mean(df$B, na.rm=TRUE)
## [1] 25.77778
```

Close to the original:

```
mean(21:30)
## [1] 25.5
```

Dangerous for sums (grow with number of entries)

```
sum(df$B, na.rm=TRUE)
## [1] 232
```

```
sum(21:30) # na.rm underestimates sum !!
## [1] 255
```



## NA-imputation: Fill missing values with estimates

Fill with mean / median / min / max / ... of the other values:

```
df$B[is.na(df$B)] <- mean(df$B, na.rm=TRUE)
```

```
df$B[is.na(df$B)] <- median(df$B, na.rm=TRUE)
```

Continue last measurement (locf: last observation carried forwards):

```
df[3,2] <- NA
```

```
zoo::na.locf(df$B) # don't forget assignment
```

```
## [1] 21 22 22 24 25 26 27 28 29 30
```

Linear interpolation:

```
approx(df$B, n=length(df$B))$y # y: see next slide
```

```
## [1] 21 22 23 24 25 26 27 28 29 30
```

```
zoo::na.approx(df$B) # less typing :)
```

```
## [1] 21 22 23 24 25 26 27 28 29 30
```

Complex (multivariate) modelling -> Statistics courses

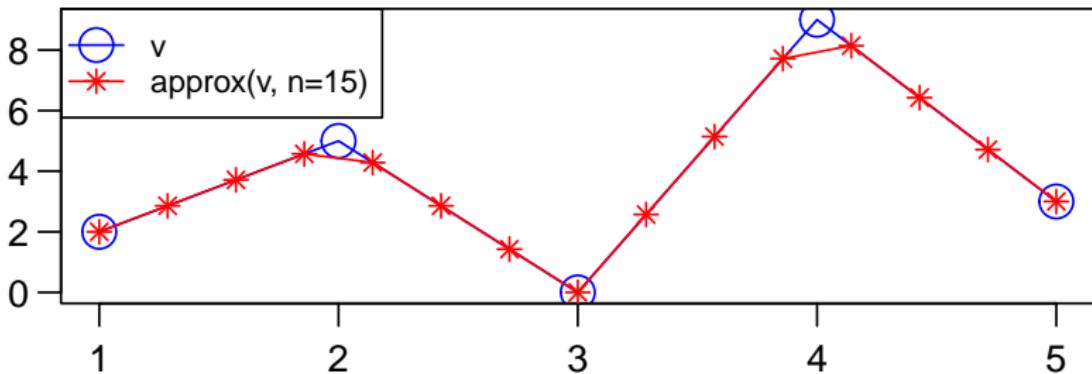


## Hint about `approx`

`approx` always returns the elements `x` and `y`, independent of the original column names.

This allows graphics functions to deal with it directly.

```
v <- c(2,5,0,9,3)
approx(v, n=15)
## $x
## [1] 1.000000 1.285714 1.571429 1.857143 2.142857 2.428571 2.714286 3.000000
## [9] 3.285714 3.571429 3.857143 4.142857 4.428571 4.714286 5.000000
## $y
## [1] 2.000000 2.857143 3.714286 4.571429 4.285714 2.857143 1.428571 0.000000
## [9] 2.571429 5.142857 7.714286 8.142857 6.428571 4.714286 3.000000
```



### manage missing data:

- ▶ NA , is.na
- ▶ na.omit
- ▶ mean / median / sum /...(na.rm=TRUE)

### NA-imputation:

- ▶ x[is.na(x)] <- median(x, na.rm=TRUE)
- ▶ x <- zoo::na.locf(x)
- ▶ x <- zoo::na.approx(x)

Report unclear tasks in the forum

Highlight the topics from this lesson in your RefCard



```
somevalues <- c(42, .007, NA, -77.45, 12/0)
somevalues
## [1] 42.000 0.007       NA -77.450      Inf
somevalues <- c(somevalues, -Inf)

is.na(somevalues)
## [1] FALSE FALSE  TRUE FALSE FALSE FALSE
is.finite(somevalues)
## [1] TRUE  TRUE FALSE  TRUE FALSE FALSE
!is.finite(somevalues)
## [1] FALSE FALSE  TRUE FALSE  TRUE  TRUE
is.infinite(somevalues)
## [1] FALSE FALSE FALSE FALSE  TRUE  TRUE
```



check argument presence in function calls with `missing`

The function `missing` has nothing to do with the NA topic:

```
somefunction <- function(a) missing(a)
somefunction(22) ; somefunction()
## [1] FALSE
## [1] TRUE
```

```
otherfunction <- function(b=33) somefunction(b)
otherfunction(22) ; otherfunction()
## [1] FALSE
## [1] FALSE
```

Conditional (complicated) code in function if an input is not specified:

```
function(arg) if(missing(arg)) ... # fails in other function
function(arg="def") if(arg!="def") ... # fails for vector input
function(arg="def") if(!identical(arg, "def")) ... # ugly
function(arg=NULL) if(is.null(arg)) ...# for use in other funs
```



- 1. Intro
  - 2. Basics
  - 3. Objects
  - 4. Data**
  - 5. Graphics
  - 6. Flow control
- 
- 4.1 Read
  - 4.2 Merge
  - 4.3 Missing data
  - 4.4 Data sources**

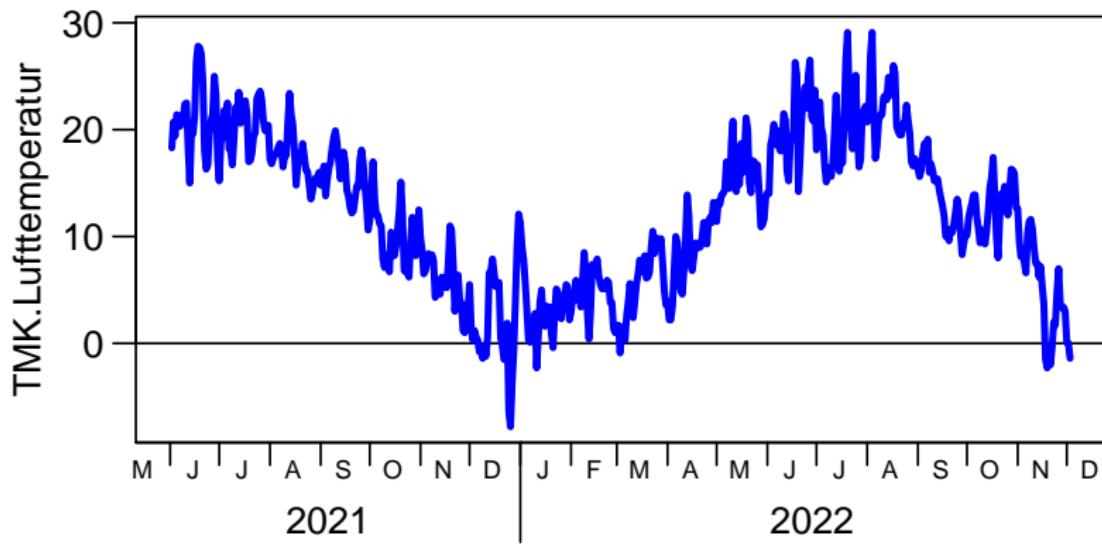
## Online data resources (small selection...)

- ▶ NOAA weather data [ncdc.noaa.gov/pub/data](http://ncdc.noaa.gov/pub/data) (USA)
- ▶ DWD weather data [opendata.dwd.de](http://opendata.dwd.de) (Germany), [rdwd](#)
- ▶ [data.fivethirtyeight.com](http://data.fivethirtyeight.com)
- ▶ [data.unicef.org](http://data.unicef.org) (add .xls as file extension if needed)
- ▶ [github.com/okulbilisim/awesome-datasience#data-sets](http://github.com/okulbilisim/awesome-datasience#data-sets)
- ▶ [github.com/caesar0301/awesome-public-datasets](http://github.com/caesar0301/awesome-public-datasets)
- ▶ [ropensci.org/packages/index.html](http://ropensci.org/packages/index.html)
- ▶ [dataviz.tools/category/data-sources/](http://dataviz.tools/category/data-sources/)
- ▶ StorytellingWithData dataviz challenge 2018
- ▶ [datasetsearch.research.google.com](http://datasetsearch.research.google.com)



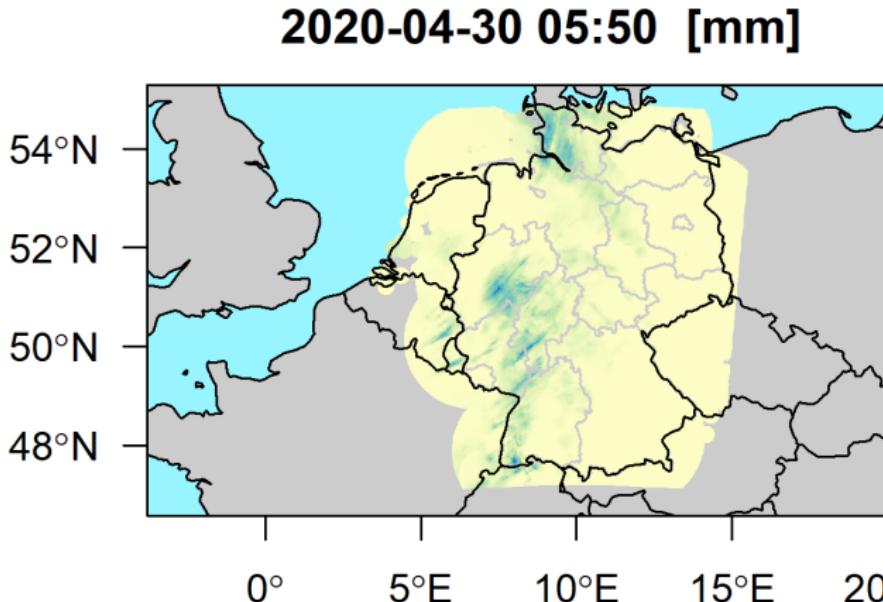
## Import DWD weather data conveniently with `rdwd`

```
# install.packages("rdwd")
library(rdwd)
link <- selectDWD("Potsdam", res="daily", var="kl", per="r")
clim <- dataDWD(link, varnames=TRUE)
plotDWD(clim, "TMK.Lufttemperatur", line0=TRUE)
```



## Import DWD radar data conveniently with rdwd

```
link <- "hourly/radolan/historical/bin/2020/RW202004.tar.gz"  
rad <- dataDWD(link, base=gridbase, joinbf=TRUE, selection=702)  
plotRadar(rad$dat, main="2020-04-30 05:50 [mm] ")
```



## Summary for 4.4 Data sources

### where to find data:

- ▶ Examples of freely available resources
- ▶ `rdwd` for weather data

*There are no exercises for this lesson*

Report unclear tasks in the forum

Highlight the topics from this lesson in your RefCard



- 1. Intro
- 2. Basics
- 3. Objects
- 4. Data
- 5. Graphics**
- 6. Flow control

- 5.1 Scatterplots**
- 5.2 Line plots
- 5.3 Barplots
- 5.4 Low level plotting
- 5.5 Composition
- 5.6 Distribution plots
- 5.7 Export
- 5.8 Outlook

## iris : built-in data set about lily blossoms



Sources: [mirlab.org](http://mirlab.org), [desirableplants.com](http://desirableplants.com), [3.bp.blogspot.com](http://3.bp.blogspot.com)

## iris : built-in data set about lily blossoms

```
head(iris)
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa

str(iris)
## 'data.frame': 150 obs. of  5 variables:
## $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 ...
## $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 ...
## $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 ...
## $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 ...
## $ Species: Factor w/ 3 levels "setosa","versicolor",...: 1 1 1 1 ...

?iris # Detailed documentation
```

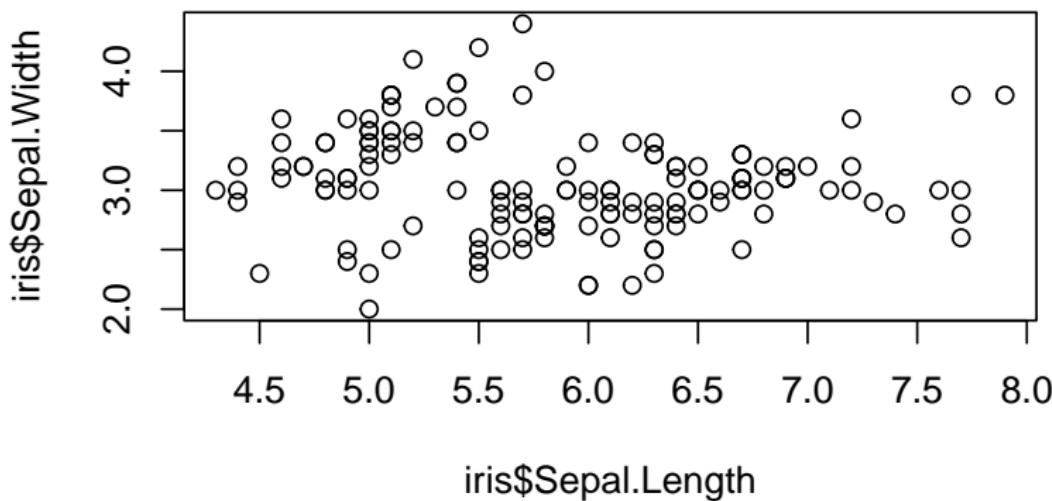


Scatterplots: `plot(x,y)`

```
plot(x=iris$Sepal.Length, y=iris$Sepal.Width)  
# Line breaks with indented code  
# prevent annoying horizontal scrolling
```

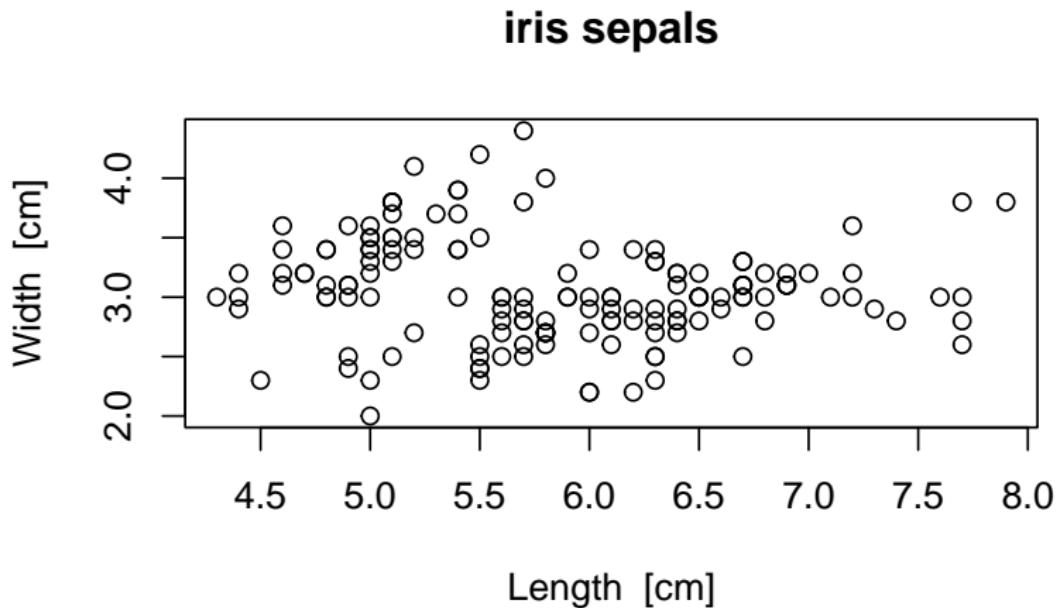
Upon error "Figure margins too large",

drag the Rstudio Plots panel bigger.



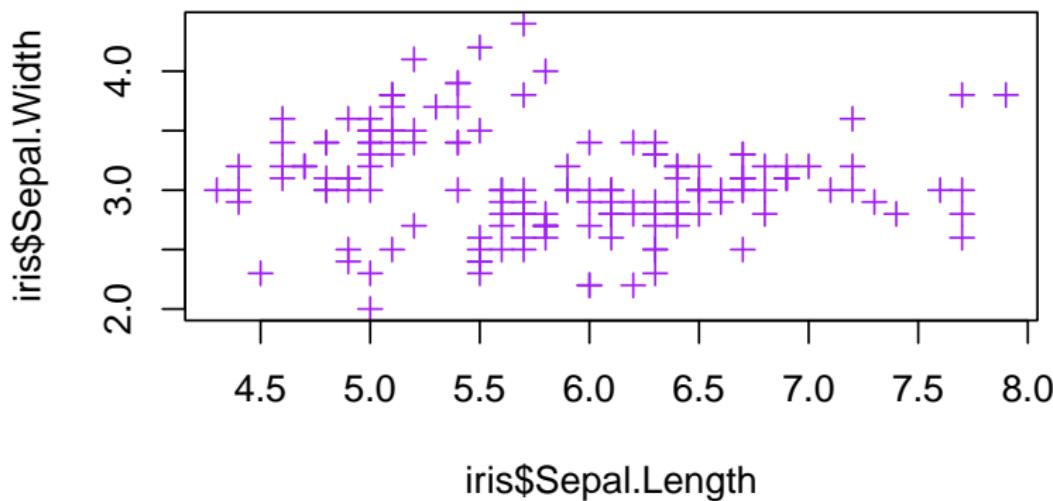
## Axis labels, title

```
plot(iris$Sepal.Length, iris$Sepal.Width,  
      xlab="Length [cm]", ylab="Width [cm]",  
      main="iris sepals") # title
```



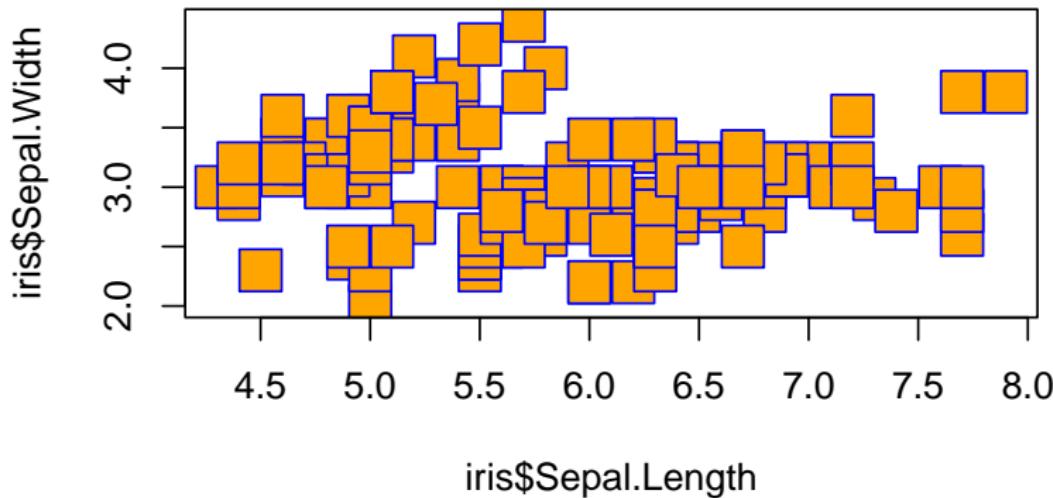
## Colors, symbol I

```
plot(iris$Sepal.Length, iris$Sepal.Width,  
      col="purple", pch=3)  
# col: COLOR, pch: Point Character
```



## Colors, symbol II

```
plot(iris$Sepal.Length, iris$Sepal.Width,  
      pch=22, col="blue", bg="orange", cex=2.8)  
# bg: BackGround, cex: Character Expansion
```



## pch overview (more in bF Anhang)

**plot ( x, y, pch = \_ )**

0	1	2	3	4	5	6	
□	○	△	+	×	◇	▽	
7	8	9	10	11	12	13	14
✉	*	◇	⊕	⊗	田	⊗	□
15	16	17	18	19	20		
■	●	▲	◆	●	●		
21	22	23	24	25	21:25 fillable with bg		
●	■	◆	▲	▼			



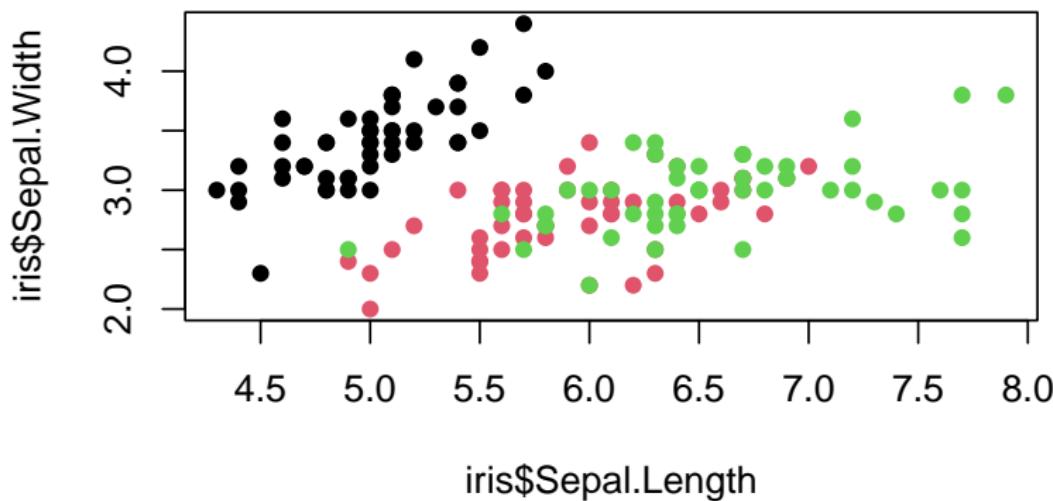
# Colors overview (more in bF Anhang)

1	peru	dimgrey	indianred	darksalmon	mediumpurple
2	pink	hotpink	lawngreen	darkviolet	midnightblue
3	plum	magenta	lightblue	dodgerblue	darkgoldenrod
4	snow	oldlace	lightcyan	ghostwhite	darkslateblue
5	white	skyblue	lightgray	lightcoral	darkslategray
6	azure	thistle	lightgrey	lightgreen	darkslategrey
7	beige	cornsilk	lightpink	mediumblue	darkturquoise
8	black	darkblue	limegreen	papayawhip	lavenderblush
#0399FF	brown	darkcyan	mintcream	powderblue	lightseagreen
grey	coral	darkgray	mistyrose	sandybrown	palegoldenrod
grey1	green	darkgrey	olivedrab	whitesmoke	paleturquoise
grey5	ivory	deeppink	orangered	darkmagenta	palevioletred
grey20	khaki	honeydew	palegreen	deepskyblue	blanchedalmond
grey40	linen	lavender	peachpuff	floralwhite	cornflowerblue
grey60	wheat	moccasin	rosybrown	forestgreen	darkolivegreen
grey80	bisque	navyblue	royalblue	greenyellow	lightgoldenrod
grey99	maroon	seagreen	slateblue	lightsalmon	lightslateblue
grey100	orange	seashell	slategray	lightyellow	lightslategray
red	orchid	aliceblue	slategrey	navajowhite	lightslategrey
tan	purple	burlywood	steelblue	saddlebrown	lightsteelblue
tan1	salmon	cadetblue	turquoise	springgreen	mediumseagreen
tan3	sienna	chocolate	violetred	yellowgreen	mediumslateblue
tan4	tomato	darkgreen	aquamarine	antiquewhite	mediumturquoise
blue	violet	darkkhaki	blueviolet	darkseagreen	mediumvioletred
cyan	yellow	firebrick	chartreuse	lemonchiffon	mediumaquamarine
gold	darkred	gainsboro	darkorange	lightskyblue	mediumspringgreen
navy	dimgray	goldenrod	darkorchid	mediumorchid	lightgoldenrodyellow



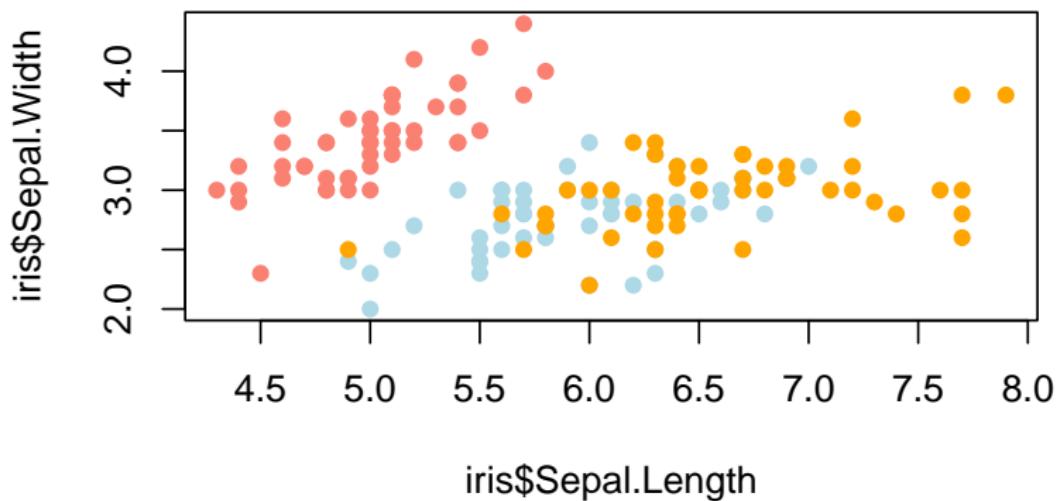
## Vector of colors

```
plot(iris$Sepal.Length, iris$Sepal.Width, pch=16,  
      col=iris$Species) # Species is a factor  
# Quick colors 1:3 from palette()
```



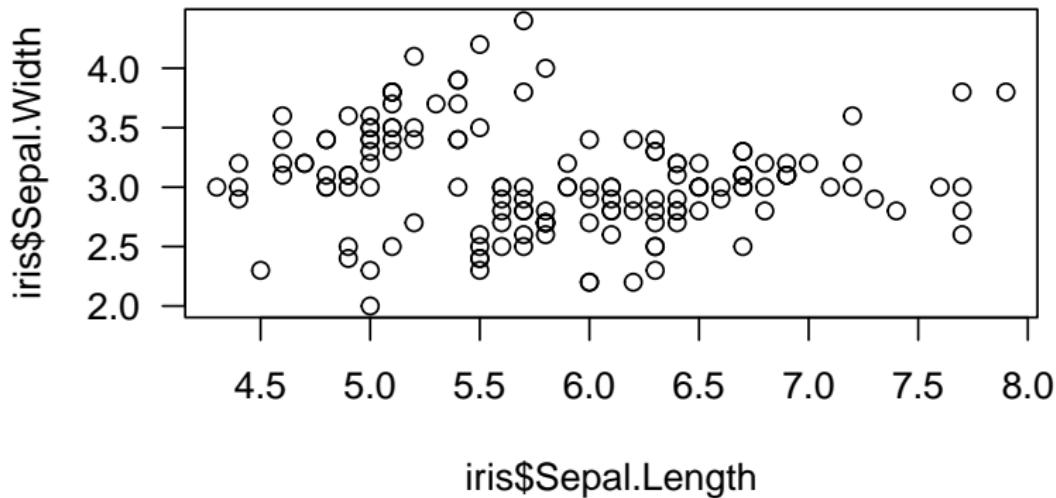
## Vector of custom colors

```
colors <- c("salmon", "lightblue", "orange")
plot(iris$Sepal.Length, iris$Sepal.Width, pch=16,
      col=colors[iris$Species])
```



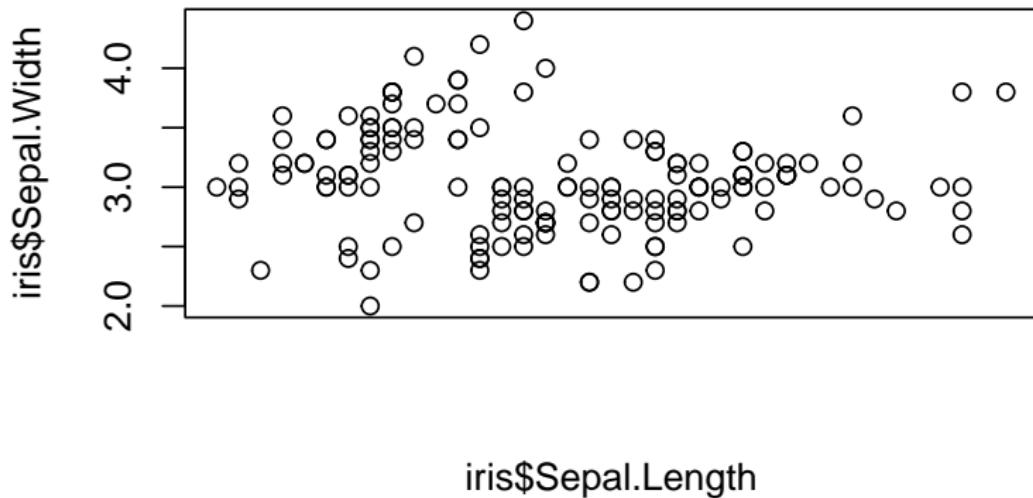
## Orientation of axis labeling

```
plot(iris$Sepal.Length, iris$Sepal.Width,  
      las=1)  
# las: LabelAxisStyle (numbers upright)
```



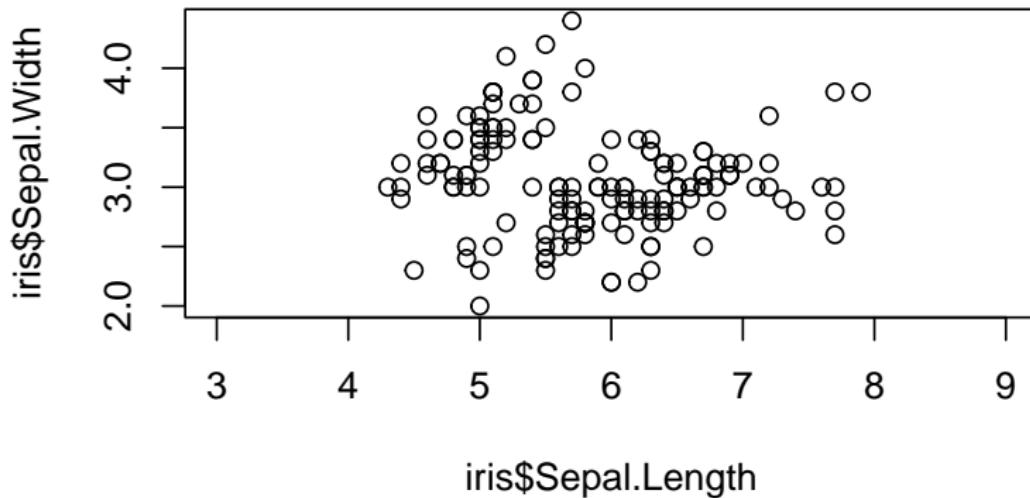
## Suppress axis labeling

```
plot(iris$Sepal.Length, iris$Sepal.Width,  
      xaxt="n")  
# xaxt: X Axis Type, n = none, nichts, nada, niente
```



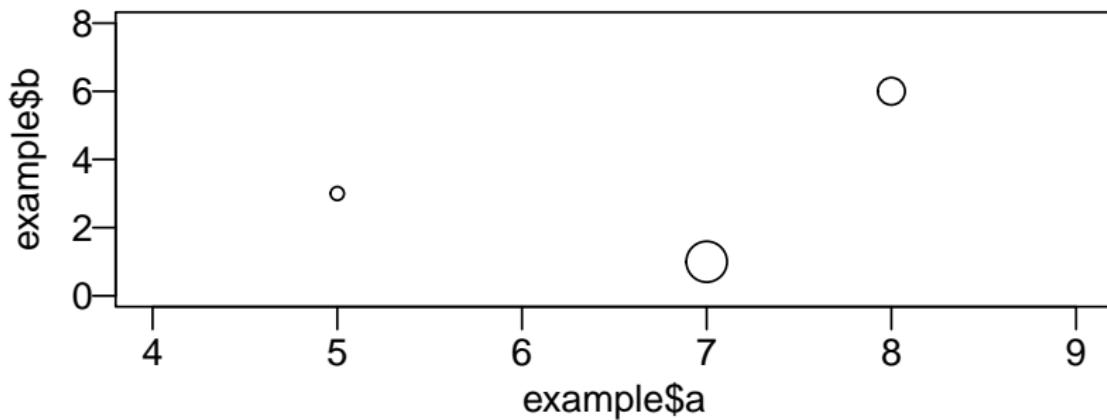
## Axis range

```
plot(iris$Sepal.Length, iris$Sepal.Width,  
      xlim=c(3,9) )  
      # xlim: X axis LIMits, analog for ylim
```



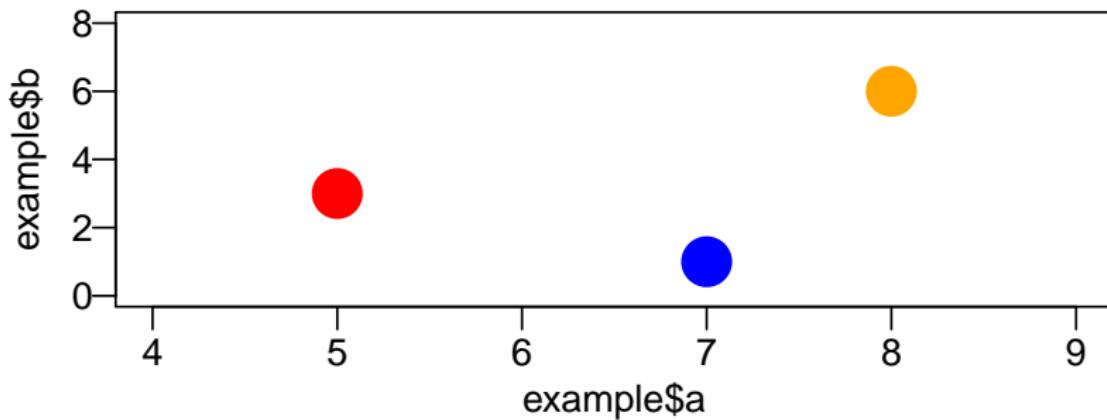
## Vectorization I

```
example <- read.table(header=TRUE, text="  
a b value index  
5 3 one 1  
8 6 two 2  
7 1 three 3 ")  
  
plot(example$a, example$b, ylim=c(0,8), xlim=c(4,9),  
      cex=example$index*0.8)  
# Vector with Character EXPansion sizes
```



## Vectorization II

```
example <- read.table(header=TRUE, text="  
a b value index  
5 3 one 1  
8 6 two 2  
7 1 three 3 ")  
  
mycolors <- c("red", "orange", "blue")  
plot(example$a, example$b, col=mycolors[example)index],  
     ylim=c(0,8), xlim=c(4,9), cex=3, pch=16)
```



## Summary for 5.1 Scatterplots

### common `plot()` arguments:

- ▶ `x`, `y`: point coordinates
- ▶ `xlab`, `ylab`: axis labels
- ▶ `main`: title
- ▶ `xlim=c(20,80)`, `ylim=1:0`: graph limits (can be reversed)
- ▶ `col`: point color(s)
- ▶ `pch`: point character (symbol)
- ▶ `cex`: character expansion (symbol size)
- ▶ `las`: label axis style (`1` for upright axis numbers)
- ▶ `xaxt`: axis type (`"n"` to suppress axis)

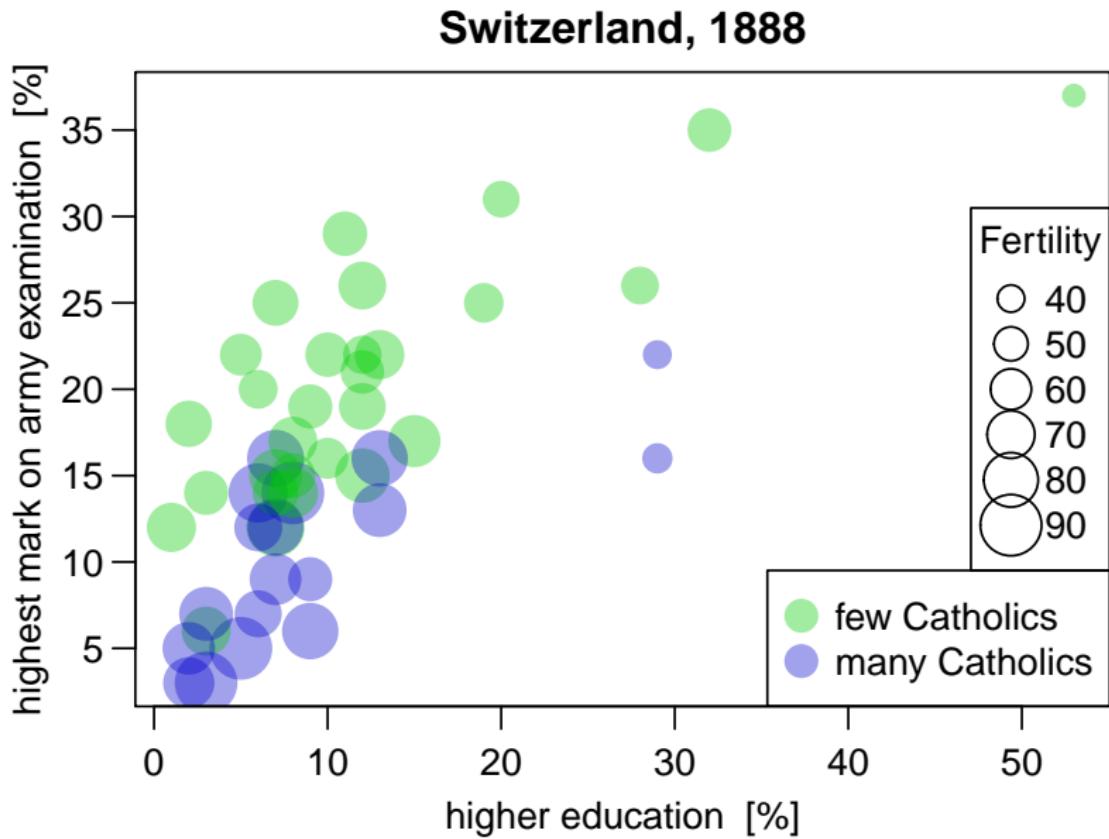
### Paul Murrell mnemonics

Report unclear tasks in the forum

Highlight the topics from this lesson in your RefCard

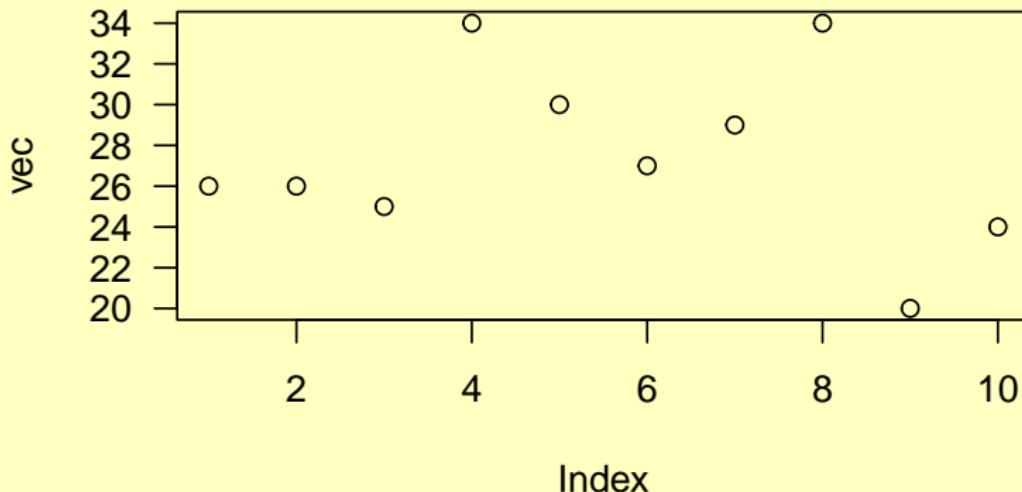


## Exercise solution



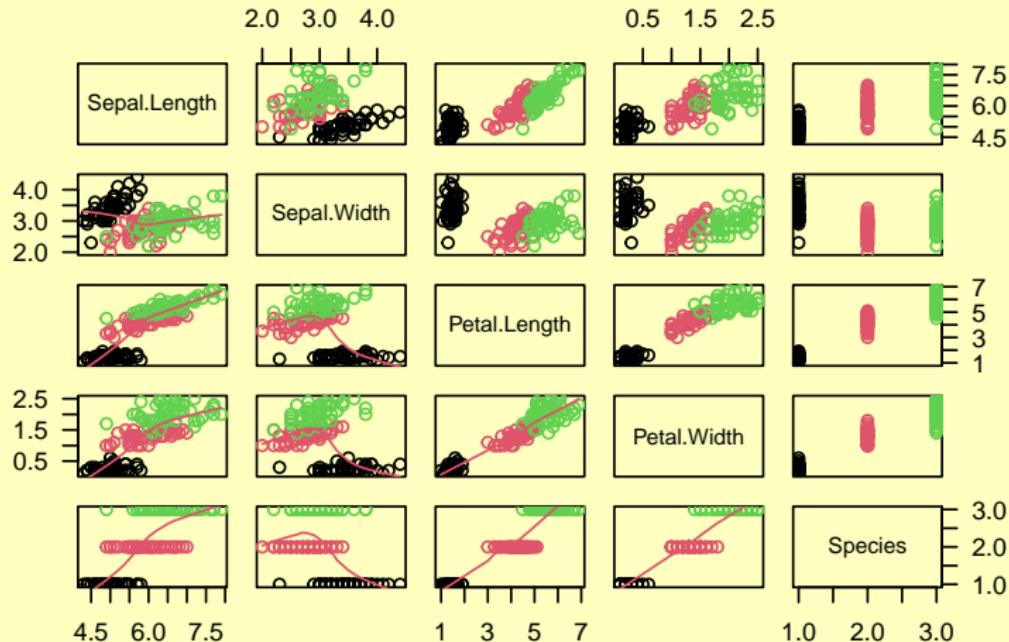
## Vector input

```
vec <- c(26,26,25,34,30,27,29,34,20,24)  
plot(vec, las=1) # Index 1:n on x-axis
```



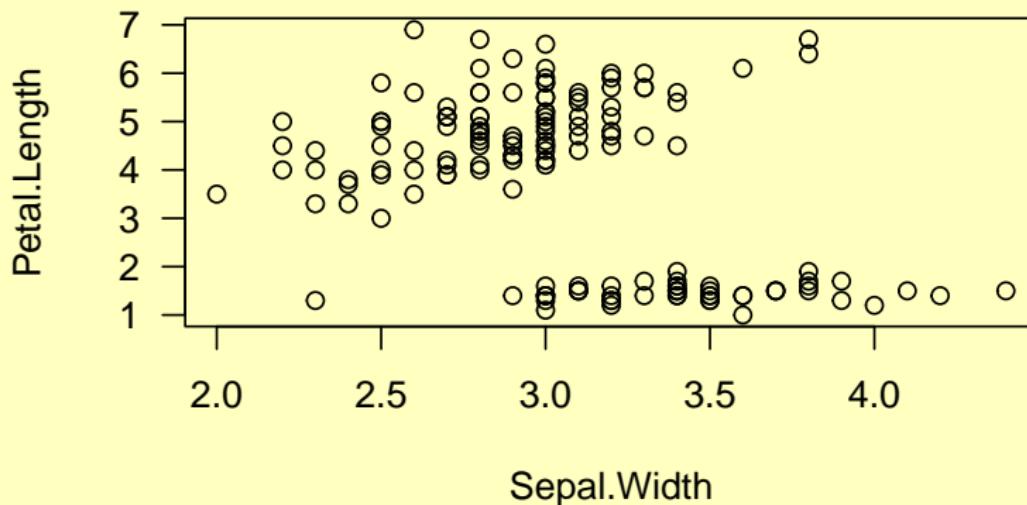
## data.frame input -> pairs -plot

```
plot(iris, col=iris$Species, lower.panel=panel.smooth)
```



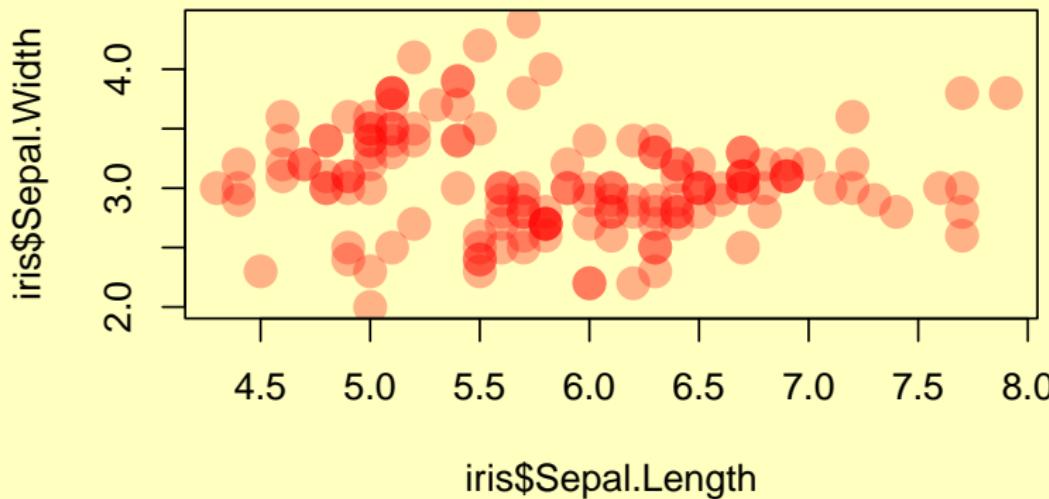
## data.frame input

```
plot(iris[,2:3],  
     las=1)  
# automatic axis names from column names
```



## Transparent colors

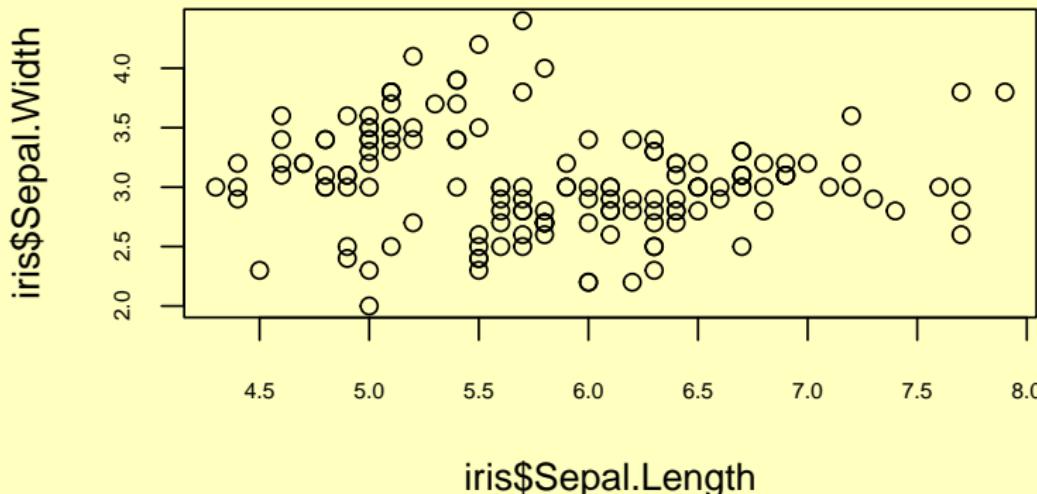
```
plot(iris$Sepal.Length, iris$Sepal.Width, cex=2,  
      col=berryFunctions::addAlpha("red"), pch=16)  
# addAlpha("red", alpha=0.3) creates "#FF00004D"
```



## Size/color/font of various elements

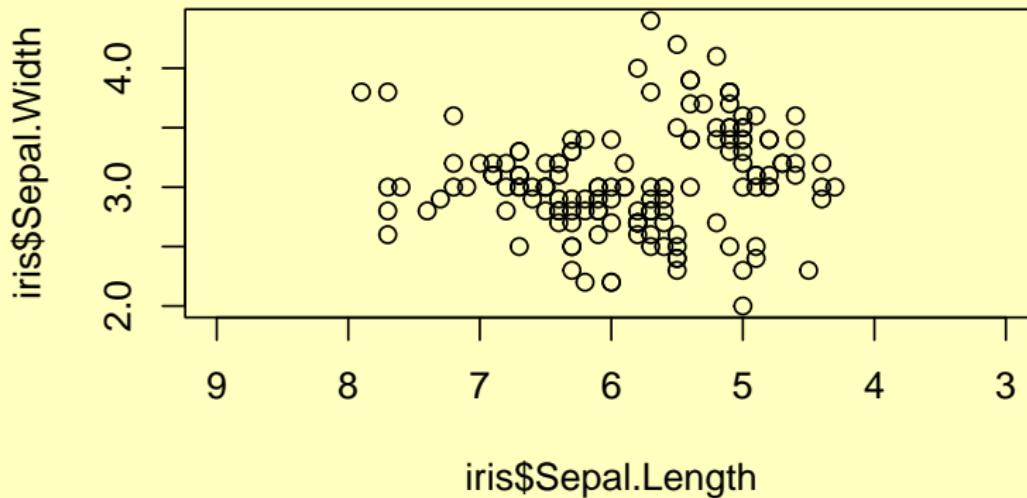
```
plot(iris$Sepal.Length, iris$Sepal.Width,  
      main="Two-line\nTitle", # font=3: italics  
      col.main="forestgreen", cex.axis=0.6, font.main=3)
```

*Two-line  
Title*



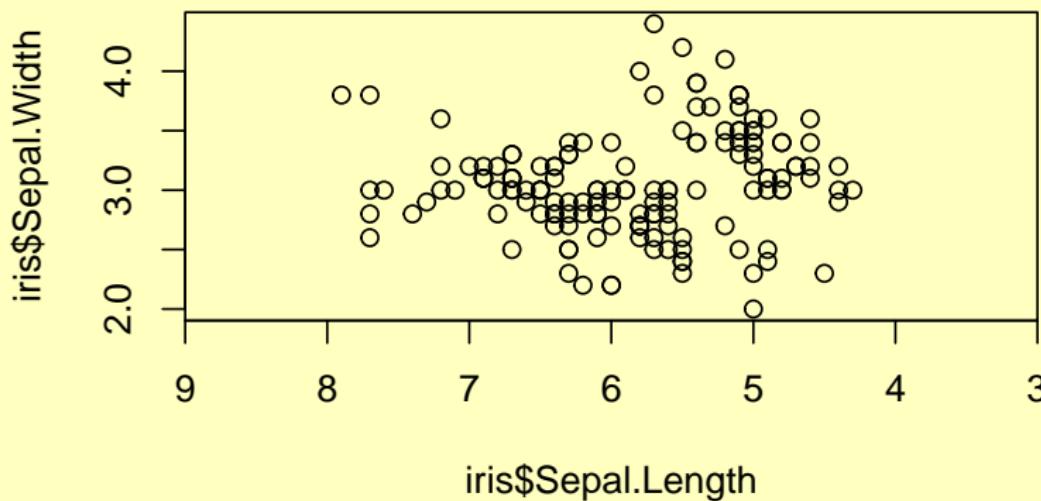
## Inverted axis

```
plot(iris$Sepal.Length, iris$Sepal.Width,  
      xlim=c(9,3) )  
# can be confusing!
```



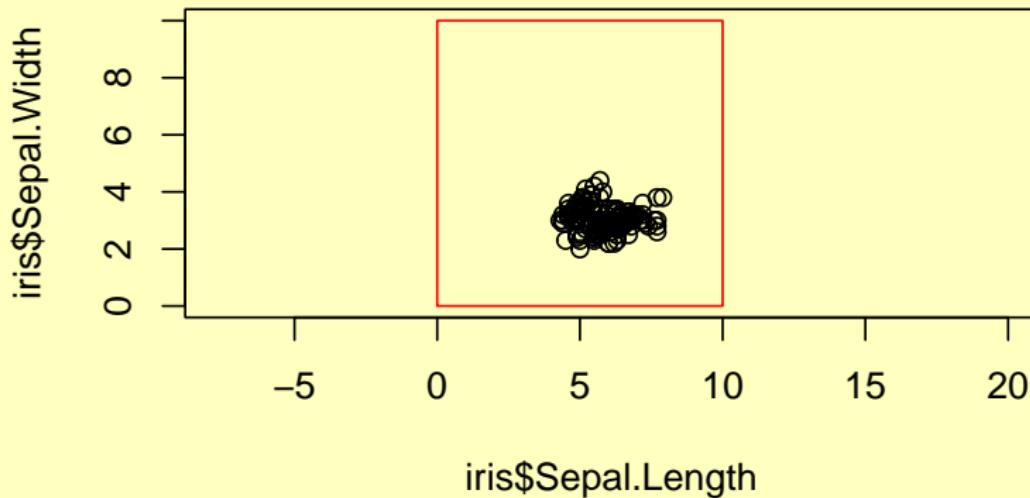
## Exact axis range

```
plot(iris$Sepal.Length, iris$Sepal.Width,  
      xlim=c(9,3), xaxs="i") # i=internal / r=regular  
# xaxis: X Axis Style. ("r" extends range by 4%)
```



## Aspect ratio: x to y proportion

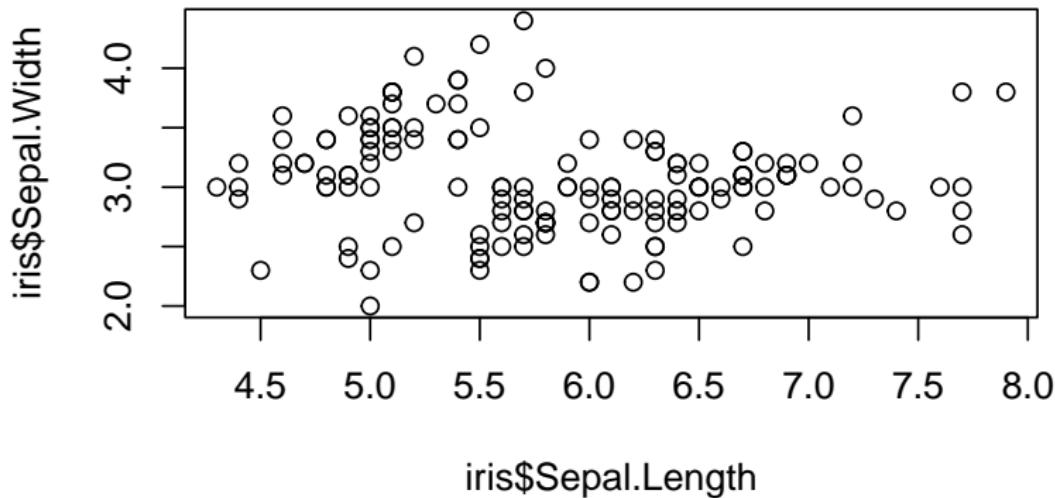
```
plot(iris$Sepal.Length, iris$Sepal.Width,  
      ylim=c(0,10), asp=1)  
# xlim automatic (Spacing per unit as for y)
```



- 1. Intro
- 2. Basics
- 3. Objects
- 4. Data
- 5. Graphics
  - 5.1 Scatterplots
  - 5.2 Line plots
  - 5.3 Barplots
  - 5.4 Low level plotting
  - 5.5 Composition
  - 5.6 Distribution plots
  - 5.7 Export
  - 5.8 Outlook
- 6. Flow control

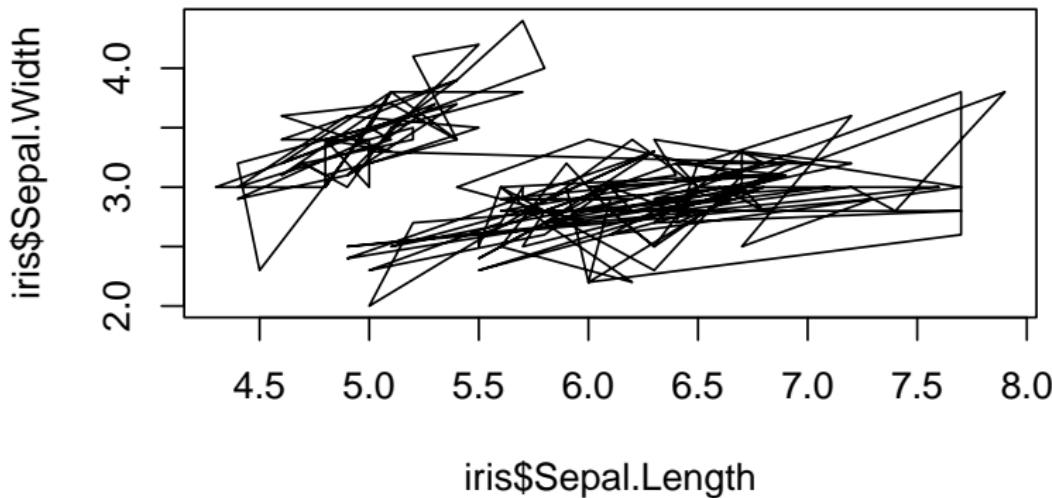
So far, we've drawn scatterplots

```
plot(iris$Sepal.Length, iris$Sepal.Width,  
      type="p")  
# type "p" (points) is the default
```



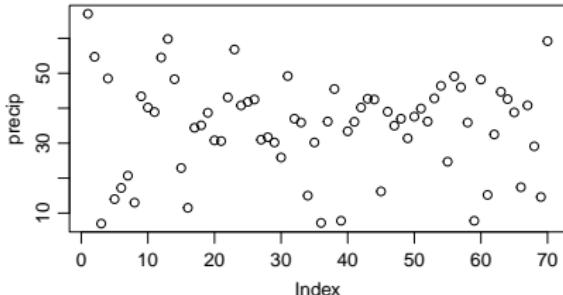
Line chart, e.g. for time series

```
plot(iris$Sepal.Length, iris$Sepal.Width,  
      type="l")  
# type: l for Lines
```



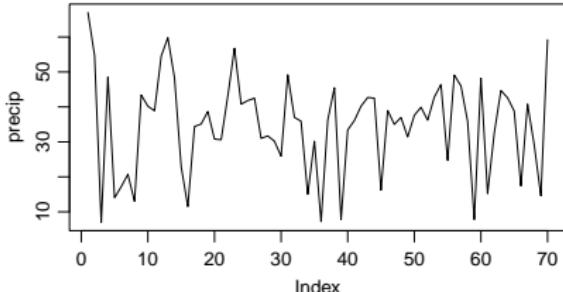
## Lines and points

```
# default: Points  
plot(precip, type="p")
```



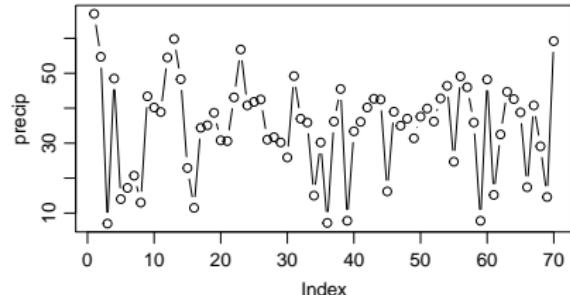
# Lines

```
plot(precip, type="l")
```



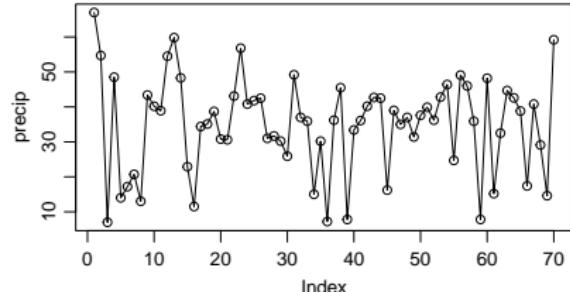
# Both

```
plot(precip, type="b")
```



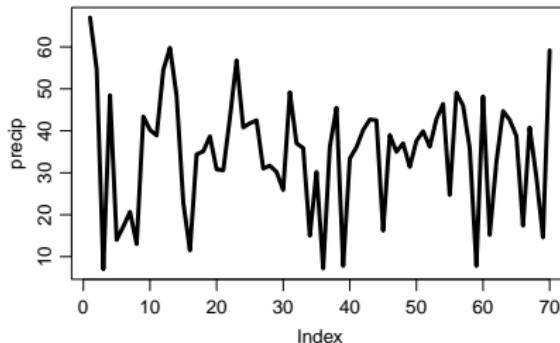
# both Overplotted

```
plot(precip, type="o")
```

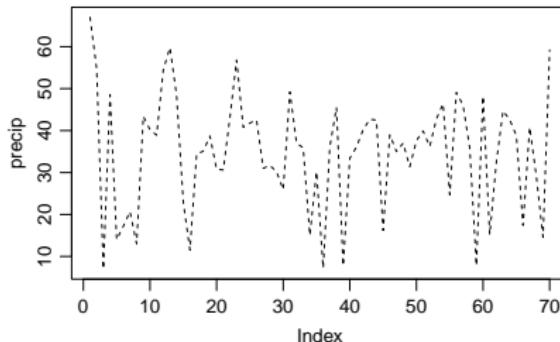


## Line types

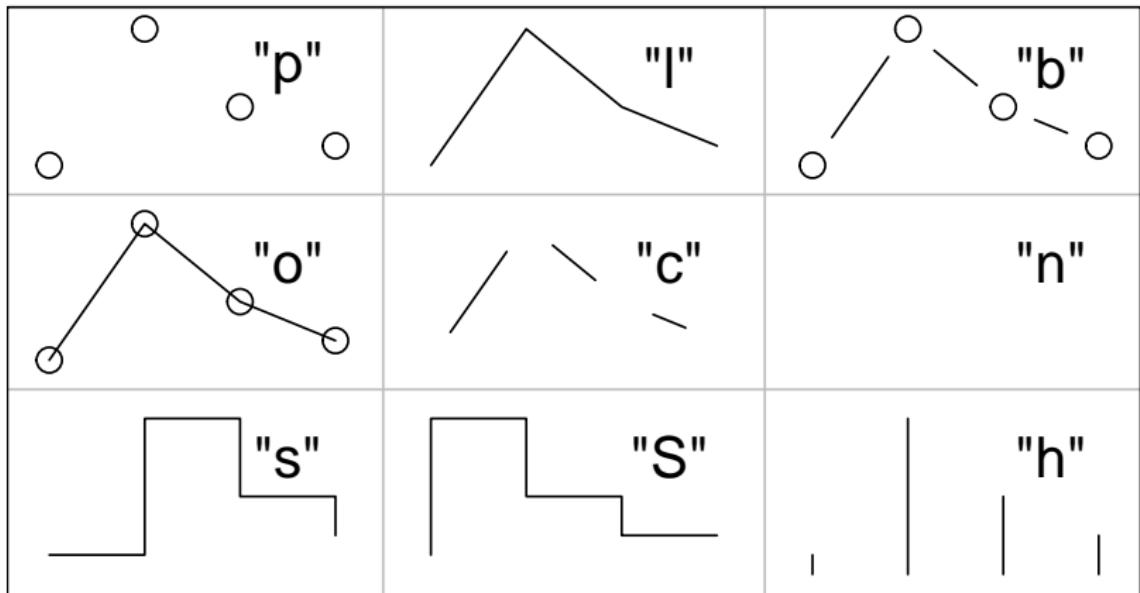
```
plot(precip, type="l", lwd=3.5) # Line WiDth
```



```
plot(precip, type="l", lty=2) # Line TYpe
```



## plot ( x, y, type = \_ )



### **plot ( x, y, lty = \_ )**

"blank" (no line) 0

"solid" (default) 1 

"dashed" 2 

"dotted" 3 

"dotdash" 4 

"longdash" 5 

"twodash" 6 

## Summary for 5.2 Line plots

### `plot` arguments for line graphs:

- ▶ `type="l"` : draw lines instead of points
- ▶ `lwd` : line width (thickness of lines)
- ▶ `lty` : line type (solid, dashed, dotted)
- ▶ overview of `type` + `lty` options

Report unclear tasks in the forum

Highlight the topics from this lesson in your RefCard



- 1. Intro
- 2. Basics
- 3. Objects
- 4. Data
- 5. **Graphics**
  - 5.1 Scatterplots
  - 5.2 Line plots
  - 5.3 Barplots**
  - 5.4 Low level plotting
  - 5.5 Composition
  - 5.6 Distribution plots
  - 5.7 Export
  - 5.8 Outlook
- 6. Flow control

## Two built-in data sets

```
str(longley)
## 'data.frame': 16 obs. of 7 variables:
## $ GNP.deflator: num 83 88.5 88.2 89.5 96.2 98.1 99 100 ...
## $ GNP          : num 234 259 258 285 ...
## $ Unemployed   : num 236 232 368 335 ...
## $ Armed.Forces: num 159 146 162 165 ...
## $ Population   : num 108 109 110 111 ...
## $ Year         : int 1947 1948 1949 1950 1951 1952 1953 1954 ...
## $ Employed     : num 60.3 61.1 60.2 61.2 ...
```

### VADeaths

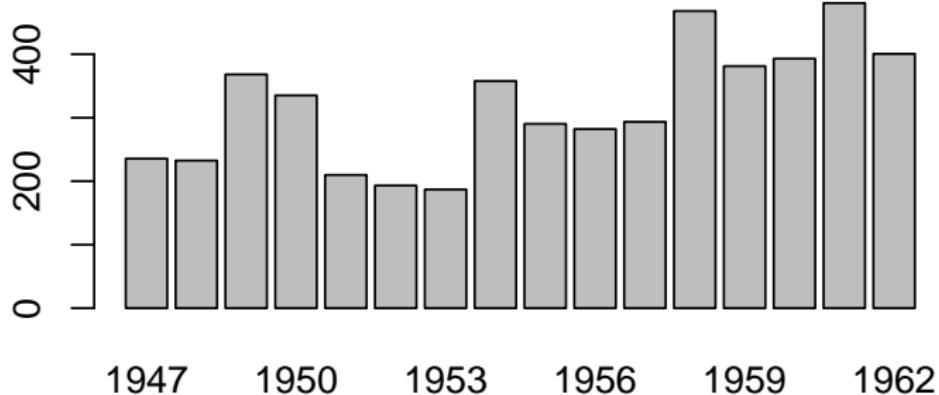
```
##      Rural Male.Rural Female.Urban Male.Urban Female.Urban
## 50-54    11.7        8.7       15.4        8.4
## 55-59    18.1       11.7       24.3       13.6
## 60-64    26.9       20.3       37.0       19.3
## 65-69    41.0       30.9       54.6       35.1
## 70-74    66.0       54.3       71.1       50.0
```

```
data("longley") # load into globalenv()
# Rstudio str + View clickable afterwards
```



## Bar charts

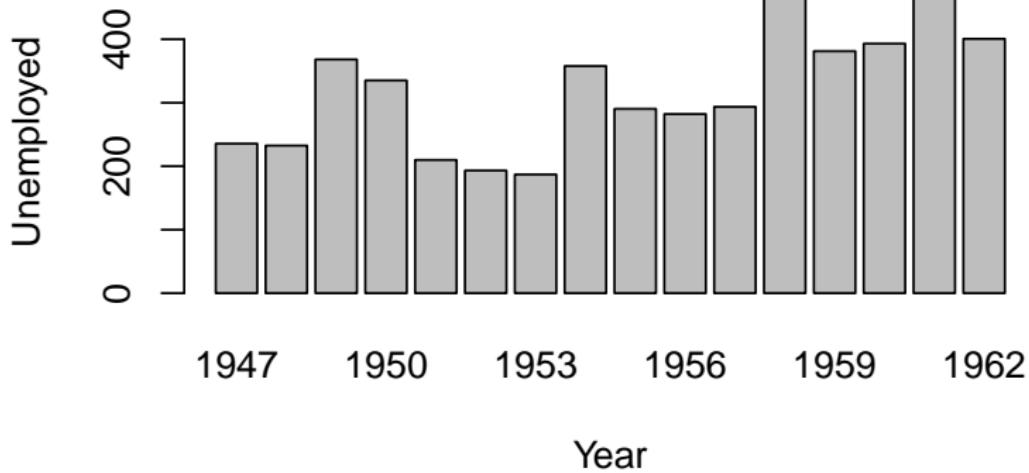
```
barplot(longley$Unemployed, names.arg=longley$Year)
```



Formula interface: less typing :). Read  $\sim$ : as 'dependent on'

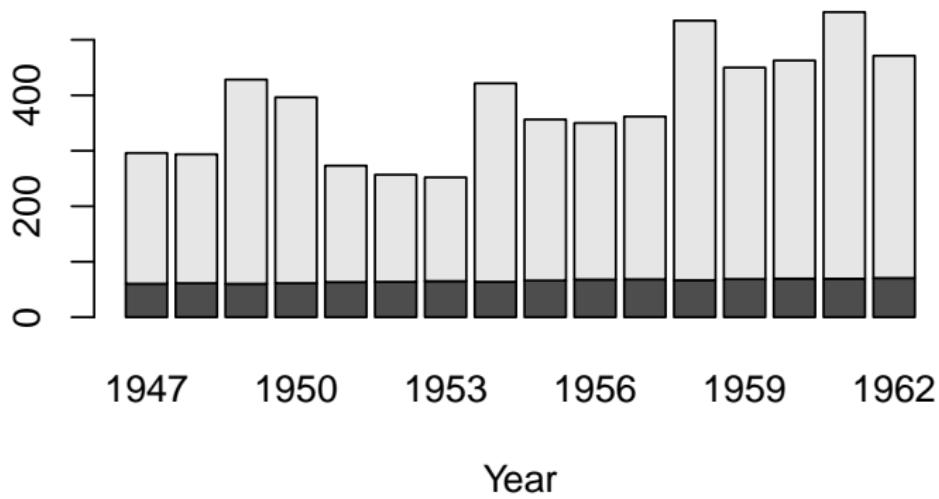
```
barplot(Unemployed ~ Year, data=longley)
```

AltGr + +, Option + N (+ Space)



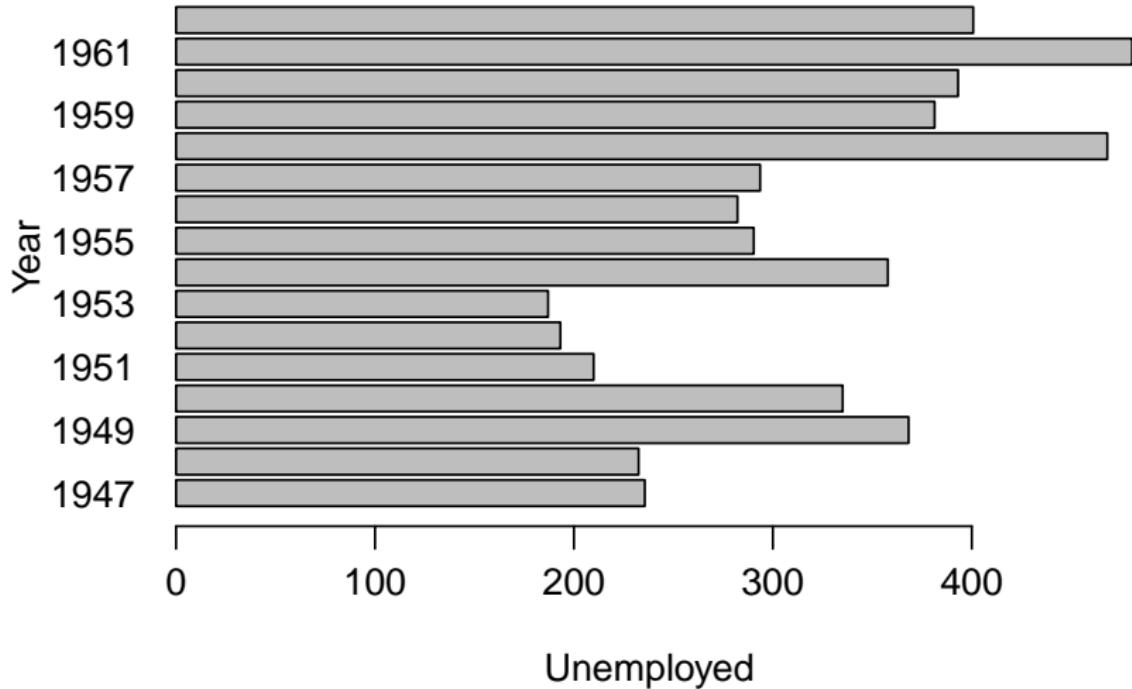
## Formula interface: flexibly expandable

```
barplot(cbind(Employed, Unemployed) ~ Year, data=longley)
```



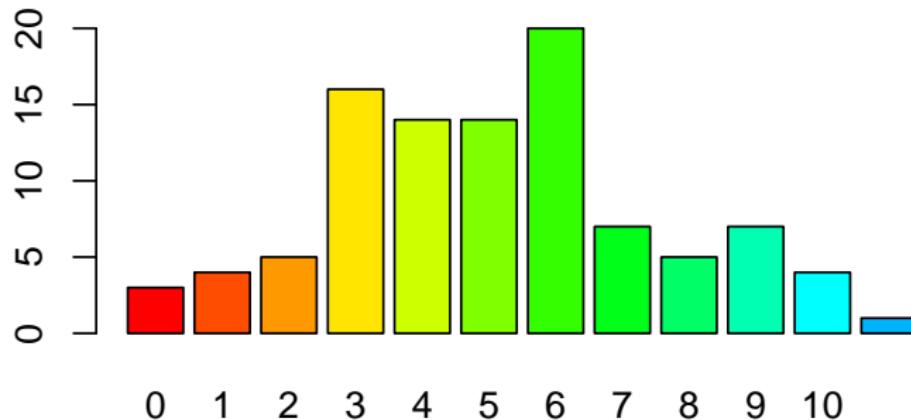
## Horizontal bar charts

```
barplot(Unemployed~Year, data=longley, horiz=TRUE, las=1)
```



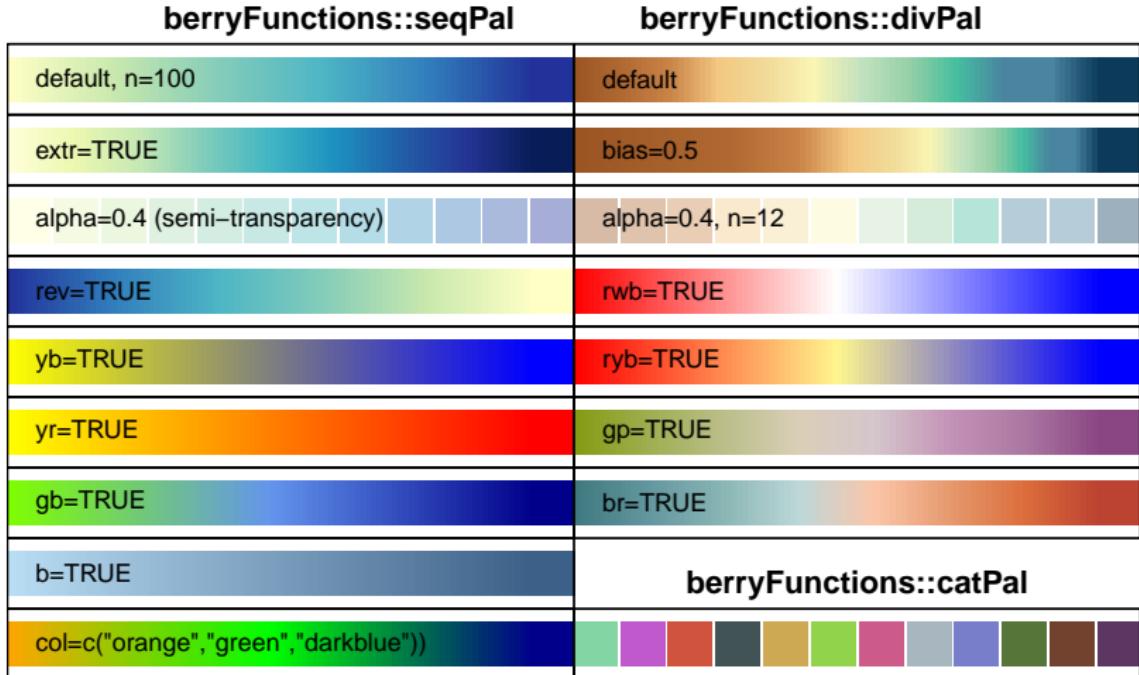
## Built-in color palettes

```
numbers <- table(stats::rpois(100, lambda=5))  
bp <- barplot(numbers, col=rainbow(20))
```



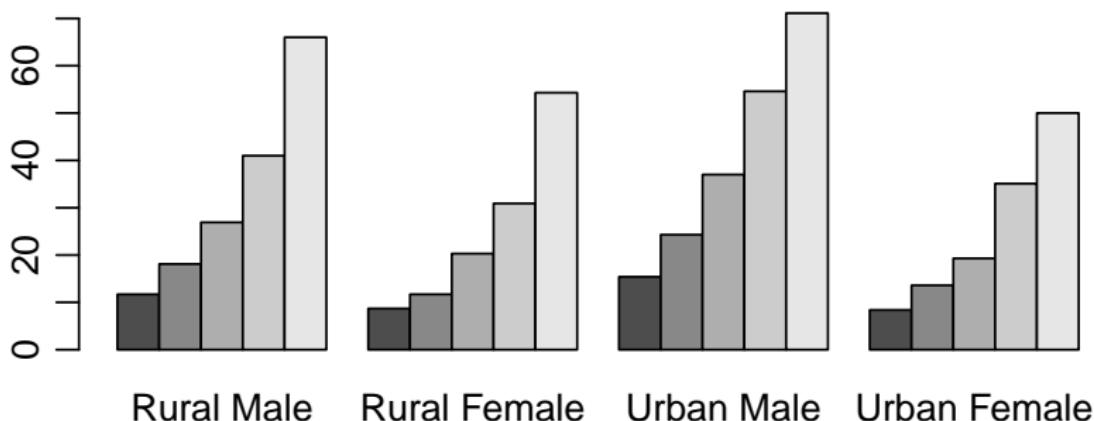
## Better color palettes

`berryFunctions::showPal()`



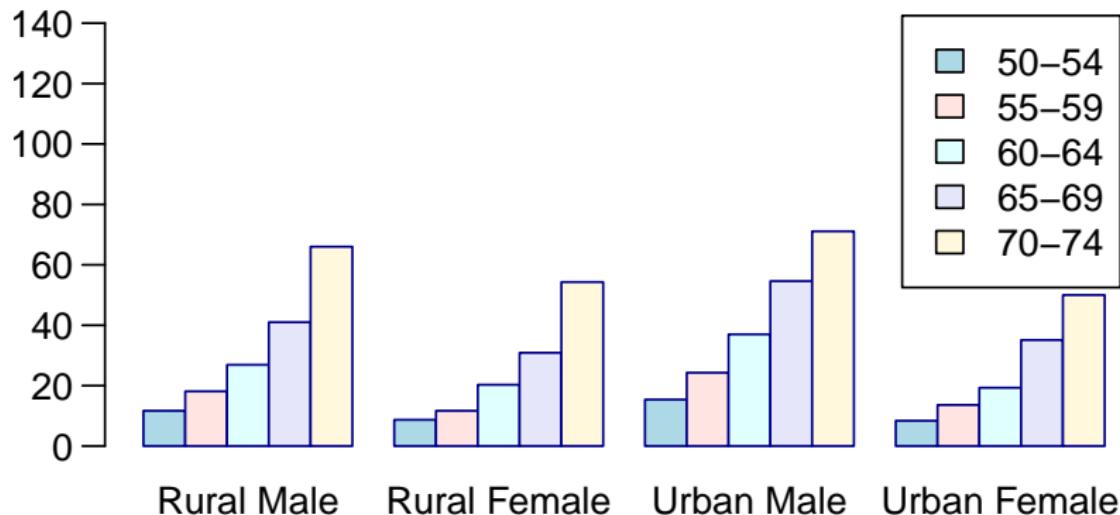
## Bar chart: grouped instead of stacked

```
midpoints <- barplot(VADeaths, beside=TRUE)  
midpoints  
##      [,1] [,2] [,3] [,4]  
## [1,]  1.5  7.5 13.5 19.5  
## [2,]  2.5  8.5 14.5 20.5  
## [3,]  3.5  9.5 15.5 21.5  
## [4,]  4.5 10.5 16.5 22.5  
## [5,]  5.5 11.5 17.5 23.5
```



## Bar chart: Legend

```
barplot(VADeaths, beside=TRUE, las=1,  
        col=c("lightblue","mistyrose","lightcyan",  
              "lavender","cornsilk"),  
        legend=TRUE, ylim=c(0, 150), border="darkblue")
```

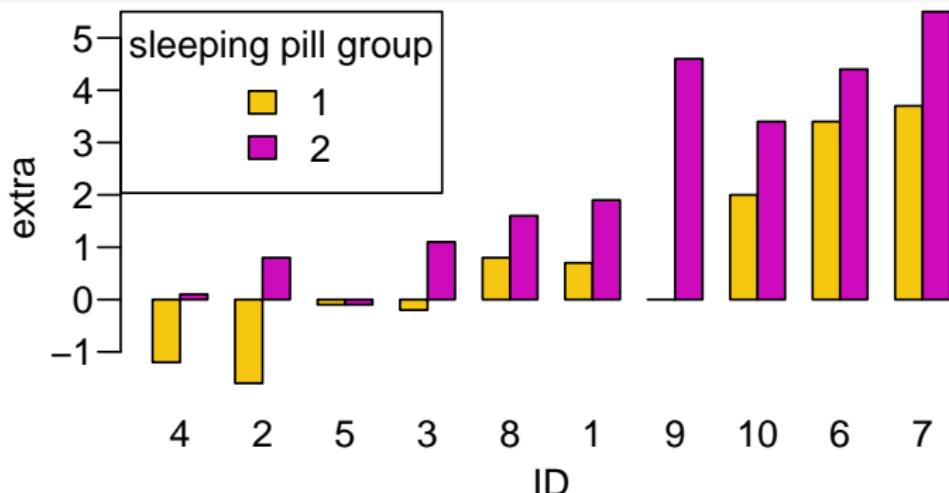


## Doubly grouped formula with +

```
head(sleep, 3)
##   extra group ID
## 1    0.7      1  1
## 2   -1.6      1  2
## 3   -0.2      1  3

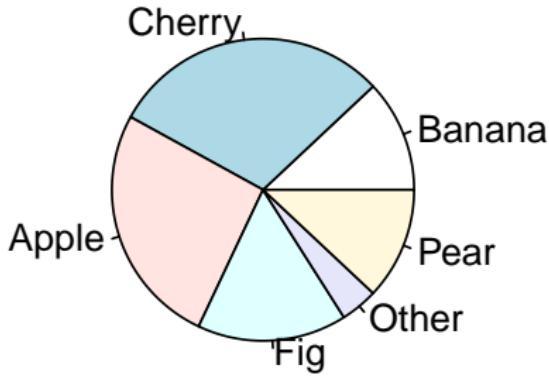
# order levels for graph: group 'extra' by 'ID', apply sum function
sleep$ID <- reorder(sleep$ID, sleep$extra, sum) # tapply(extra, ID, sum)

barplot(extra~group+ID, data=sleep, legend=TRUE, col=7:6, beside=TRUE,
        args.legend=list(title="sleeping pill group", x="topleft"))
```



## Don't use pie charts

```
vec <- c(12, 30, 26, 16, 4, 12)
names(vec) <- c("Banana", "Cherry", "Apple", "Fig", "Other", "Pear")
pie(vec)
```



If you want chart viewers to be able to compare proportions, use barplots instead. See e.g. [death to pie charts](#) or [save the pies for dessert](#).

## Summary for 5.3 Barplots

### bar charts, formulas:

- ▶ `barplot`
- ▶ Formula interface: `y~x`, `y~x1+x2`, `cbind(y1,y2)~x` (`paste`)
- ▶ Choose color palettes wisely
- ▶ Don't use pie charts

### `barplot` arguments:

- ▶ `height`
- ▶ `horiz`
- ▶ `beside`
- ▶ `legend`
- ▶ `las`, `col`, `ylim`, ...

Report unclear tasks in the forum

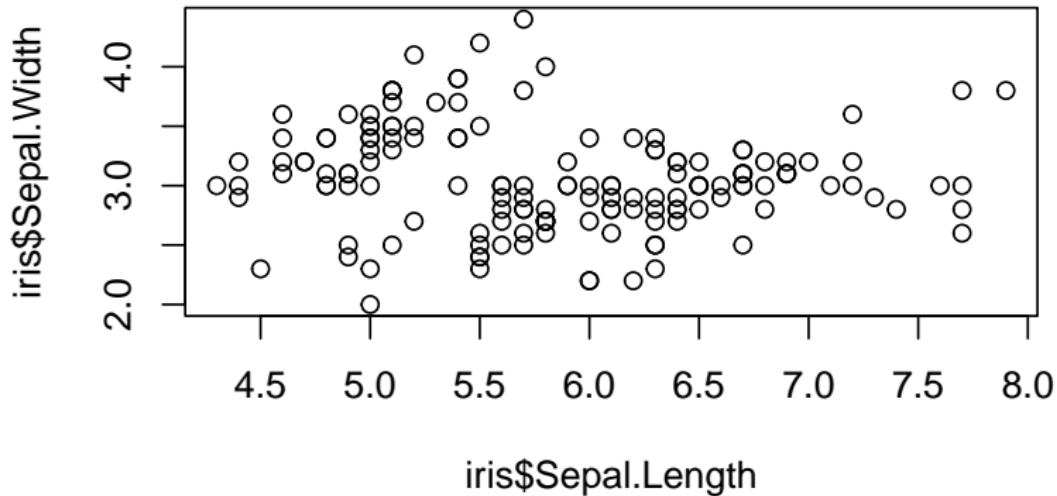
Highlight the topics from this lesson in your RefCard



- 1. Intro
  - 2. Basics
  - 3. Objects
  - 4. Data
  - 5. Graphics**
  - 6. Flow control
- 5.1 Scatterplots
  - 5.2 Line plots
  - 5.3 Barplots
  - 5.4 Low level plotting**
  - 5.5 Composition
  - 5.6 Distribution plots
  - 5.7 Export
  - 5.8 Outlook

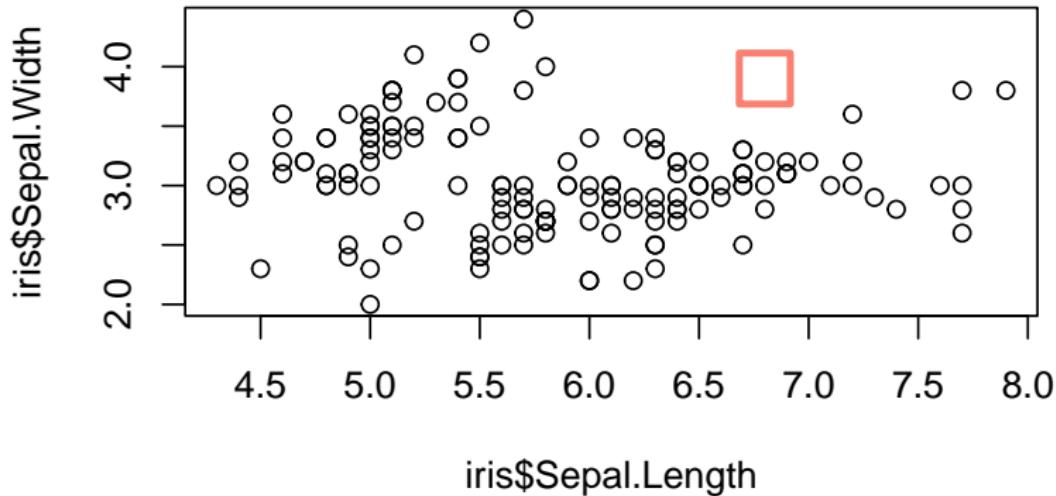
High-level commands like `plot` create a complete, new graphic

```
plot(x=iris$Sepal.Length, y=iris$Sepal.Width)
# Dataset see section 5.1 (scatterplots)
```



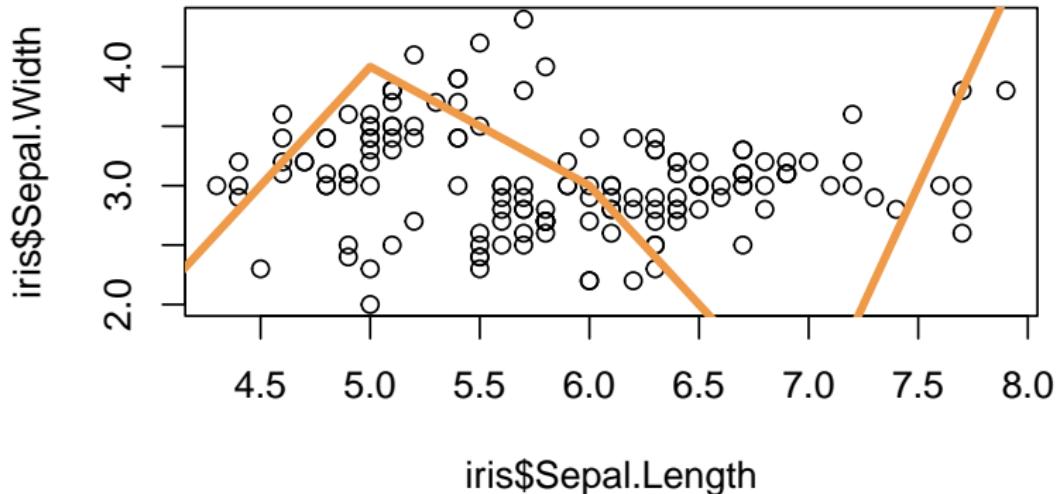
Low-level commands add something to an existing graphic

```
plot(x=iris$Sepal.Length, y=iris$Sepal.Width)
points(x=6.8, y=3.9, pch=0, cex=3, col="salmon", lwd=3)
```



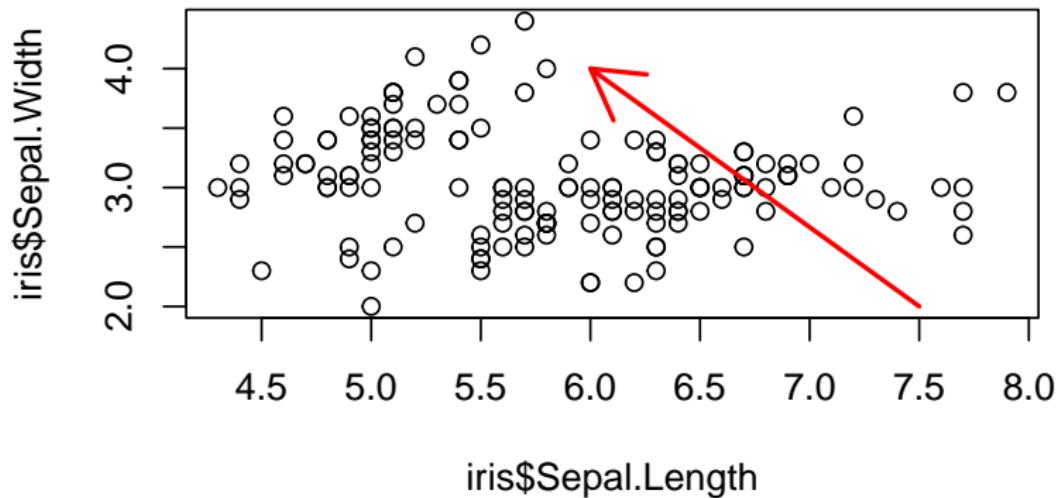
points and lines accept many arguments of plot.default

```
plot(x=iris$Sepal.Length, y=iris$Sepal.Width)
lines(x=4:8, y=c(2,4,3,1,5), lwd=3.5, col="tan2")
```



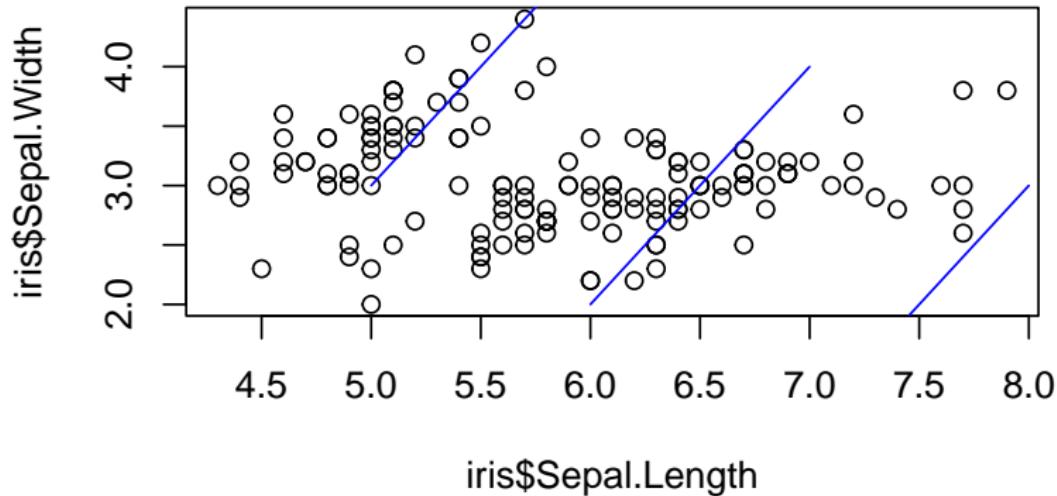
## Arrows

```
plot(x=iris$Sepal.Length, y=iris$Sepal.Width)
arrows(x0=7.5,y0=2, x1=6,y1=4, col="red", lwd=2)
```



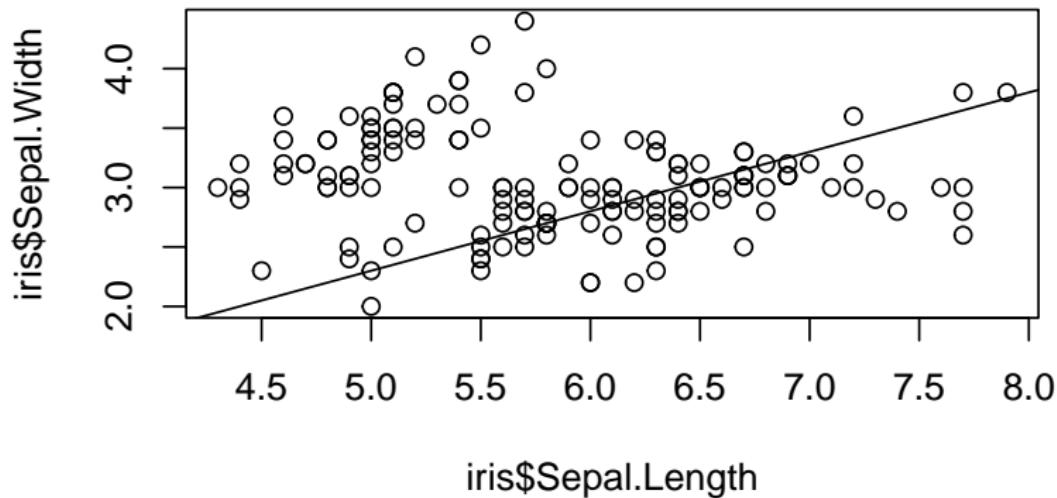
## Segments

```
plot(x=iris$Sepal.Length, y=iris$Sepal.Width)
segments(x0=5:7, y0=3:1, x1=6:8, y1=5:3, col="blue")
```



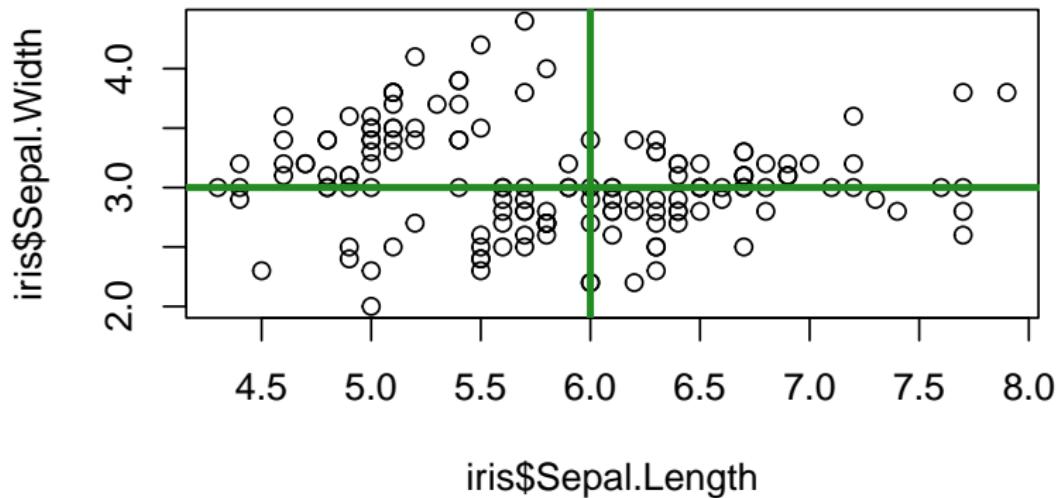
## Straight line

```
plot(x=iris$Sepal.Length, y=iris$Sepal.Width)
abline(a=-0.2, b=0.5) # y-axis intercept, slope
```



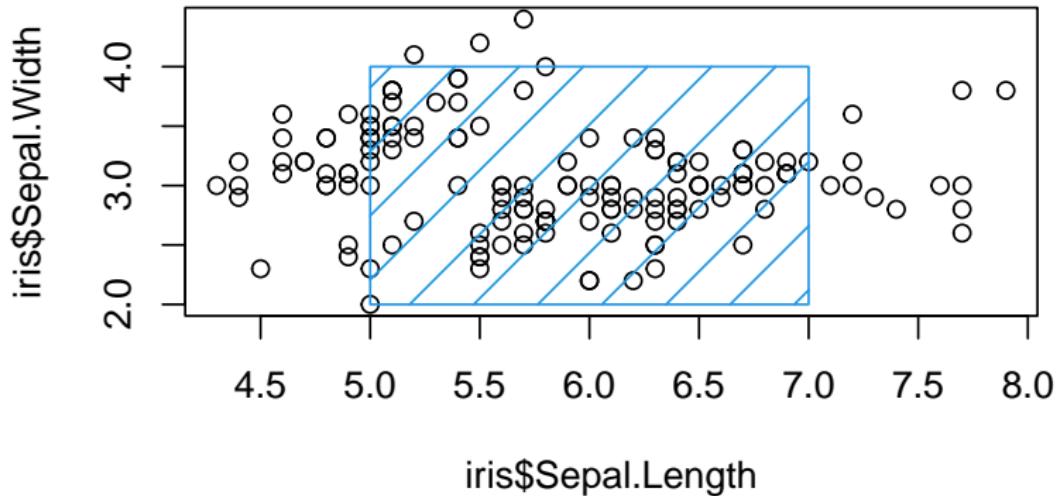
## Vertical / horizontal lines

```
plot(x=iris$Sepal.Length, y=iris$Sepal.Width)
abline(h=3, v=6, lwd=3, col="forestgreen")
```



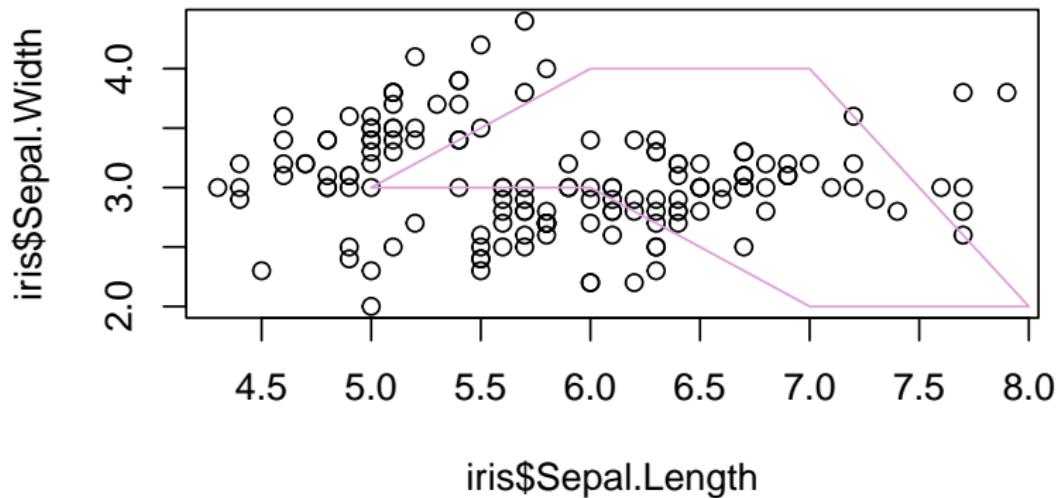
## Rectangles

```
plot(x=iris$Sepal.Length, y=iris$Sepal.Width)
rect(xleft=5,ybottom=2, xright=7,ytop=4, col=4, density=5)
```



## Polygons

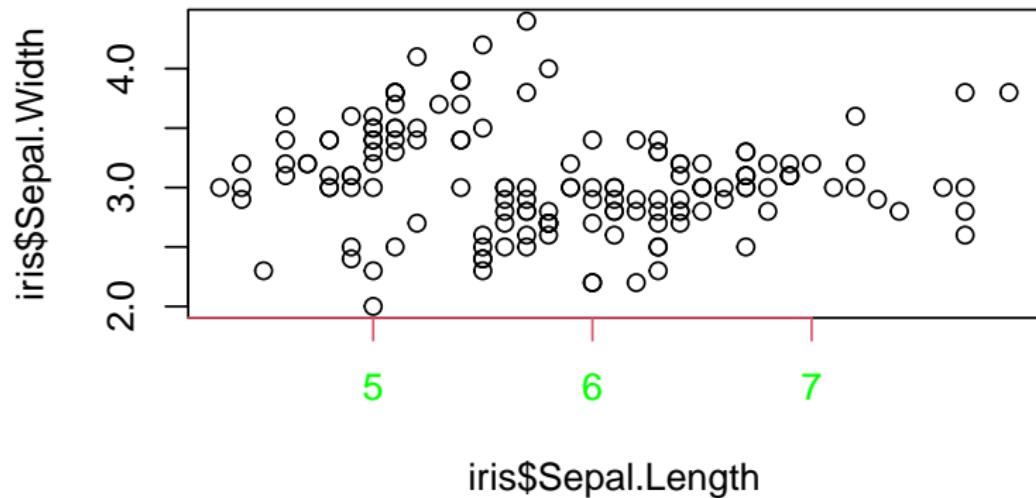
```
plot(x=iris$Sepal.Length, y=iris$Sepal.Width)
polygon(x=c(5,6,7,8,7,6), y=c(3,4,4,2,2,3), border="plum")
```



## Axes

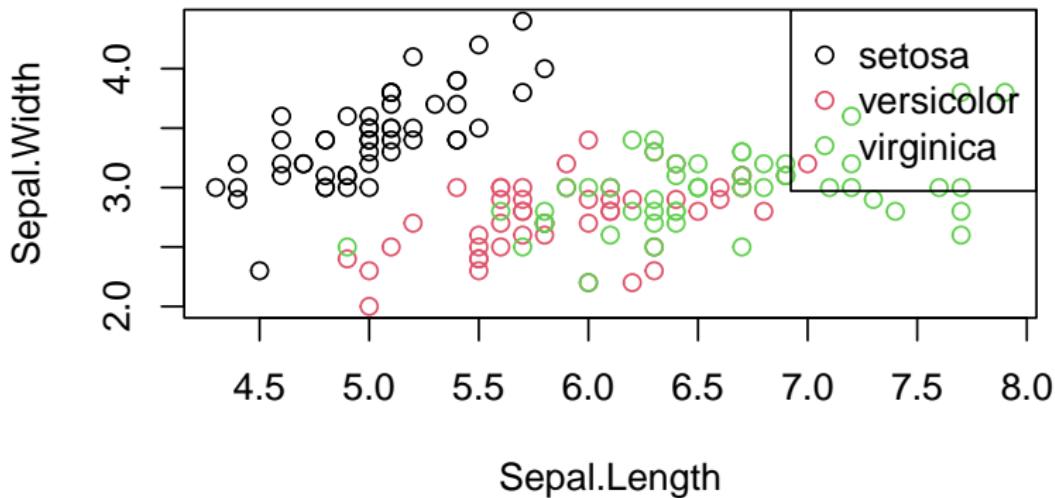
```
plot(x=iris$Sepal.Length, y=iris$Sepal.Width, xaxt="n")
axis(side=1, at=4:7, col="#DF536B", col.axis="green")
```

omit `at` for automatic number determination



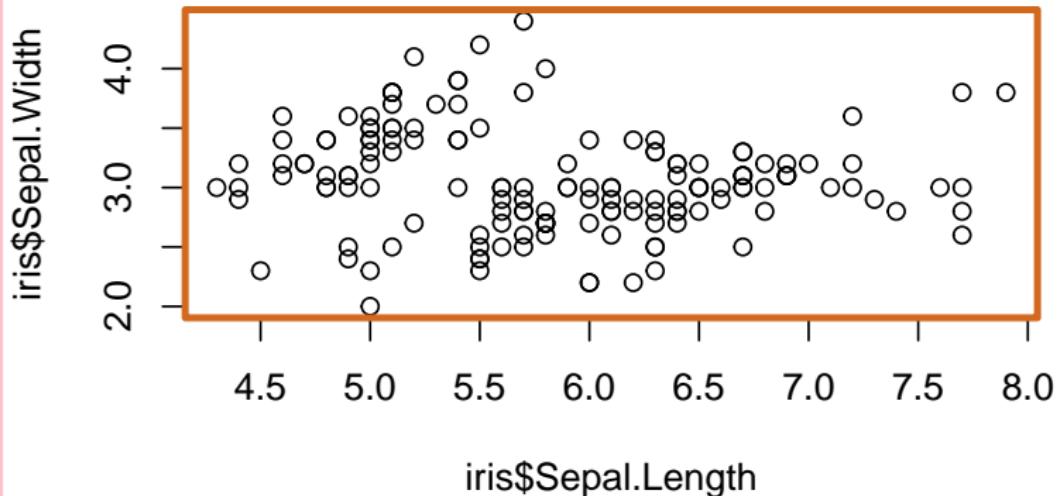
## Legend

```
plot(Sepal.Width~Sepal.Length, data=iris, col=Species)
legend("topright", levels(iris$Species), col=1:3, pch=1)
```



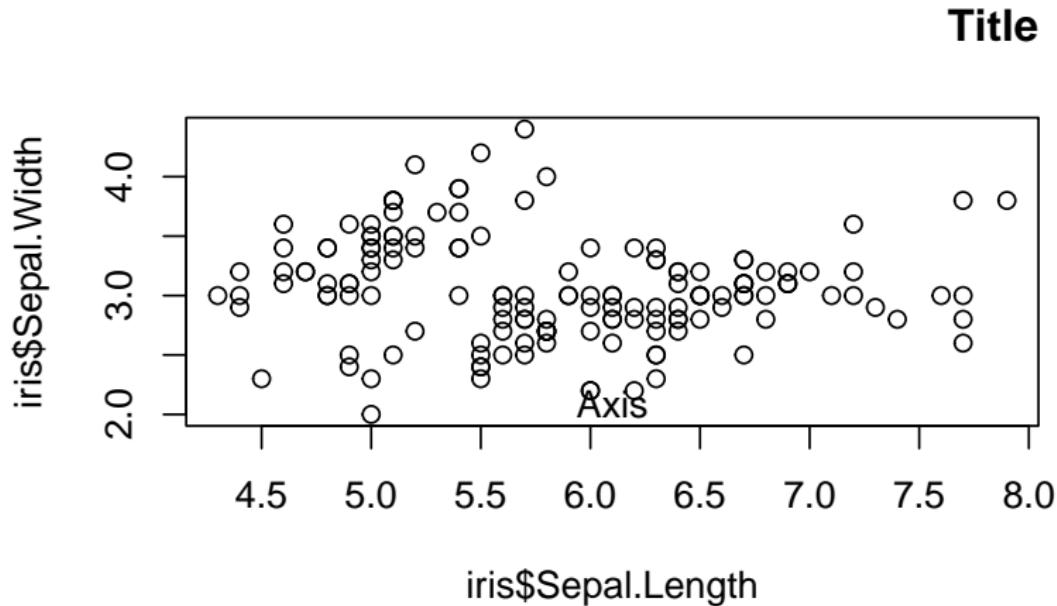
## Border

```
plot(x=iris$Sepal.Length, y=iris$Sepal.Width)
box(col="chocolate", lwd=3) ; box("outer", col="pink")
```



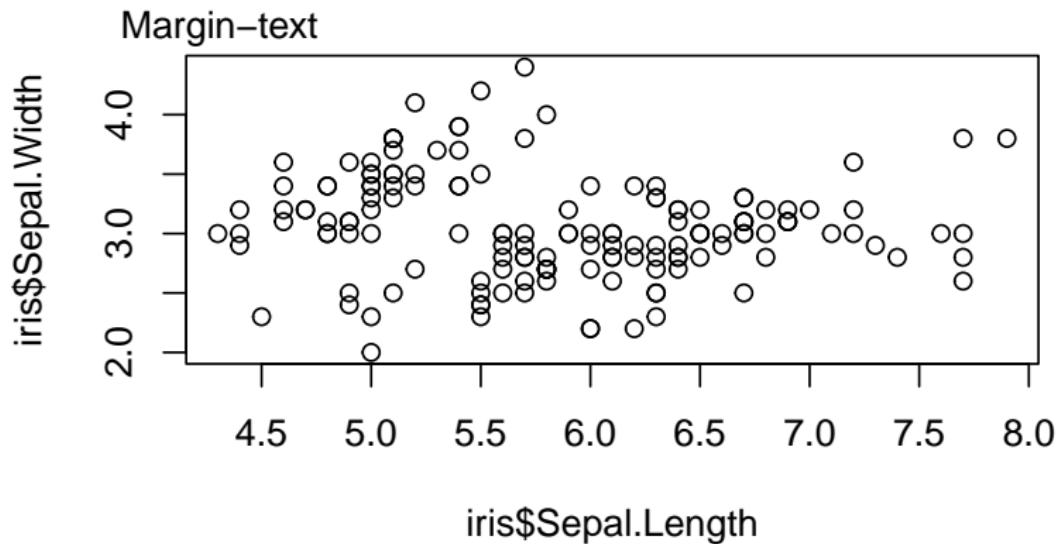
Add title elements afterwards

```
plot(x=iris$Sepal.Length, y=iris$Sepal.Width)
title(main="Title", adj=1) ; title(xlab="Axis", line=-1)
```



Text in the margin

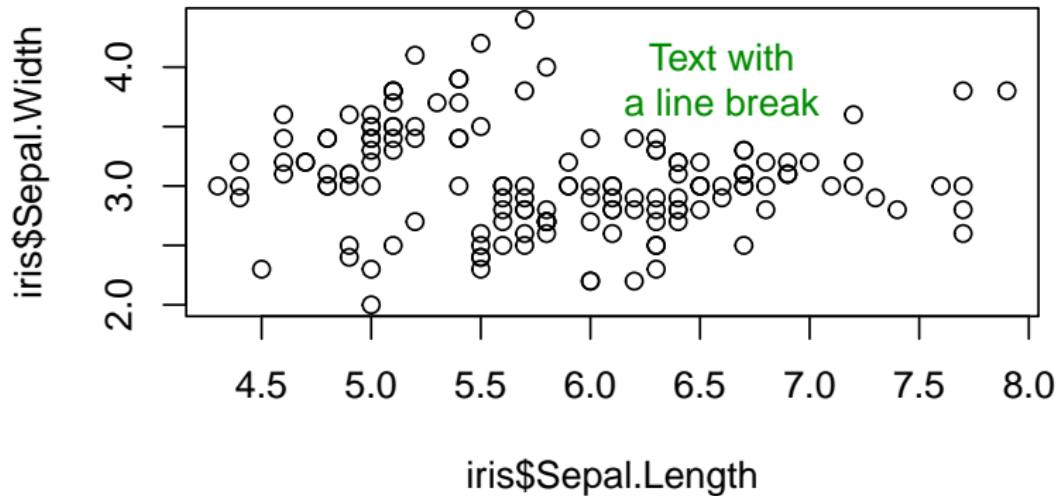
```
plot(x=iris$Sepal.Length, y=iris$Sepal.Width)
mtext(side=3, text="Margin-text", line=0.2, adj=-0.1)
```



Text in the graphic

```
plot(x=iris$Sepal.Length, y=iris$Sepal.Width)
text(6.6, 3.9, "Text with\na line break", col="green4")
```

The argument names are `x`, `y`, `labels`



## Summary for 5.4 Low level plotting

**flexible diagrams with low-level-commands:**

- ▶ `points`, `lines`
- ▶ `arrows`, `segments`, `abline`, `rect`, `polygon`
- ▶ `axis`, `legend`, `box`
- ▶ `title`, `mtext`, `text`

Nice [usage examples](#)

Report unclear tasks in the forum

Highlight the topics from this lesson in your RefCard



```
legend("topright", levels(iris$Species), col=1:3, pch=1)
```

with explicitly named arguments:

```
legend(x="topright", legend=levels(iris$Species),  
       col=1:3, pch=1)
```

This is important for the exercises. For the test script, out-of-order arguments must be explicitly specified by name.

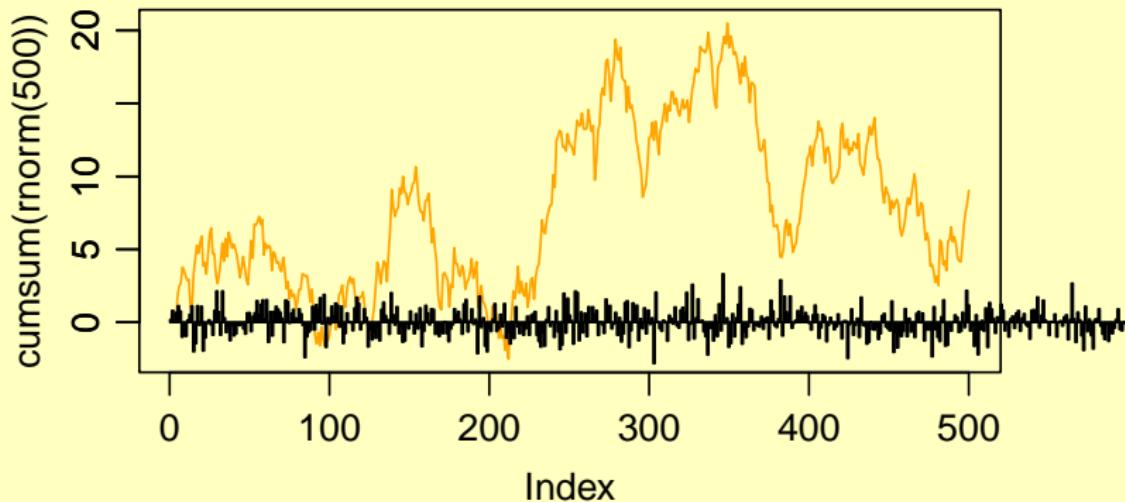
`str(legend)` shows that the second argument is `y`. If that is not specified and the second one looks like a legend entry, `legend` conveniently uses that as entries for us. The test system cannot do this.

## Another way to add to graphs

not through a low level command, but:

many high level plotting functions accept the argument `add=TRUE`:

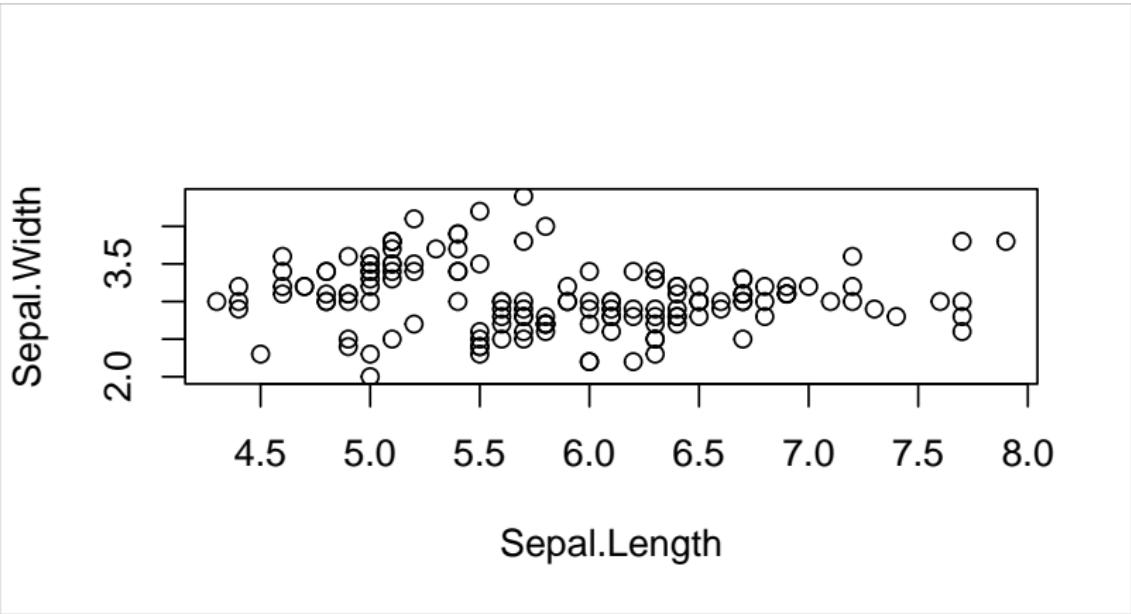
```
plot(cumsum(rnorm(500)), type="l", col="orange")
barplot(rnorm(500), add=TRUE)
```



- 1. Intro
- 2. Basics
- 3. Objects
- 4. Data
- 5. Graphics**
  - 5.1 Scatterplots
  - 5.2 Line plots
  - 5.3 Barplots
  - 5.4 Low level plotting
  - 5.5 Composition**
  - 5.6 Distribution plots
  - 5.7 Export
  - 5.8 Outlook
- 6. Flow control

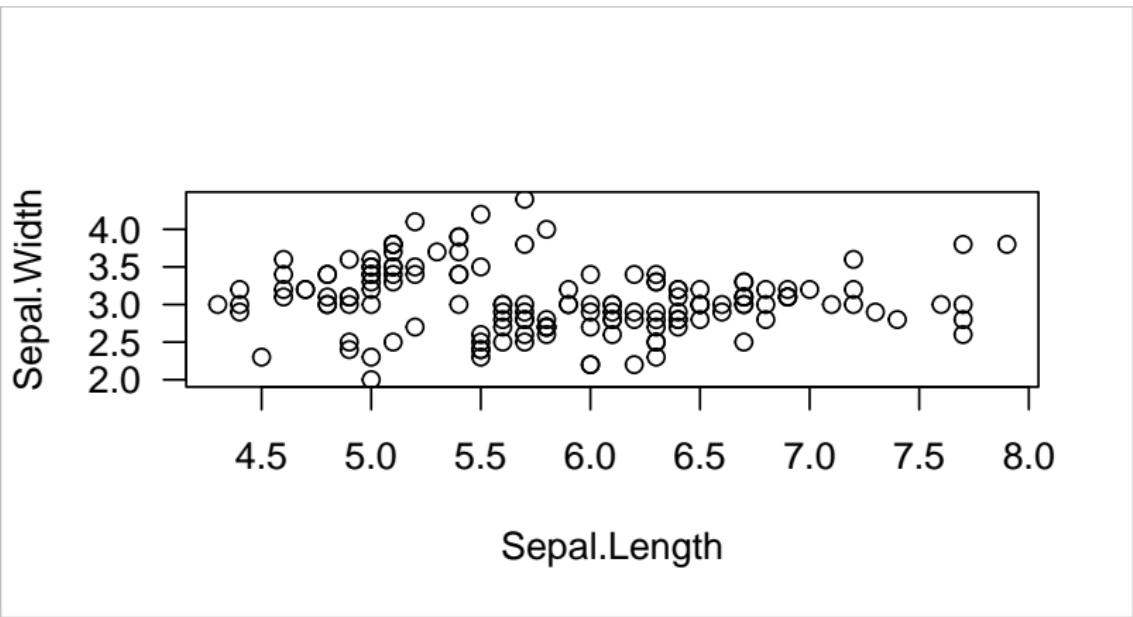
## par : default parameters for graphics

```
# default (standard) parameter:  
plot(Sepal.Width~Sepal.Length, data=iris)
```



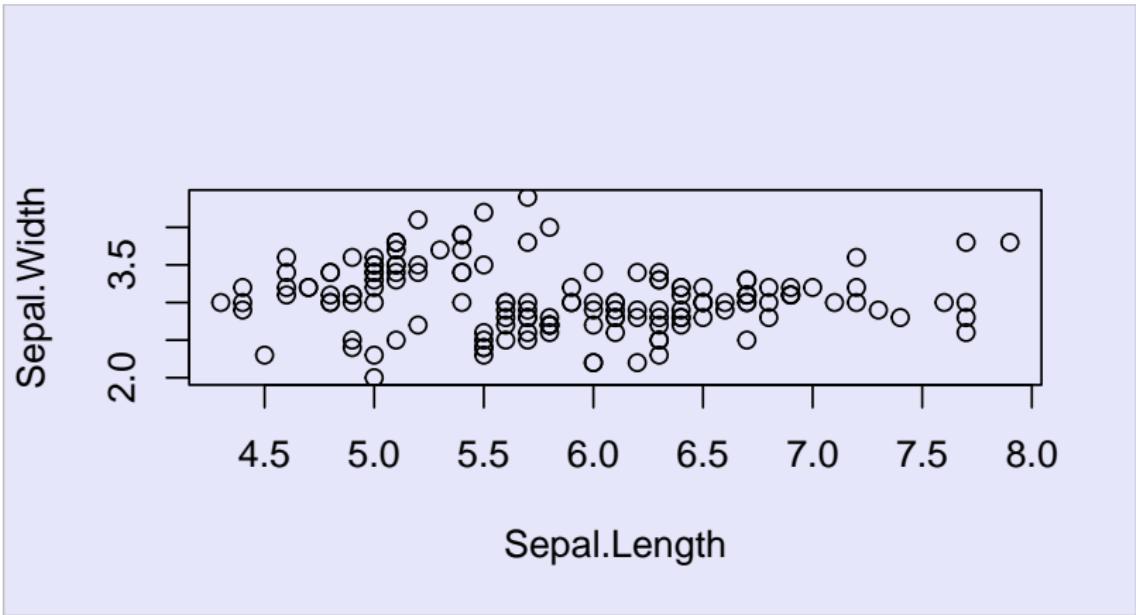
par : parameters of all following graphics

```
par(las=1) # from now on for all plots in the same device:  
plot(Sepal.Width~Sepal.Length, data=iris)
```



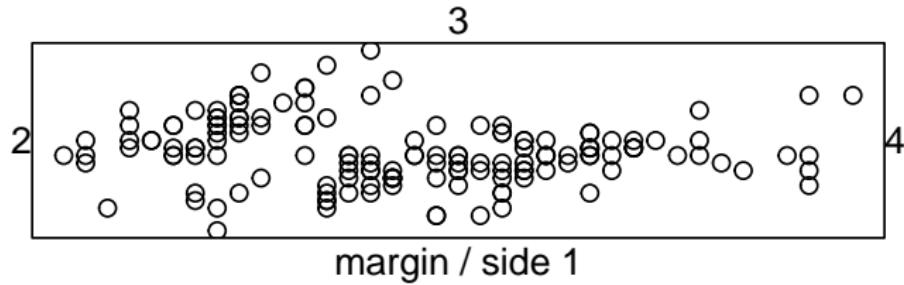
par : bg - **background** not for pch=21:25 , but for device

```
par(bg="lavender") # Device = e.g. pdf, RStudio graphics  
plot(Sepal.Width~Sepal.Length, data=iris)
```



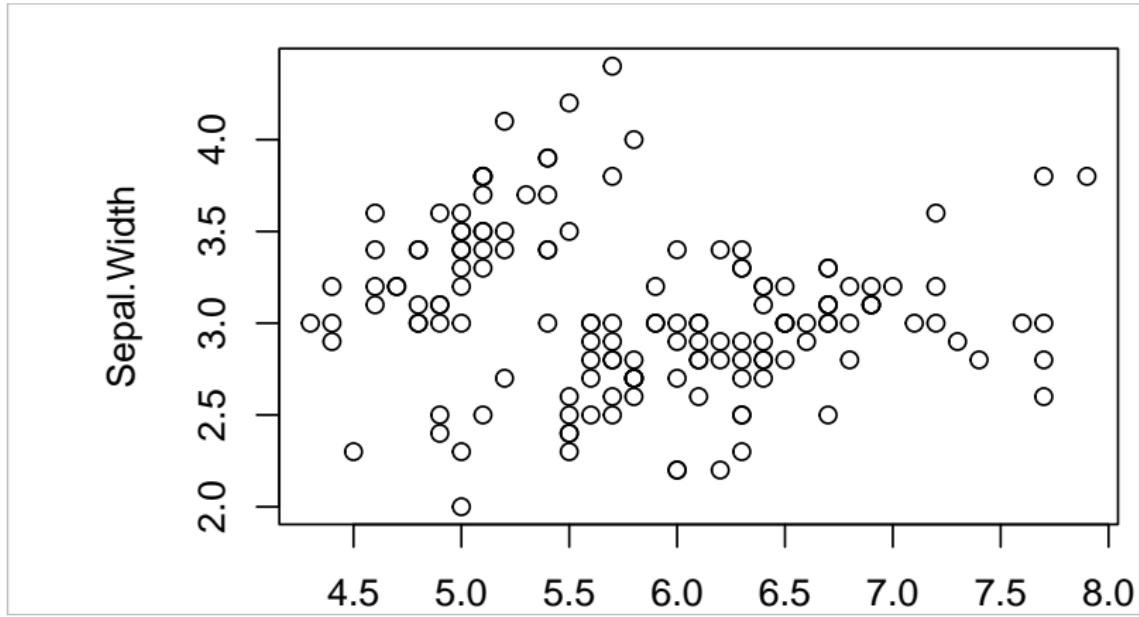
par : order of margin elements

```
plot(Sepal.Width~Sepal.Length, data=iris, axes=F, ann=F)
mtext(c("margin / side 1",2:4), side=1:4, las=1) ; box()
```



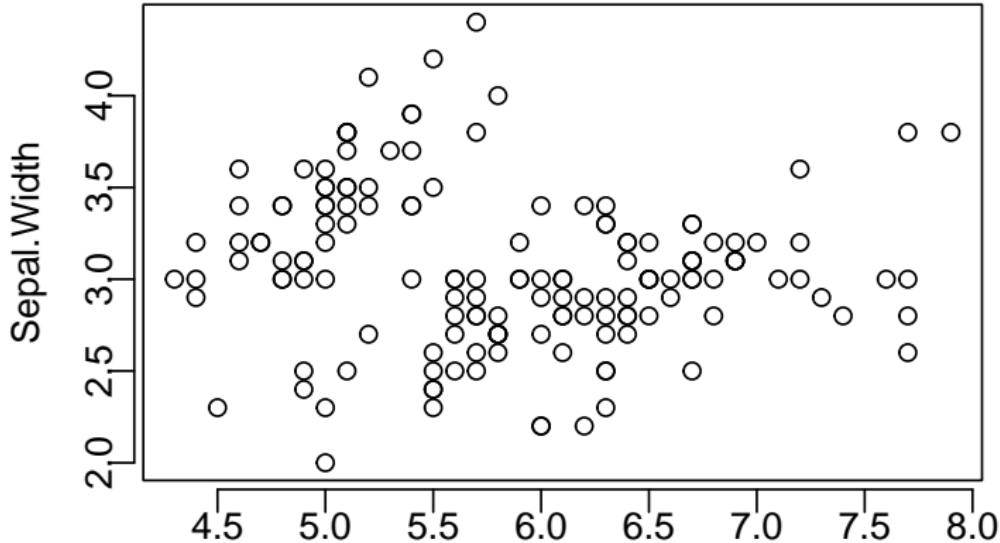
par : mar - margins

```
par(mar=c(2,6,1,0.5)) # unit: Lines of text  
plot(Sepal.Width~Sepal.Length, data=iris)
```



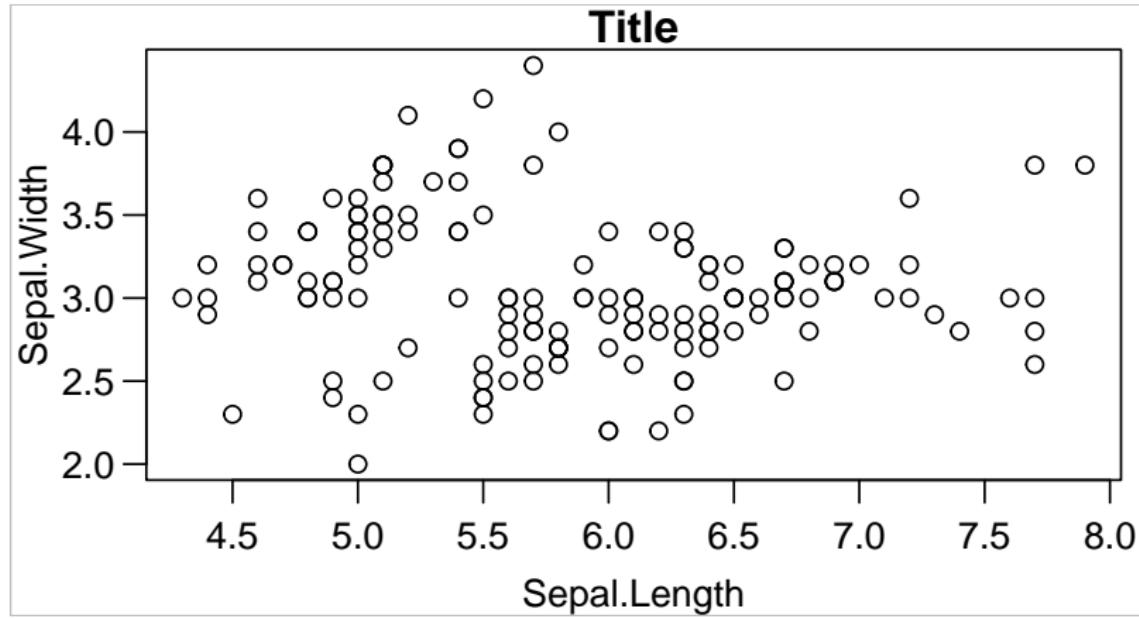
## par : mgp - margin placements

```
par(mar=c(2,6,1,0.5), mgp=c(2.1, 0.5, 0.2)) # dist. title,  
plot(Sepal.Width~Sepal.Length, data=iris) # numbers, line
```



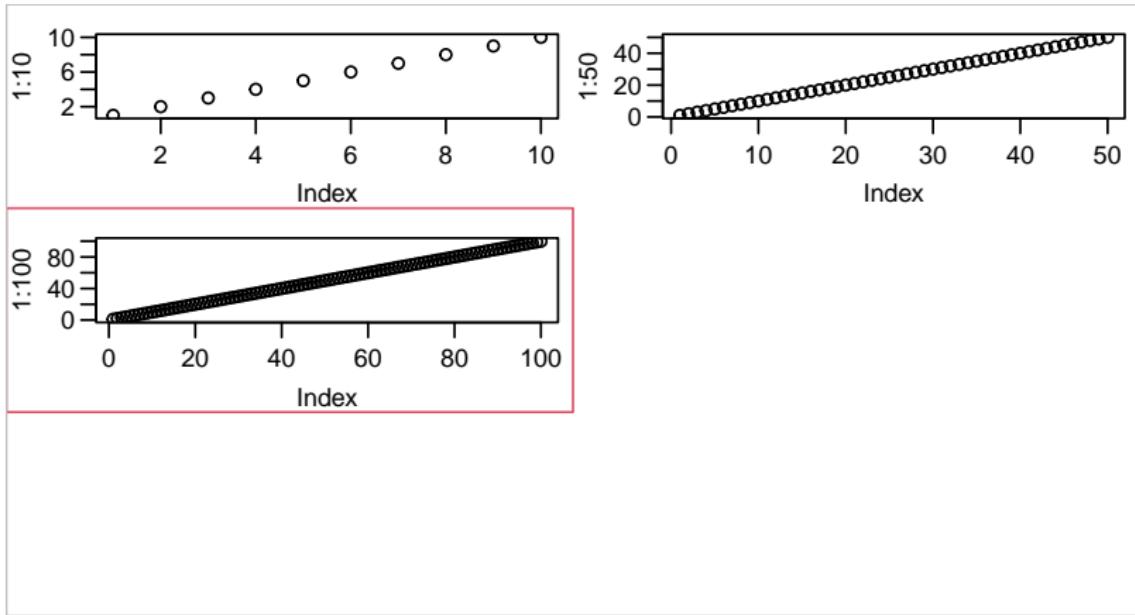
par : Good setting for small margins

```
par(mar=c(3,3,1,0.5), mgp=c(2,0.7,0), las=1)
plot(Sepal.Width~Sepal.Length, data=iris, main="Title")
```



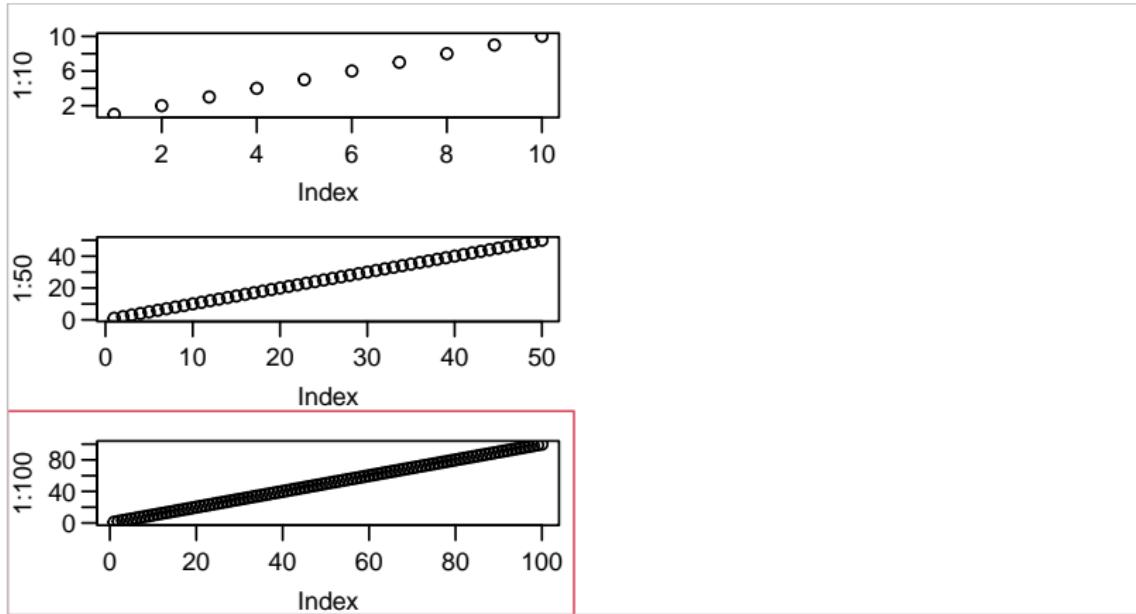
par : **mfrow** - **m**ultiple **f**igures, filled **r**ow-wise

```
par(mfrow=c(3,2), mar=c(3,3,1,0.5), mgp=c(2,0.7,0), las=1)
plot(1:10); plot(1:50); plot(1:100); box("figure", col=2)
```



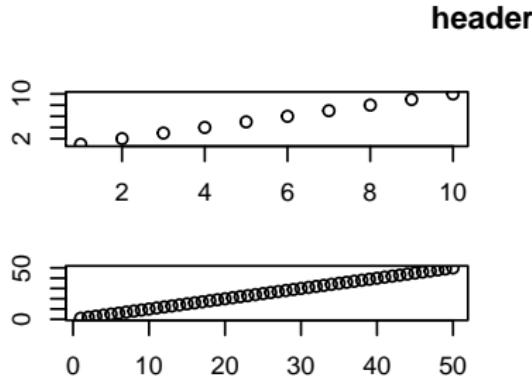
par : mfcol - multiple figures, filled column-wise

```
par(mfcol=c(3,2), mar=c(3,3,1,0.5), mgp=c(2,0.7,0), las=1)
plot(1:10); plot(1:50); plot(1:100); box("figure", col=2)
```



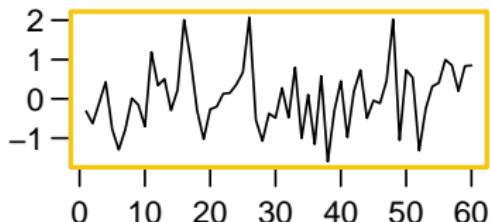
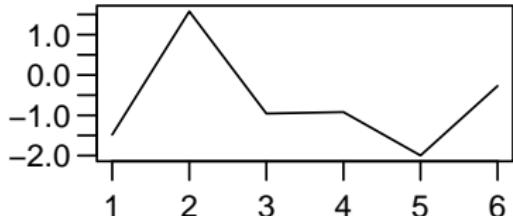
par : oma - outer margins

```
par(oma=c(0,4,3,0), mfcoll=c(3,2), mar=c(3,3,1,0.5) )  
plot(1:10); plot(1:50); title(main="header", outer=TRUE)
```



## box and par(mfrow/mar/oma) overview

```
par(    mfrow=c(2,2) , oma=c(0,1,3,0) )
```

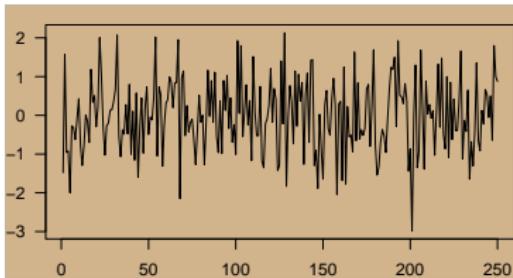


- box() = box('plot')
- box('figure'), after par(mfrow) or layout()
- box('inner')
- box('outer'), if outer margins were set by par(oma)

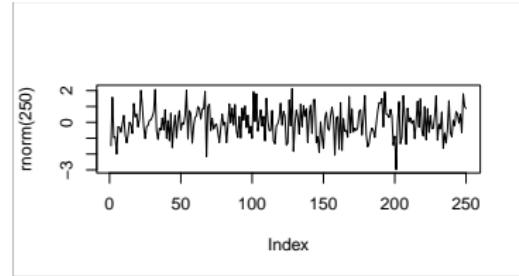


## Restore `par` settings

```
op <- par(mar=c(2,2,1,0),  
          las=1, bg="tan")  
plot(rnorm(250), type="l")  
box("figure", col="grey70")
```



```
par(op)  
plot(rnorm(250), type="l")  
box("figure", col="grey70")
```

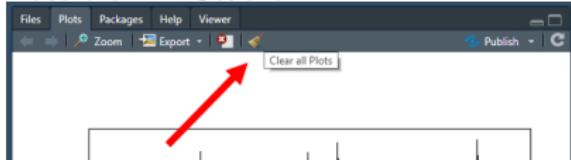


Alternative:

```
dev.off() # graphics.off()
```

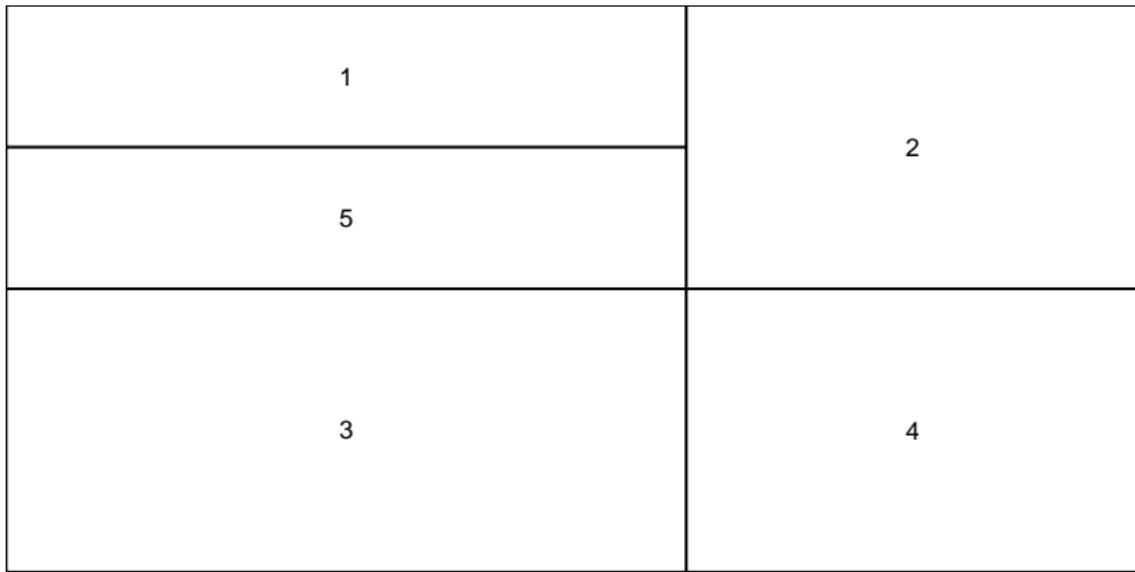
Close device -> all defaults are restored in the next plot.

Possible in RStudio:



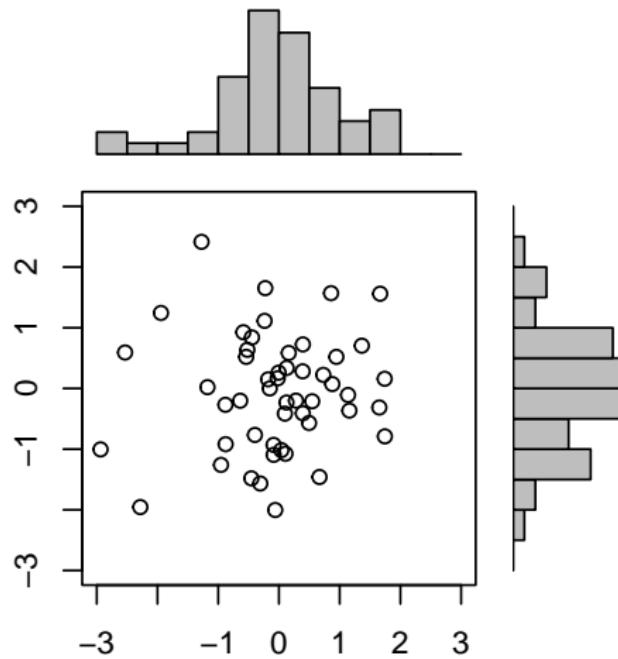
## Flexible multipanel diagrams with `layout`

```
lay <- layout(matrix(c(1,1,2,2,
                      5,5,2,2,
                      3,3,4,4,
                      3,3,4,4), ncol=4, byrow=TRUE),
               widths=c(6,6,4,4))
layout.show(lay)
```



layout example with marginal histogram

Code in Examples-section of `?layout`:



This is possible with `par(fig)`, see [SO post](#), but `layout` is more elegant.

### **multipanel diagrams, par settings (parameters) for graphics:**

- ▶ `par (mfrow,mfcol, mar,oma,mgp, las, bg)`
- ▶ `box (plot, figure, inner, outer)`
- ▶ `layout , layout.show`

Report unclear tasks in the forum

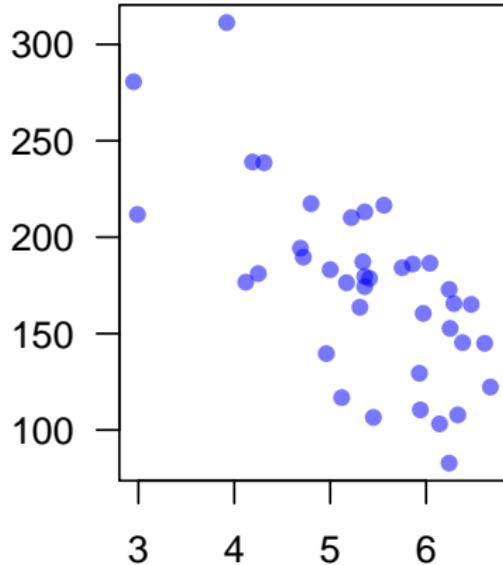
Highlight the topics from this lesson in your RefCard



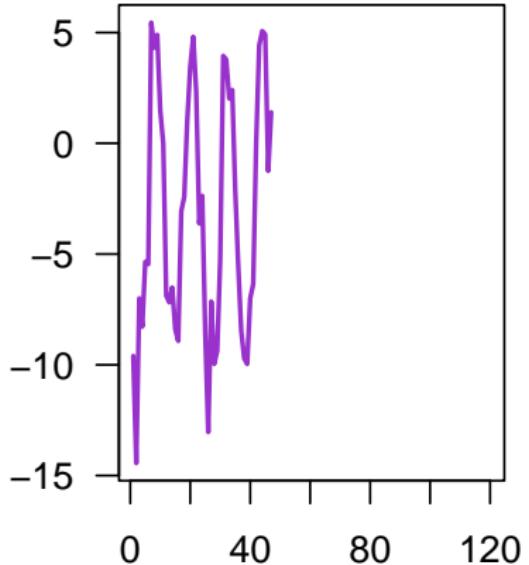
## Exercise T9 - intended solution

Zugspitze 1900:2022

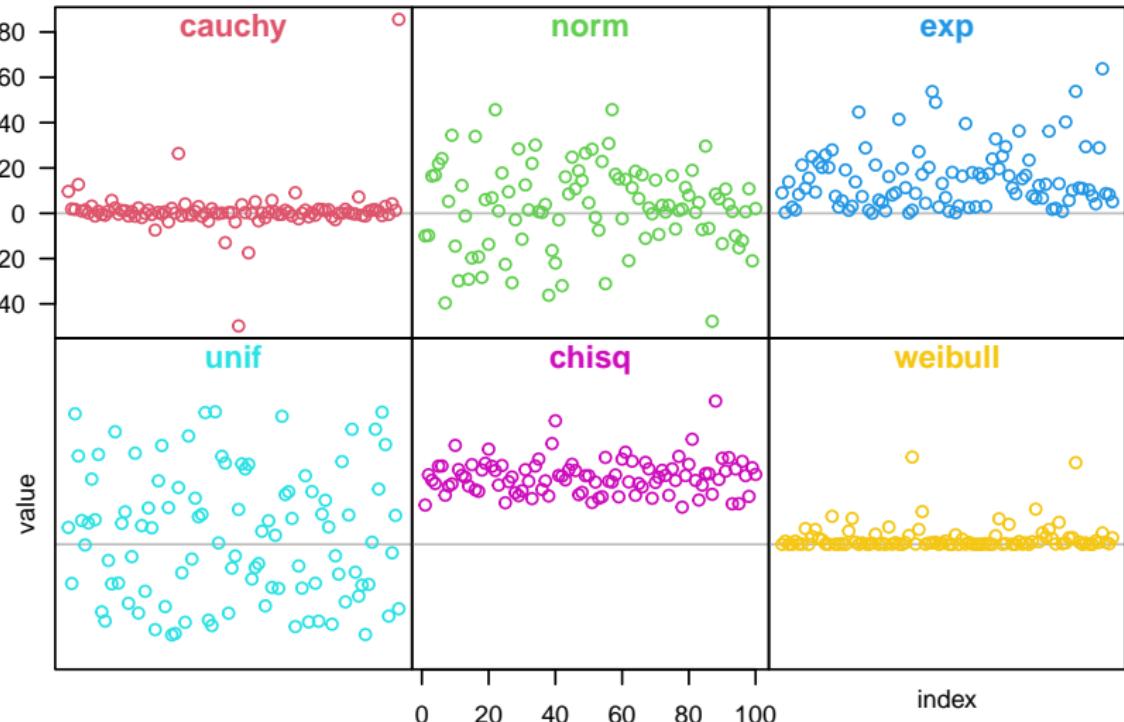
Sunshine [monthly hours]  
over cloud cover [eights]



First 120 months of Temperature

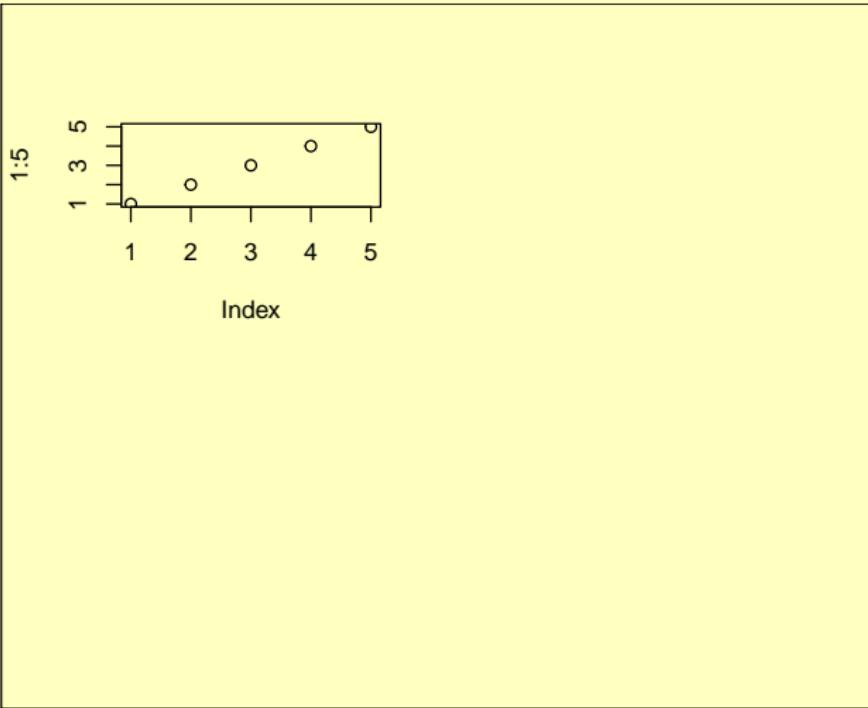


## Exercise B2 - intended solution



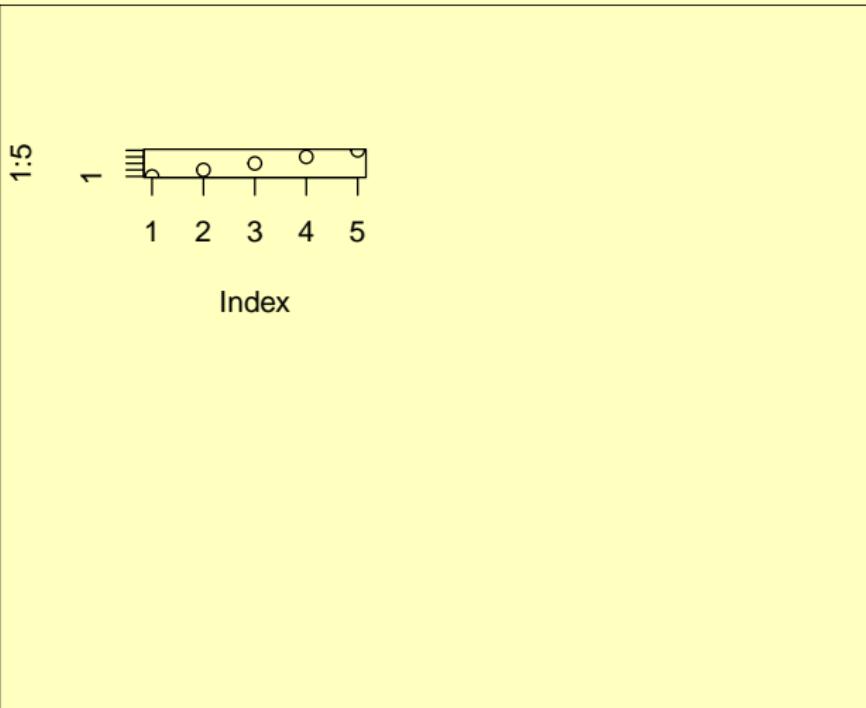
## par order important for cex and mfrow |

```
par(cex=1, mfrow=c(2,2))  
plot(1:5) # cex reduced after mfrow
```



## par order important for cex and mfrow II

```
par(mfrow=c(2,2), cex=1)  
plot(1:5) # cex manually reset to 1
```

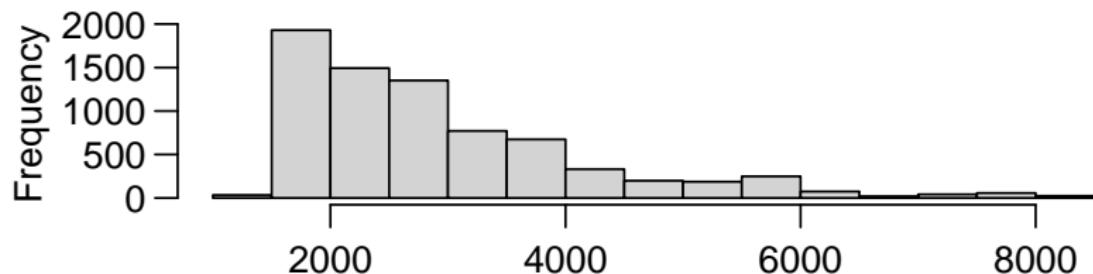


- 1. Intro
- 2. Basics
- 3. Objects
- 4. Data
- 5. **Graphics**
  - 5.1 Scatterplots
  - 5.2 Line plots
  - 5.3 Barplots
  - 5.4 Low level plotting
  - 5.5 Composition
  - 5.6 Distribution plots**
  - 5.7 Export
  - 5.8 Outlook
- 6. Flow control

## Histogramm: Number of values per range (bin)

```
h <- hist(EuStockMarkets, las=1, main="90s stock index")
```

90s stock index



EuStockMarkets

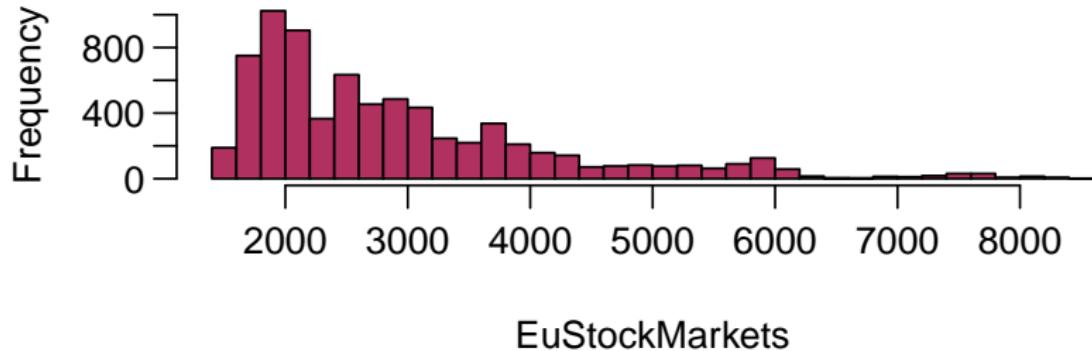
```
str(h)
## List of 6
## $ breaks  : int [1:16] 1000 1500 2000 2500 3000 ...
## $ counts   : int [1:15] 34 1930 1493 1352 771 ...
## $ density  : num [1:15] 9.14e-06 5.19e-04 ...
## $ mids     : num [1:15] 1250 1750 2250 2750 3250 ...
## $ xname    : chr "EuStockMarkets"
## $ equidist: logi TRUE
```



hist : number of bins (classes)

```
hist(EuStockMarkets, las=1, breaks=50, col="maroon")
```

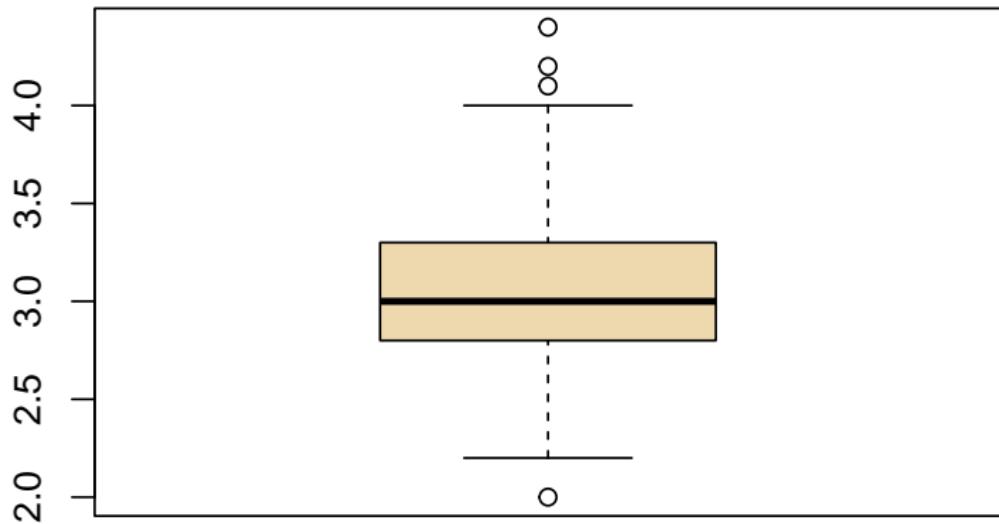
Histogram of EuStockMarkets



## Boxplot: Quartiles displayed graphically

```
summary(iris$Sepal.Width)
##      Min. 1st Qu. Median      Mean 3rd Qu.      Max.
## 2.000   2.800   3.000   3.057   3.300   4.400
```

```
boxplot(iris$Sepal.Width, col="wheat2")
```

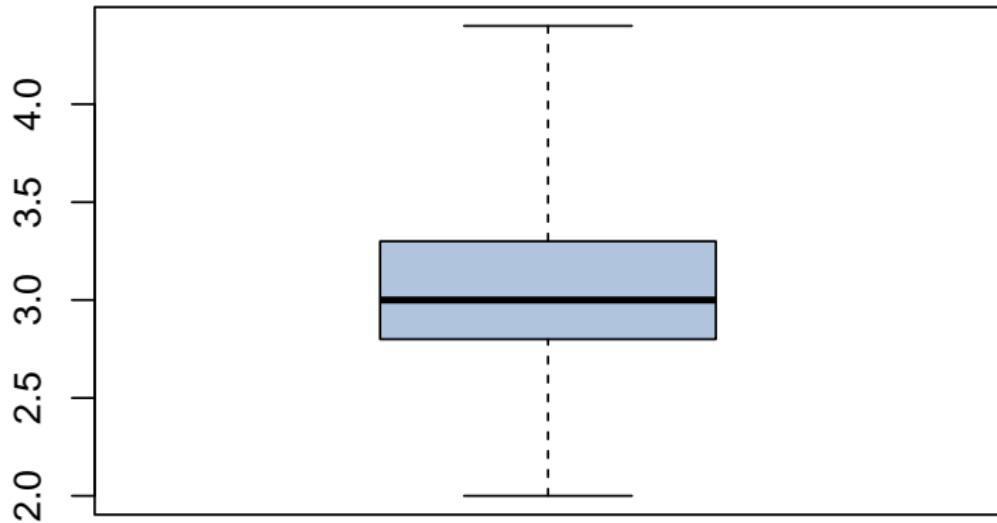


Points are usually not an 'outlier' but belong to the data

`boxplot()` by default plots 'outliers' as individual points.

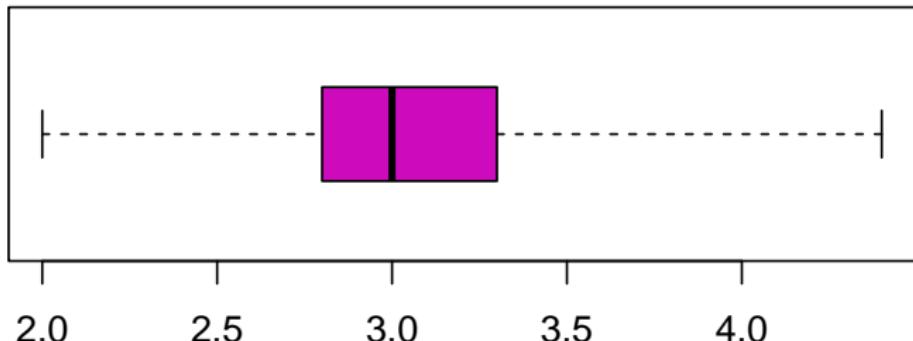
The threshold is an arbitrary distance from the IQR box, beyond which data values are considered outliers. (IQR = InterQuartileRange = range between 25% and 75% quantiles of the data)

```
boxplot(iris$Sepal.Width, range=0, col="lightsteelblue")
```

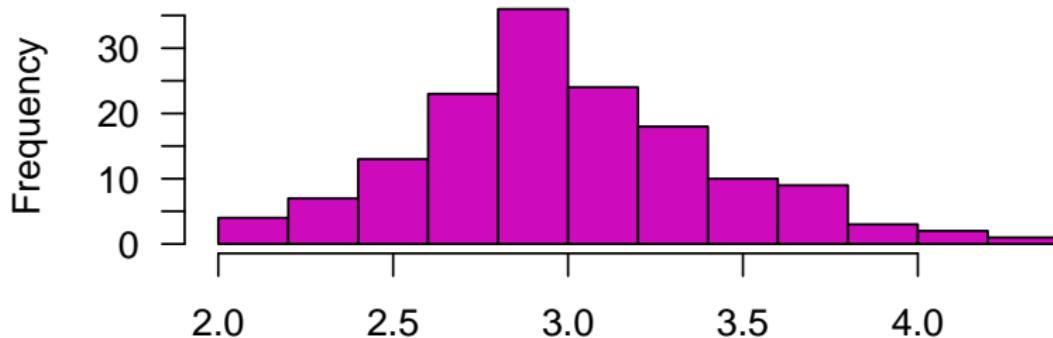


## Horizontal boxplots

```
boxplot(iris$Sepal.Width, horizontal=TRUE, range=0, col=6)
```

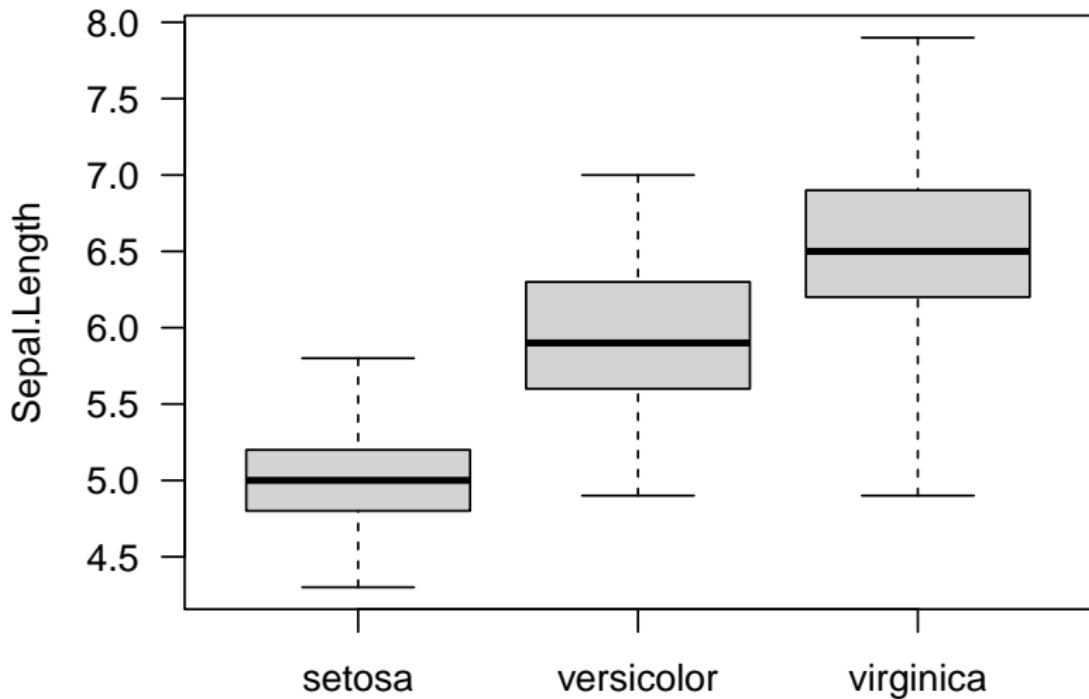


```
hist(iris$Sepal.Width, col=6, main="", las=1)
```



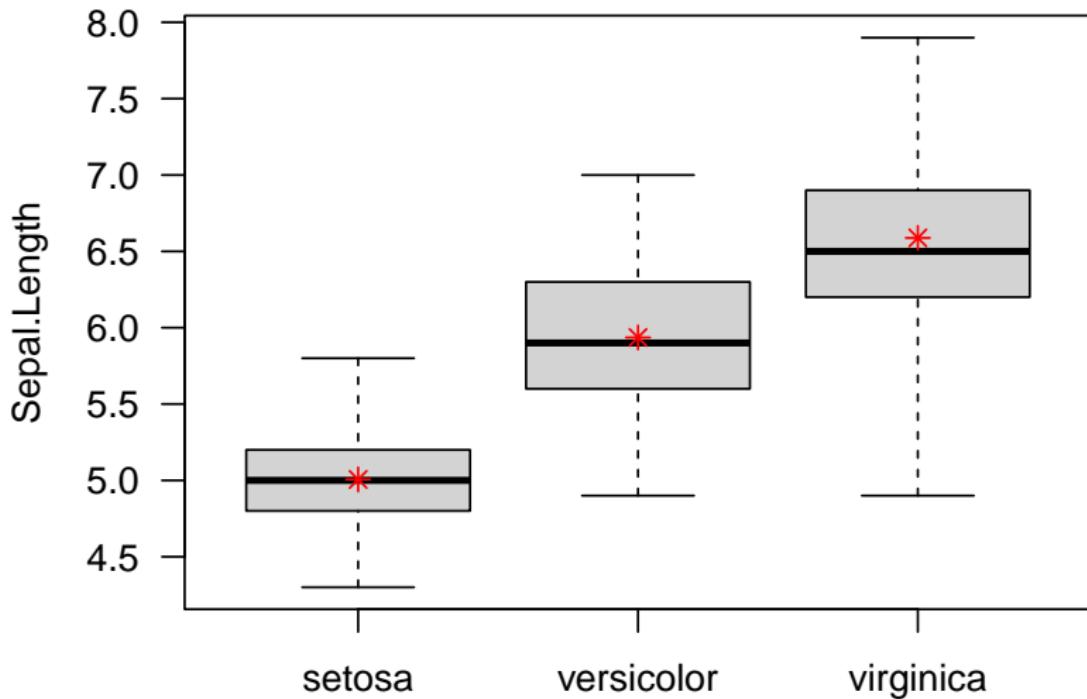
## Boxplots of several columns

```
boxplot(Sepal.Length ~ Species, data=iris, range=0, las=1)
# also potentially useful (and simpler as for barplots):
values ~ group1+group2 # for doubly grouped data
```

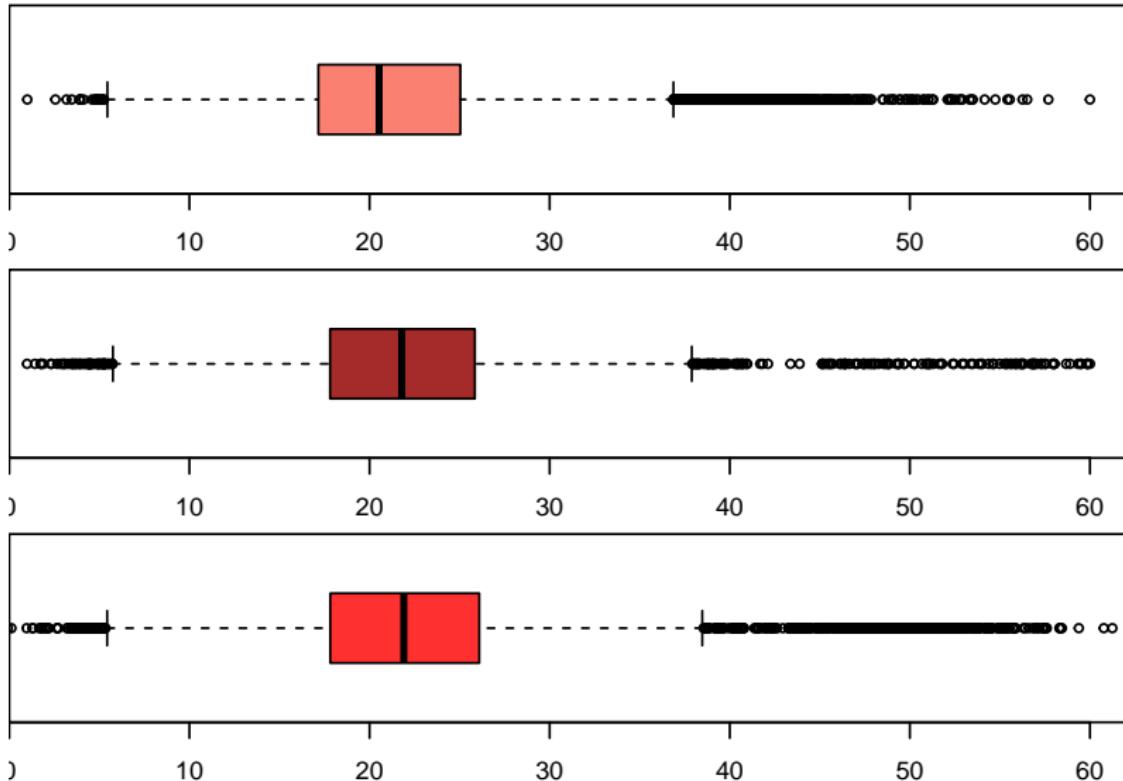


## Boxplots with averages

```
boxplot(Sepal.Length ~ Species, data=iris, range=0, las=1)
averages <- tapply(iris$Sepal.Length, iris$Species, mean)
points(1:3, averages, pch=8, col="red")
```

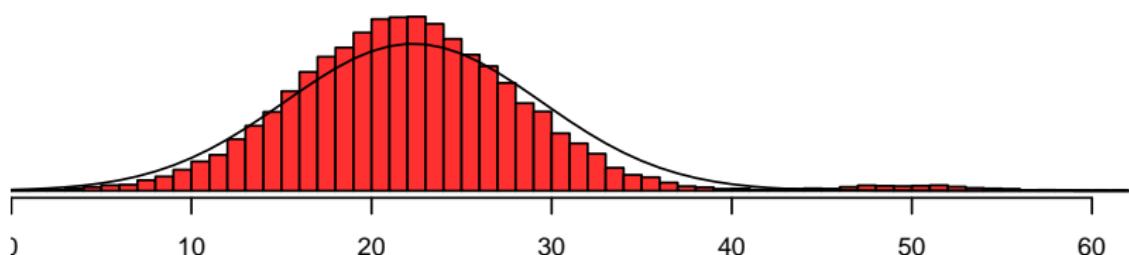
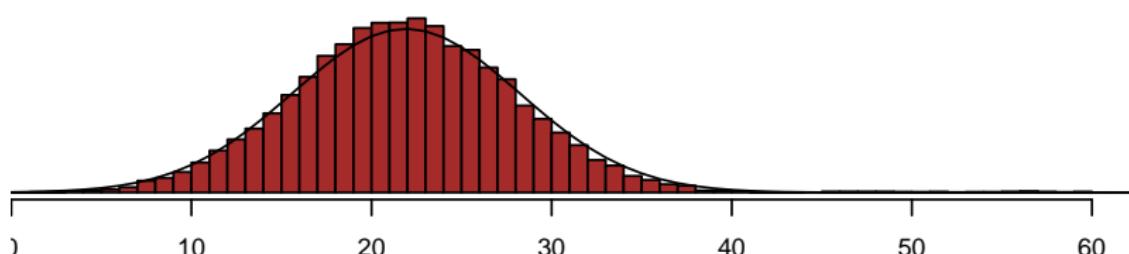
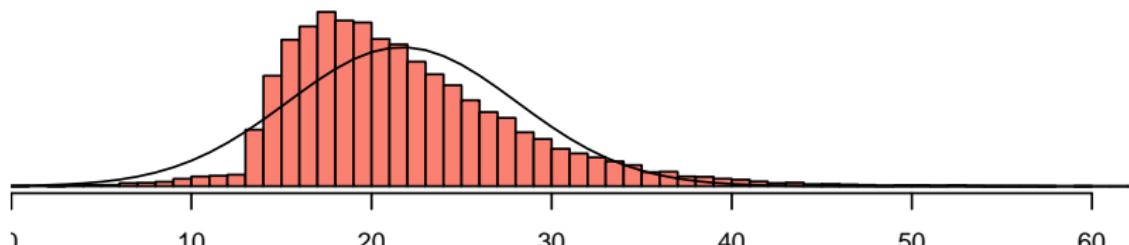


Boxplots should only be compared, ...



... if the underlying distributions are similar!

see e.g. [why not to trust statistics](#) and [boxplot R](#)



### histograms and boxplots: graphical representations of value distributions:

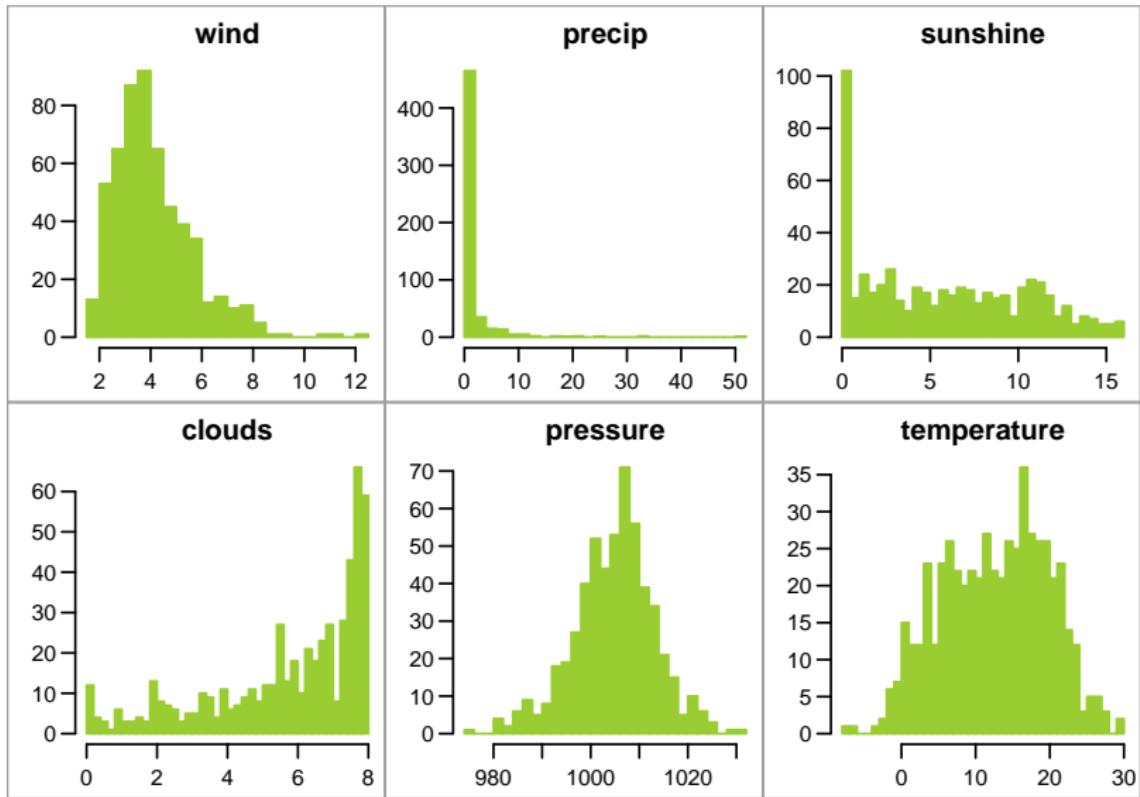
- ▶ `hist (breaks, las,col,main)`
- ▶ `boxplot (range=0, horizontal)`
- ▶ `boxplot (y ~ x)`
- ▶ Use boxplots to compare many groups, histograms to actually see the distribution

Report unclear tasks in the forum

Highlight the topics from this lesson in your RefCard

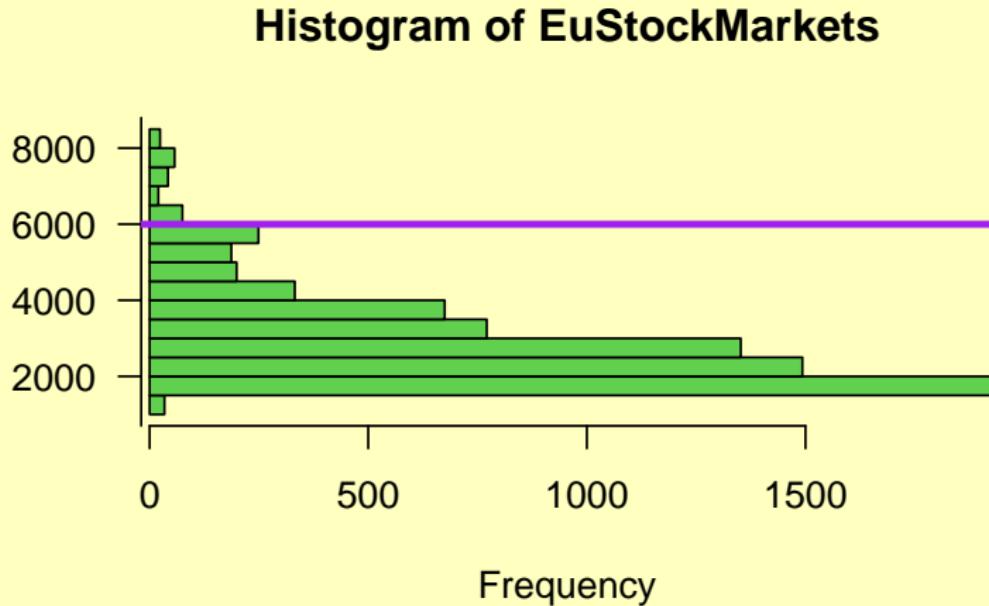


## Possible solution to Exercise T11



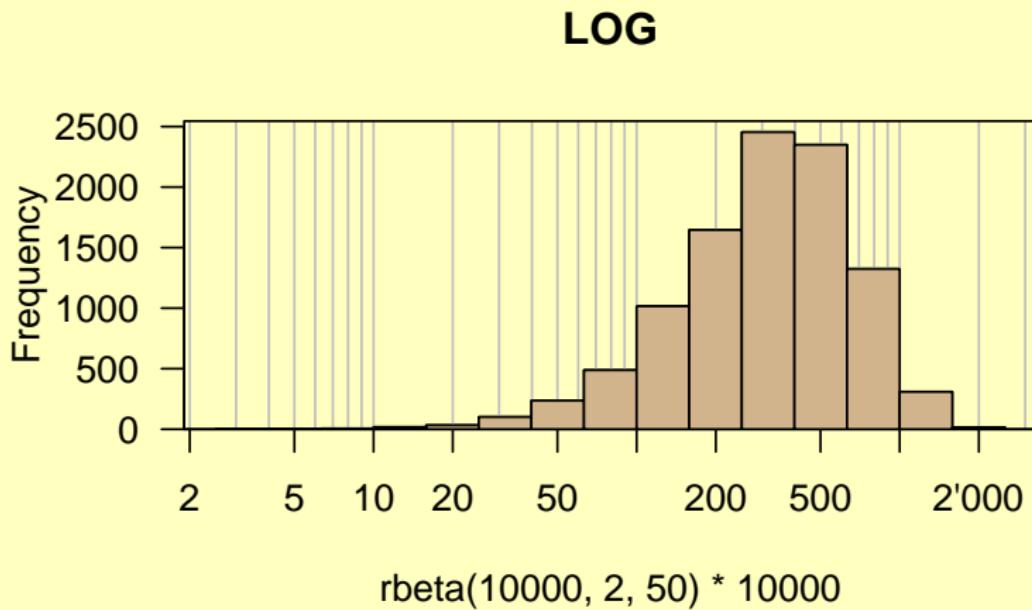
## Horizontal histogramm

```
hpos <- berryFunctions::horizHist(EuStockMarkets, col=3)  
abline(h=hpos(6000), col="purple", lwd=3)
```



## Logarithmic histogramm

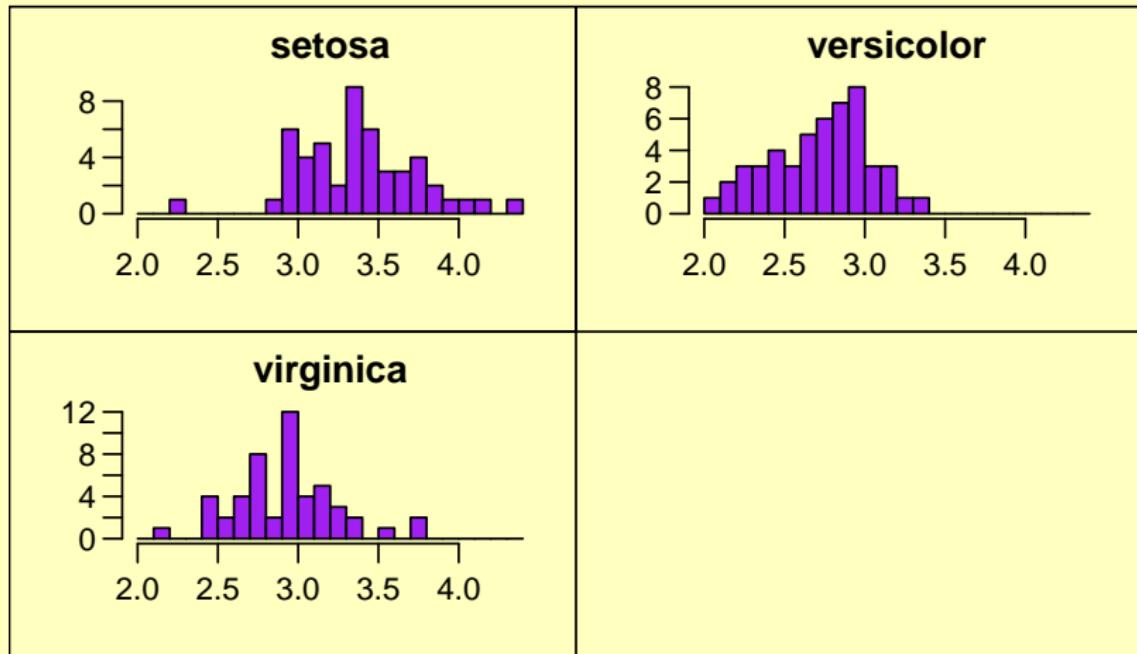
```
berryFunctions::logHist(rbeta(1e4, 2, 50)*1e4, main="LOG")
```



## Grouped histogramm

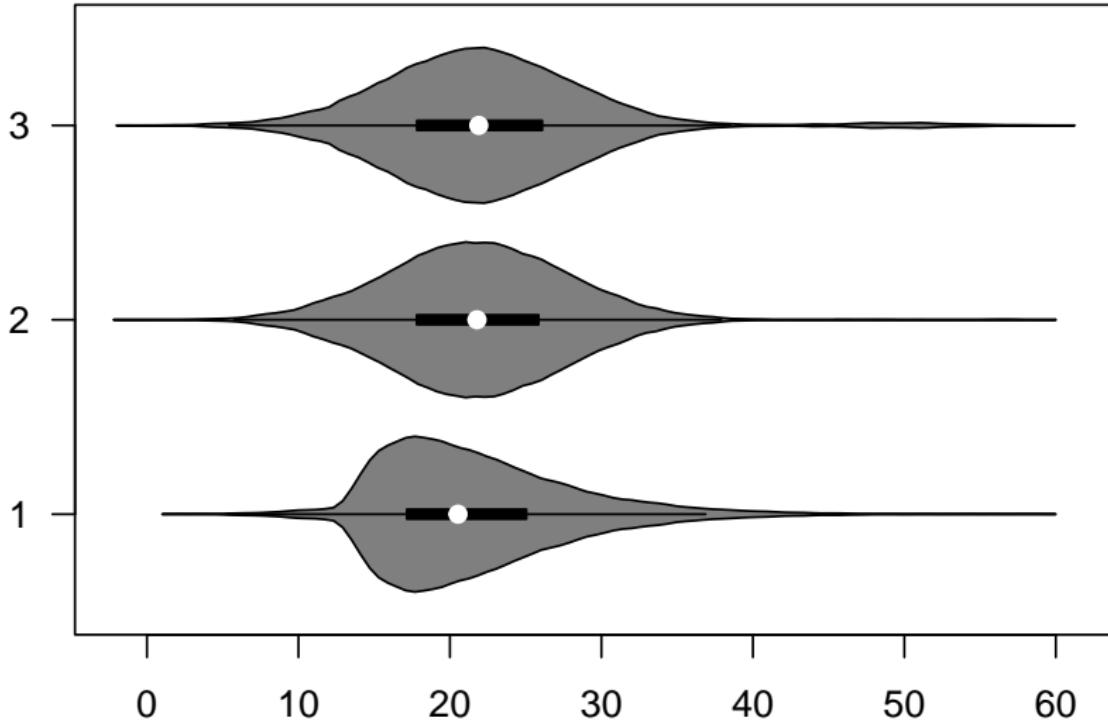
```
berryFunctions::groupHist(iris, "Sepal.Width", "Species",  
                           unit="cm")
```

Histograms of Sepal.Width [cm] in iris, grouped by Species



## Violinplots: good middle ground between boxplot and histogram

```
vioplot::vioplot(x,y,z, h=0.5, horizontal=TRUE)
```



- 1. Intro
- 2. Basics
- 3. Objects
- 4. Data
- 5. Graphics
  - 5.1 Scatterplots
  - 5.2 Line plots
  - 5.3 Barplots
  - 5.4 Low level plotting
  - 5.5 Composition
  - 5.6 Distribution plots
  - 5.7 Export
  - 5.8 Outlook
- 6. Flow control

## Save graphics as PDF

File formats:

PDF : Vector-graphic (zoomable), several pages

PNG : Raster-graphic, small file size

JPG : Anti-Aliasing, for photos (not images)

```
pdf("SampleGraphic.pdf") # open PDF-graphics-device
par(mar=c(3,3,1,0), las=1)
# These settings are for the pdf graph
plot(rnorm(700), col=colors(), col.axis="turquoise")
# This plot doesn't appear in the default Rstudio device
dev.off() # Close external device
```

width and height both default to 7 inches (17.8 cm).

Indenting code indicates that it belongs together (to humans, not to R).



## Save multiple graphics

Each graphic has its own page in the PDF:

```
pdf("SampleGraphic.pdf", height=5)
  plot(rnorm(700), col=colors(), col.axis="turquoise")
  boxplot(iris)
  hist(rnorm(700))
dev.off()
```

Group rows, send to R blocked (in chunks)

```
7777 # some preceding code
{
  pdf("SampleGraphic.pdf")
  boxplot(iris)
  hist(rnorm(700))
  dev.off()
}
888 # more other things
```



## Tips and tricks on exporting

Code between `pdf/postscript/png/jpeg/bmp/tiff/xfig/X11/svg`

and `dev.off()` is routed to the external graphic.

In case of an error within blocked execution, do not forget to call `dev.off()` manually.

Open PDFs in Sumatra (only on Windows OS) so that they can be edited while they are open. Changes are visible live in real time.

Comes shipped with Rstudio. [Sumatra Notes](#)

Ubuntu: evince (document viewer) + eog (eye of gnome, viewer)

Open graphic file from R:

```
{ pdf("SampleGraphic.pdf")
  boxplot(iris)
  dev.off()
  file.show("SampleGraphic.pdf") # fails for Umläute
}
```

```
berryFunctions::openFile()
```

```
berryFunctions::openPDF() # für SumatraPDF
```



## Save graphics as PNG

```
png("FileName.png", width=12, height=9, units="in",
    res=300, bg="transparent", pointsize=14)
par(mar=c(3,3,1,0.5), las=1)
plot(1:20)
title(main="Plot title")
points(5,9)
dev.off()
```

Use `units` in inches to be able to change to `pdf` if needed (only handles inches, not cm).

Consecutive numbering for multiple images:

```
png("Image_%03d.png")
par(bg="thistle")
plot(2:8, type="h", lwd=7) ; plot(1:8)
dev.off() # without dev.off, the last file is still empty
```

`_%03d.png` \_001.png Good choice :)

`_%3d.png` \_ 1.png Spaces in filenames: generally not a good idea

`_%d.png` \_1.png Sorted by file name, `10.png` comes before `2.png`.



### save graphics in a file:

- ▶ `pdf("file.pdf") ; plot(42) ; dev.off()`
- ▶ Use a PDF viewer that does not lock opened files against editing
- ▶ `png (width, height, units, res, bg, pointsize)`
- ▶ `{}` to execute code blocks

Report unclear tasks in the forum

Highlight the topics from this lesson in your RefCard

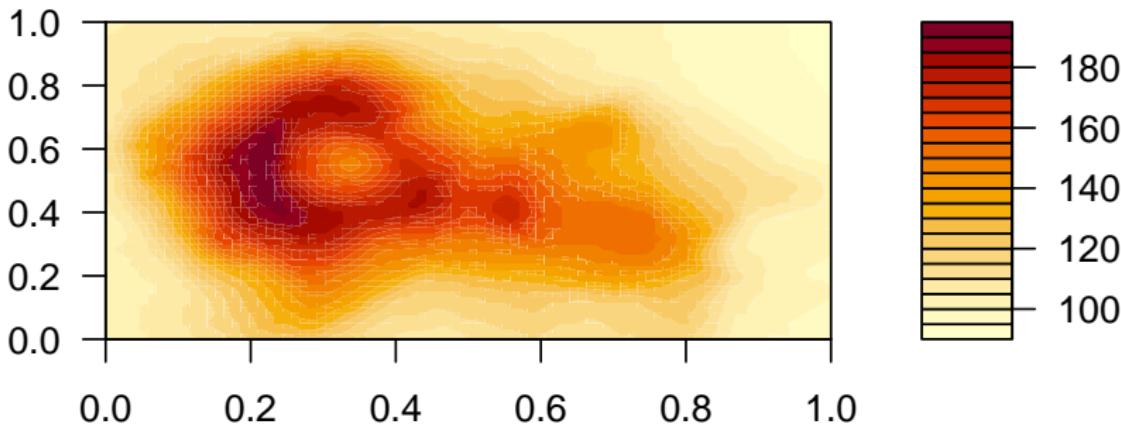


- 1. Intro
- 2. Basics
- 3. Objects
- 4. Data
- 5. **Graphics**
  - 5.1 Scatterplots
  - 5.2 Line plots
  - 5.3 Barplots
  - 5.4 Low level plotting
  - 5.5 Composition
  - 5.6 Distribution plots
  - 5.7 Export
  - 5.8 Outlook
- 6. Flow control

## Animation

Using `ffmpeg`:

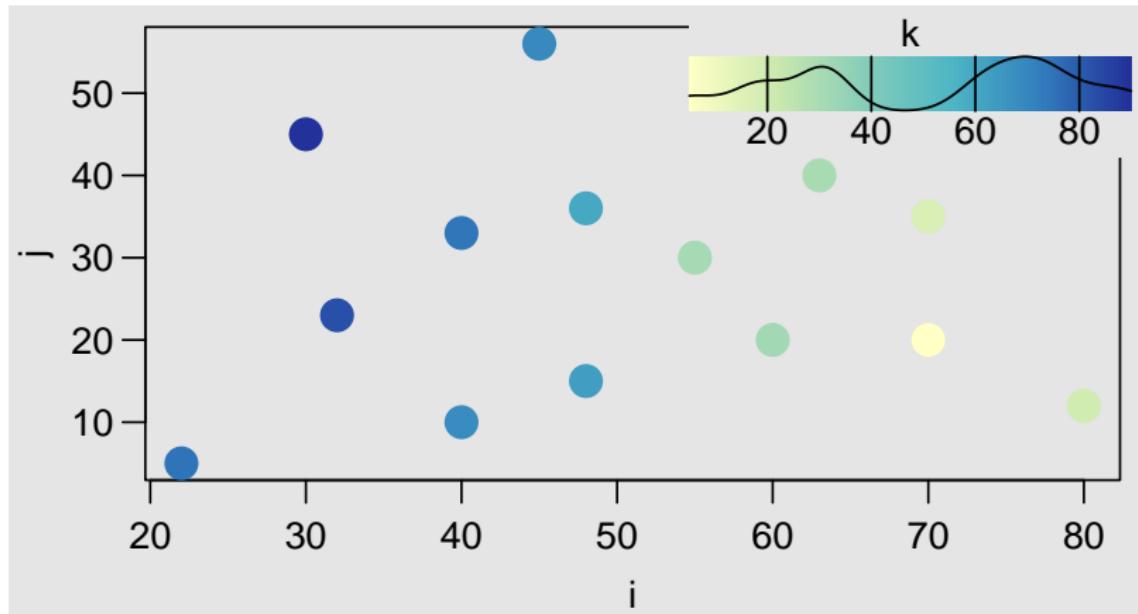
```
set.seed(12)
volc <- apply(volcano, 1:2, function(x) x+cumsum(rnorm(100)))
library(animation); library(pbapply)
saveVideo(pbsapply(1:100, function(i)
    filled.contour(volc[i,,], zlim=range(volc))),
    video.name="volc.mp4", interval=0.07,
    ffmpeg="C:/ff_folder/bin/ffmpeg.exe")
```



## 3D points in color scale

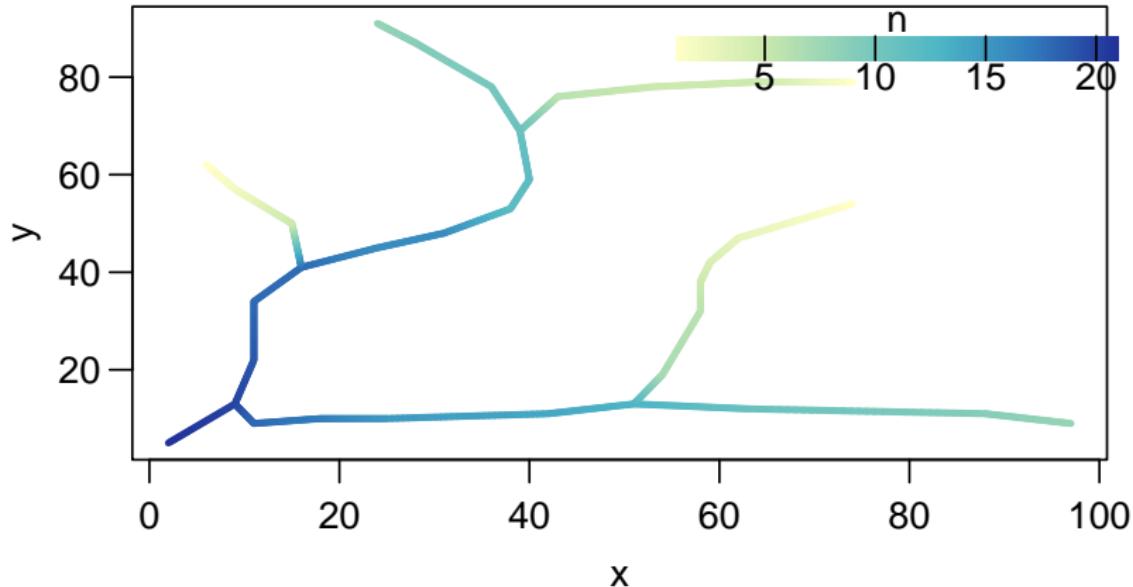
```
i <- c(22, 40, 48, 60, 80, 70, 70, 63, 55, 48, 45, 40, 30, 32)  
j <- c( 5, 10, 15, 20, 12, 20, 35, 40, 30, 36, 56, 33, 45, 23)  
k <- c(75, 68, 63, 32, 20, 05, 17, 30, 31, 60, 69, 74, 90, 83)
```

```
berryFunctions::colPoints(i,j,k, add=FALSE, y1=0.75,  
                           density=list(bw=5), cex=2)
```



## 3D lines in color scale

```
tfile <- system.file("extdata/rivers.txt", package="berryFunctions")
rivers <- read.table(tfile, header=TRUE, dec=",")
berryFunctions::colPoints(x,y,n, data=rivers, add=FALSE,
                           lines=TRUE, lwd=3, y1=0.8, density=FALSE)
```



## Tabular data in color scale

```
berryFunctions::tableColVal(t(longley[5:14,-6]), digits=1,  
                           nameswidth=0.2, cex=0.6)
```

t(longley[5:14, -6])	1951	1952	1953	1954	1955	1956	1957	1958	1959	1960
GNP.deflator	96.2	98.1	99.0	100.0	101.2	104.6	108.4	110.8	112.6	114.2
GNP	329.0	347.0	365.4	363.1	397.5	419.2	442.8	444.5	482.7	502.6
Unemployed	209.9	193.2	187.0	357.8	290.4	282.2	293.6	468.1	381.3	393.1
Armed.Forces	309.9	359.4	354.7	335.0	304.8	285.7	279.8	263.7	255.2	251.4
Population	112.1	113.3	115.1	116.2	117.4	118.7	120.4	122.0	123.4	125.4
Employed	63.2	63.6	65.0	63.8	66.0	67.9	68.2	66.5	68.7	69.6



This entire chapter created graphics using the so-called base R.  
A popular 'dialect' is available in the `ggplot2` package.  
Both approaches have their advantages and their right to exist.  
Some helpful links about this:

<http://minimaxir.com/2017/08/ggplot2-web>

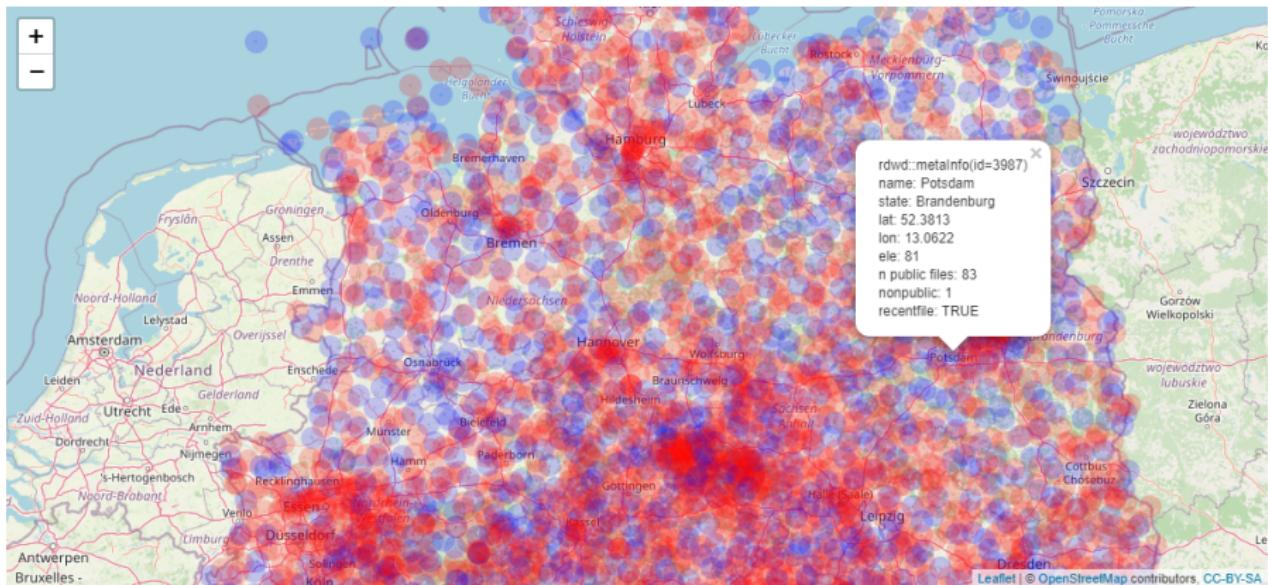
[https://uc-r.github.io/ggplot\\_intro](https://uc-r.github.io/ggplot_intro)

<http://flowingdata.com/2016/03/22/comparing-ggplot2-and-r-base-graphics>



## leaflet : interactive maps (This example online)

```
library(rdwrd) ; data(geoIndex) ; library(leaflet)
leaflet(geoIndex) %>% addTiles() %>%
  addCircles(~lon, ~lat, radius=900, stroke=F, color=~col) %>%
  addCircleMarkers(~lon, ~lat, popup=~display, stroke=F, color=~col)
```



### various graphics for specific applications:

- ▶ `berryFunctions::colPoints+tableColVal`
- ▶ `leaflet`
- ▶ `ggplot2` for a different dialect

*There are no exercises for this lesson*

Report unclear tasks in the forum

Highlight the topics from this lesson in your RefCard



- 1. Intro
  - 2. Basics
  - 3. Objects
  - 4. Data
  - 5. Graphics
  - 6. Flow control
- 6.1 Conditions
  - 6.2 Debugging
  - 6.3 Loops



**if** : execute something if a condition is fulfilled

7-3 > 2	TRUE
<b>class</b> (7-3 > 2 )	logical = boolean = truth value
<b>if</b> (7-3 > 2) 18	condition is met, 18 is executed
<b>if</b> (7-3 > 5) 18	condition is wrong, so nothing happens*
<b>if</b> (7-3 > 5) 18 <b>else</b> 17	condition is wrong, so 17 is returned

\*: technically, **NULL** is returned invisibly

Syntax for a single truth value:

```
if(condition) {expression1} else {expression2}
```

Syntax for a vector with multiple T/F values:

```
ifelse(condition, expression1, expression2)
```

If condition == TRUE, expression1 is executed,

If condition == FALSE, expression2 is run.



## Bundling multiple commands |

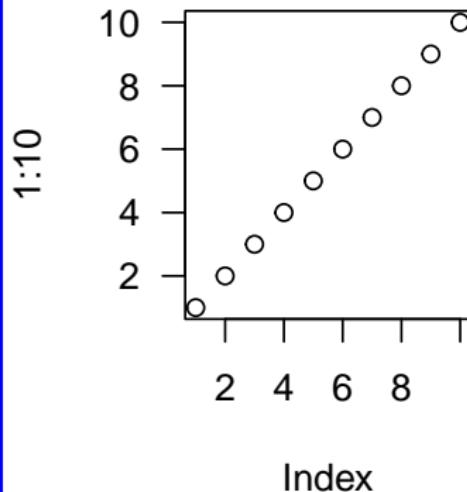
```
# curly braces:  
if(TRUE)  
{  
  plot(1:10, main="TRUE as condition")  
  box("figure", col="blue", lwd=3)  
} else  
# perform this if TRUE above is replaced with FALSE:  
{  
  plot(rnorm(500), main="FALSE in code")  
}  
# The last brackets are technically optional  
# Indenting code makes it readable for humans
```

```
par(mfrow=c(1,2), cex=1, las=1)
```

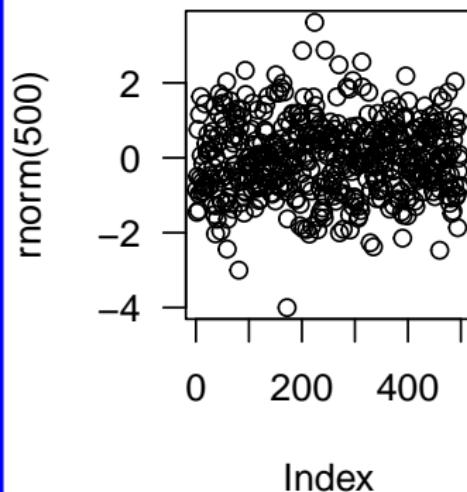


## Bundling multiple commands II

**TRUE as condition**



**FALSE in code**



Vectorization: `if(c) e1 else e2` vs `ifelse(c, e1, e2)`

```
v <- c(13, 14, 15, 16, 17)
v>14
## [1] FALSE FALSE  TRUE  TRUE  TRUE

ifelse(v>14, v+10, NA) # for vector
## [1] NA NA 25 26 27

if(v>14) v+10 else NA # for a single value
## Error in if (v > 14) v + 10 else NA: the condition has
length > 1
```



## Note on `ifelse` execution

```
options(digits=3)
vec <- -3:4
log(vec)
## Warning in log(vec):  NaNs produced
## [1]    NaN    NaN    NaN  -Inf  0.000  0.693  1.099  1.386
```

```
ifelse(vec>0, log(vec), NA)
## Warning in log(vec):  NaNs produced
## [1]    NA    NA    NA    NA  0.000  0.693  1.099  1.386
```

`ifelse` evaluates both expressions completely (hence the warning in the second example).

It then selects from each result elementwise (according to the condition), recycling vectors if needed.



## Note on curly braces

- ▶ `if(c){ex1}` is valid code, so R does not expect an `else`.
- ▶ When executing line by line (e.g. in a script),  
    `}` and `else` must be on the same line.
- ▶ In functions etc., this is also good practice, but technically this works:

```
{  
if(cond  
{  
  ex1a  
  ex1b  
}  
else # normally in previous line  
  ex2  
}
```



## Note on conditions, actual usage

```
if(logicalValue==TRUE) ... is unnecessary, as  
if(logicalValue) ... , this is enough, but  
if(isTRUE(logicalValue)) ... this helps with NAs.
```

Exemplary `if else` use: `mad` / `hist` source code:

[github.com/wch/r-source](https://github.com/wch/r-source) -> `src / library / stats / R / mad.R`

[github.com/wch/r-source](https://github.com/wch/r-source) -> `src / library / graphics / R / hist.R`

## Multiply nested conditionals

```
if(cond==a) b else if(cond==c) d else e
```

can be avoided with `switch`:

```
switch(cond,  
       a = b,  
       c = d,  
       e  
)
```



## Conditional return

Handy for not nesting conditions.

```
stat(1:10, "max") # 10  
stat(1:10, "sum") # message: no method 'sum' available.
```

```
stat <- function(x, fun) { # choose function to be used on x  
  if(fun=="mean"){  
    return(mean(x))}  
  else if(fun=="median"){  
    return(median(x))}  
  else if(fun=="max"){  
    return(max(x))}  
  else {message("no method '",fun,"' available.")}  
}
```

```
stat <- function(x, fun) { # conditional returns need no 'else'  
  if(fun=="mean" ) return( mean(x))  
  if(fun=="median") return(median(x))  
  if(fun=="max" ) return( max(x))  
  message("no method '",fun,"' available.")  
}
```

Real life examples: [1](#) [2](#) [3](#) [4](#)



### **actual 'programming': conditional code execution:**

- ▶ `if(condition) A else B`
- ▶ `ifelse(condition, A , B)`
- ▶ `{ , switch`

Report unclear tasks in the forum

Highlight the topics from this lesson in your RefCard

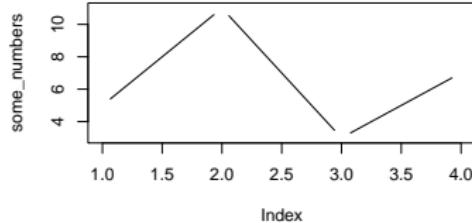
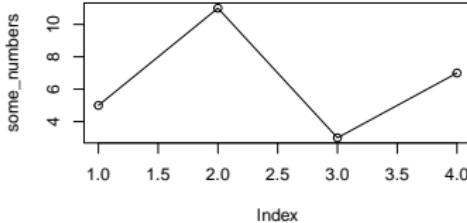


- 1. Intro
  - 2. Basics
  - 3. Objects
  - 4. Data
  - 5. Graphics
  - 6. Flow control
- 
- 6.1 Conditions
  - 6.2 Debugging
  - 6.3 Loops

## Default values for arguments

```
just_plot <- function(some_numbers, type="o")  
{  
  plot(some_numbers, type=type)  
}
```

```
just_plot(c(5,11,3,7) )    just_plot(c(5,11,3,7), type="c")
```



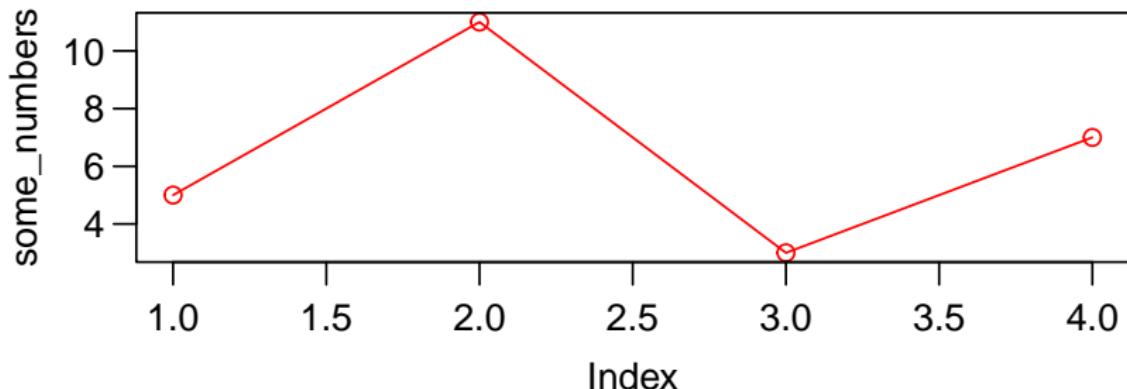
If an argument is not specified, the default value is used.  
Arguments without default must be specified.

## Arbitrary arguments to pass on

Ellipsis / three dots:

```
just_plot <- function(some_numbers, type="o", ...)  
{  
  # ... : pass arguments to plot()  
  plot(some_numbers, type=type, ...)  
}
```

```
just_plot(c(5,11,3,7), col="red", las=1)
```



... for separate functions: `berryFunctions::owa`

## Check inputs

Good functions check the input and give informative messages:

```
if(length(input)>1) stop("length must be 1, not ", length(input))

if(!is.numeric(i)) stop("i must be numeric, not ",toString(class(i)))

if(any(val > maxval)) warning(sum(val > maxval), " out of ",
                           length(val), " > maxval (",maxval,"), set to NA")
val[val>maxval] <- NA

if(any(diff(x)<0)) message("x not in ascending order.",
                            "Proceeding anyways")
```

`stop` : stop function execution, generate error

`warning` : continue execution, but give warning

`message` : inform only, do not warn

`cat` / `print` : should not be used, hard to capture

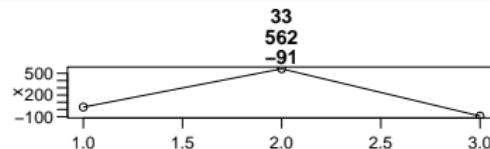
`suppressMessages` / `suppressWarnings` / `options(warn=0)`

-1 :ignore, 0 :show at end, 1 :show immediately, 2 :convert to error

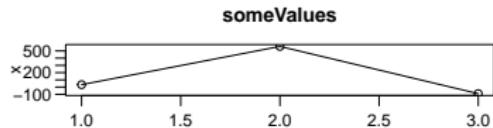


## Input names

```
someValues <- c(33,562,-91)
someFun <- function(x)
{
  plot(x, type="o", main=x)
}
someFun(someValues)
```



```
# with intended title
someFun <- function(x)
{
  n <- deparse(substitute(x))
  plot(x, type="o", main=n)
}
someFun(someValues)
```



`substitute(x)` does not evaluate the expression `x` - it returns the parse tree instead. `deparse(x)` transforms that to a character string.

```
checkInput <- function(x){
  n <- deparse(substitute(x)) # run before changing x
  x <- x[1:6]
  if(any(x<0)) stop("x (",n,") must be positive.", call.=FALSE)
}
checkInput(someValues)
## Error: x (someValues) must be positive.
```

## Scoping I

```
obj <- 17
helper <- function() obj-3
helper() # works fine
## [1] 14

helper <- function() y-3
main <- function(x) {y <- 2*x ; helper()}
main(8) # does not work!
## Error in helper(): object 'y' not found

helper <- function() get("y")-3
main(8) # gives the same error
```

R has lexical scoping. Objects (variables) that are not defined within a function are found in the parent environment of where they are **defined**, not in the parent frame of where they are **called**. The latter would be dynamic scoping and can experimentally be achieved with `dynGet`. More at e.g. `adv-r` or `trestletech`.

In which 3 ways can you solve the example above?



## Scoping II: 3 solutions to the previous example

```
# use dynamic scoping:  
main <- function(x) {y <- 2*x ; helper()}  
helper <- function() dynGet("y")-3  
main(8)  
## [1] 13  
  
# define helper within main function:  
main <- function(x)  
  {y <- 2*x ; helper <- function() y-3; helper()}  
main(8)  
## [1] 13  
  
# pass objects explicitly:  
main <- function(x) {y <- 2*x ; helper(y)}  
helper <- function(o) o-3  
main(8)  
## [1] 13
```

## Debugging: finding and fixing errors in (your own) code

- ▶ When one function calls the next and this in turn calls another and that generates an error:  `traceback()` to find the path of the error.
- ▶ Write  `browser()` into the source code (body) of a custom function. When called, R stops at that point and lets you try things in the function's environment with the objects there.
- ▶ Set  `options(error=recover)` , run code: R stops at the level of the error (if one occurs). Reset: RStudio -> Debug -> On Error -> 'Error Inspector'



## Debugging: useful features

source("projectFuncs.R")	Run complete script
traceback()	Find source of error in function sequence
options(warn=2)	Convert warnings to errors. reset: 0
browser()	Enter the function environment
n, s, f, c, Q	Navigate there. <code>print(n)</code> to display enter env. of error source
options(error=recover)	
debug(func)	run funct line by line
undebug(func)	reset

R. Peng (2002): Interactive Debugging Tools in R

D. Murdoch (2010): Debugging in R

H. Wickham (2015): Advanced R: debugging

Example: Pete Werner Blog Post (2013)

<http://r4ds.hadley.nz/functions.html>



## capturing / managing errors I

```
result <- log("2")
## Error in log("2"): non-numeric argument to
mathematical function
```

If you suspect code to fail, you can use `try`:

```
result <- try(log("2"), silent=TRUE)
result[1] # the error message
## [1] "Error in log(\"2\") : non-numeric argument
##       to mathematical function\n"
class(result)
## [1] "try-error"
ifinherits(result,"try-error")) warning("Failed with ",result)
## Warning: Failed with Error in log("2") : non-numeric
argument to mathematical function
```

```
result2 <- try(log(222), silent=TRUE)
result2 # the actual result
## [1] 5.4
```



## capturing / managing errors II

```
efun <- function(e) warning("Failed with ",  
                      sub("\n$","",e), call.=FALSE)  
res <- tryCatch(log(222), error=efun, finally=message("done"))  
## done  
res  
## [1] 5.4  
  
res <- tryCatch(log("2"), error=efun, finally=message("done"))  
## Warning: Failed with Error in log("2"): non-numeric  
argument to mathematical function  
## done  
substr(res,1,52)  
## [1] "Failed with Error in log(\"2\"): non-numeric argument "
```

with traceback stack, for errors/warnings/messages, optionally to logfile:  
`berryFunctions::tryStack`

```
myfun <- function(x, ...) hist(x, xlim=range(x), ...)  
myfun(c(4,3,NA,5), col="red")  
## Error in plot.window(xlim, ylim, "") : need finite 'xlim' values  
berryFunctions::tryStack( myfun(c(4,3,NA,5), col="red") , short=TRUE)  
## tryStack error in plot.window(xlim, ylim, "") : need finite 'xlim' values  
## -- tryStack sys.calls: berryFunctions::tryStack -> myfun -> hist -> hist.default ->  
##     plot -> plot.histogram -> plot.window -> plot.window(xlim, ylim, ...)
```



## Summary for 6.2 Debugging functions 2.0 + debugging:

- ▶ 

```
myFun <- function(arg1, arg2="default", ...)  
{out <- doStuff(arg1); plot(out, arg2=arg2, ...)}
```
- ▶ Check input, write informative messages
- ▶ `stop` / `warning` / `message`
- ▶ `deparse(substitute(x))` for input code as charstring
- ▶ lexical (not dynamic) scoping
- ▶ Debugging: `traceback`, `browser`, `options(error=recover)`
- ▶ catch errors:

```
res <- try(expr)  
ifinherits(res,"try-error")) warning("message: ",res)  
  
res <- tryCatch(expr, error=function(e) warning("message: ",e))
```

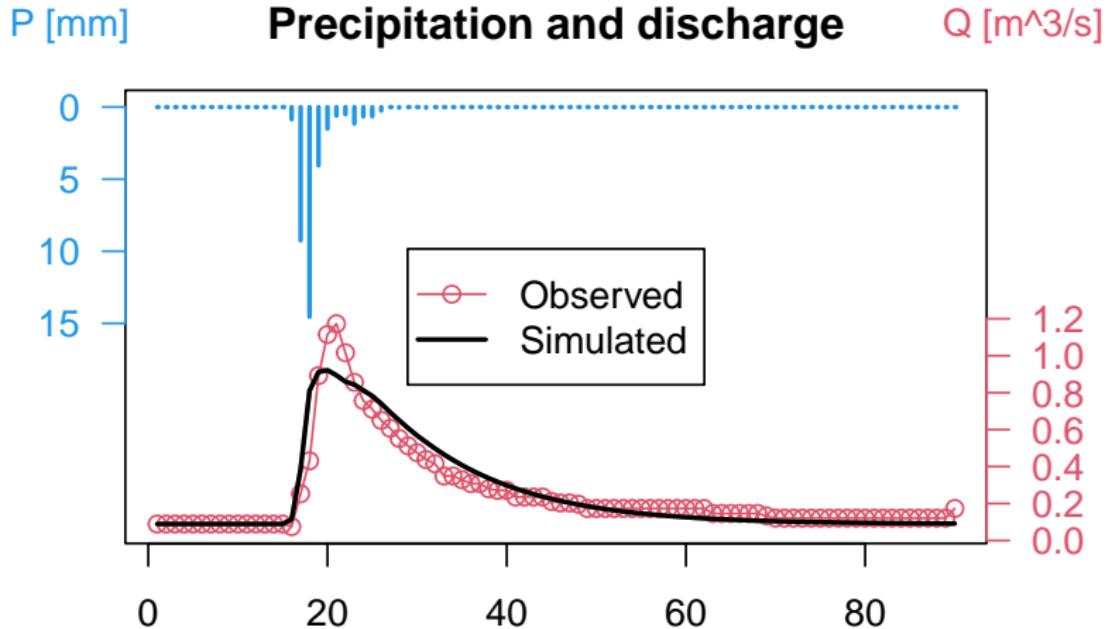
Report unclear tasks in the forum

Highlight the topics from this lesson in your RefCard



## Figure for exercises

```
berryFunctions::lsc(calib$P, calib$Q, area=1.6)
```



- 1. Intro
- 2. Basics
- 3. Objects
- 4. Data
- 5. Graphics
- 6. Flow control
  - 6.1 Conditions
  - 6.2 Debugging
  - 6.3 Loops

## for -Loops: execute code multiple times

Run code several times, each time with a different input.

Syntax: `for(runningVariable in vector){ doSomething }`

i (for index) is used often: `for(i in 1:n) doThis (i)`

*Curly braces are optional if only one command is executed.*

```
print(1:2)  
print(1:5)  
print(1:9)
```

Easier with one single line, less error prone when copypasting:

```
for(i in c(2,5,9) ) { print(1:i) }  
## [1] 1 2  
## [1] 1 2 3 4 5  
## [1] 1 2 3 4 5 6 7 8 9
```

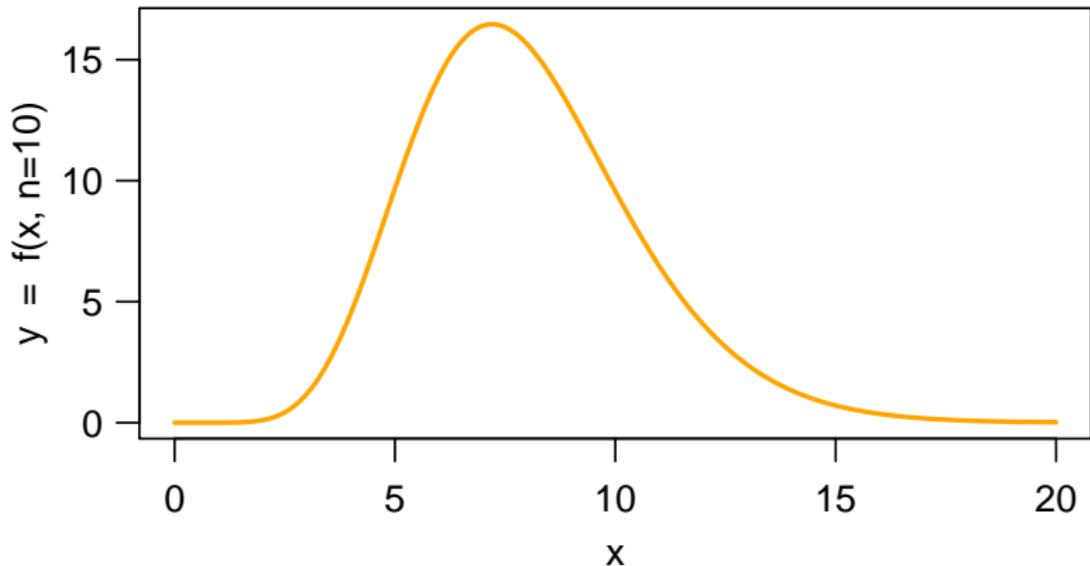
In principle, `for` does the following internally:

```
i <- 2 ; print(1:i)  
i <- 5 ; print(1:i)  
i <- 9 ; print(1:i)
```



## for -loop: several lines in a graphic I

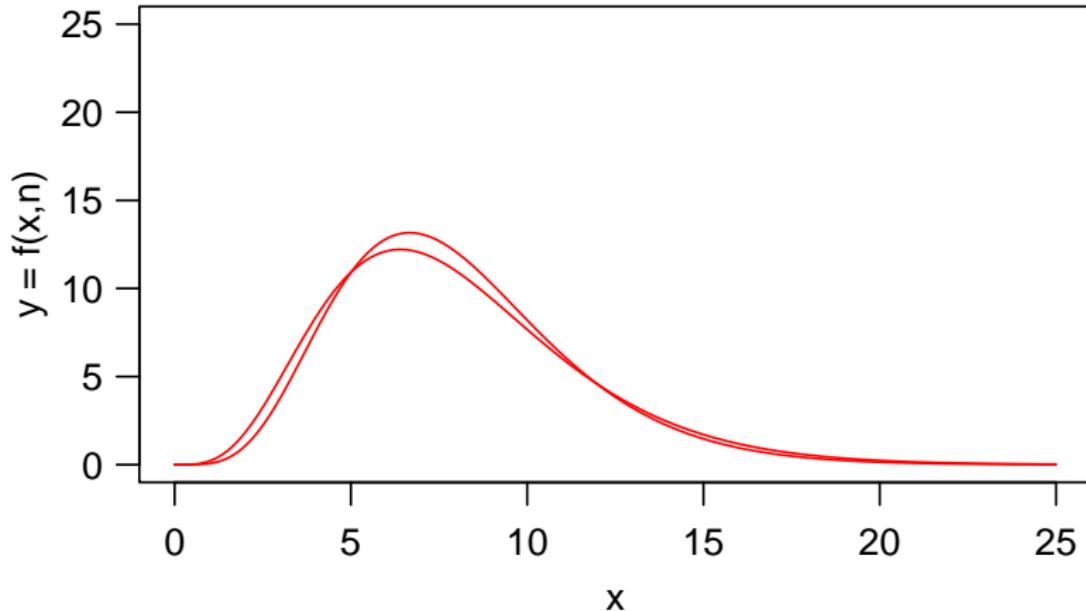
$$y = f(x, n) = \frac{12.5 \cdot n}{(n-1)!} \cdot \left(\frac{nx}{8}\right)^{(n-1)} \cdot e^{-\frac{nx}{8}}$$



In the next slide, we draw the line for several n-values (5,6,7,...,25)

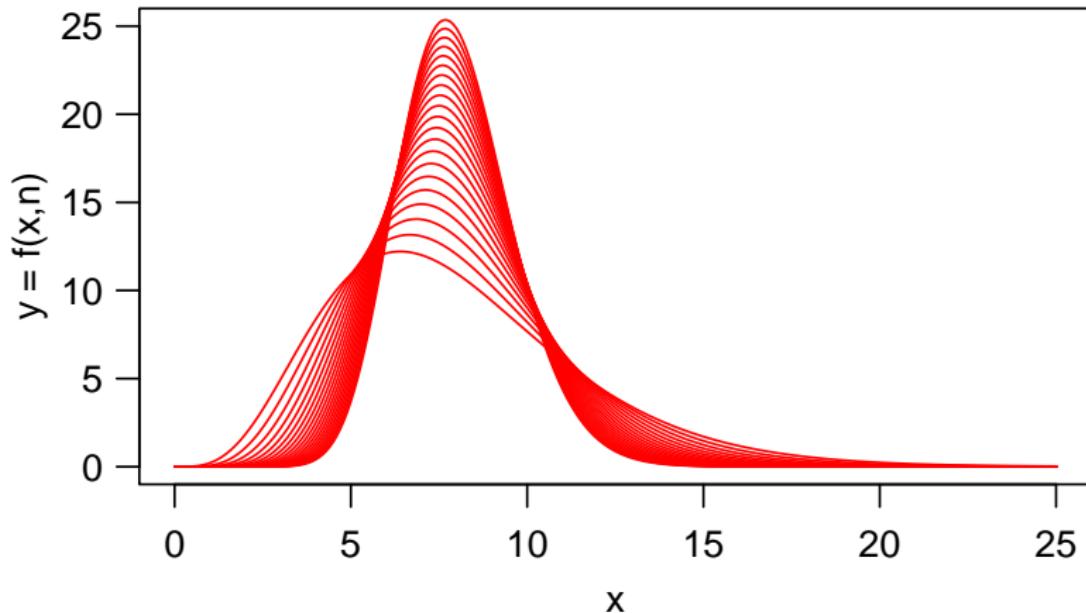
## for -loop: several lines in a graphic II

```
x <- seq(0,25,0.1)
plot(x,x, type="n", ylab="y = f(x,n)")
lines(x, 12.5*5/factorial(5-1)*(x/8*5)^(5-1)*exp(-x/8*5), col="red")
lines(x, 12.5*6/factorial(6-1)*(x/8*6)^(6-1)*exp(-x/8*6), col="red")
```



## for -loop: several lines in a graphic III

```
x <- seq(0,25,0.1)
plot(x,x, type="n", ylab="y = f(x,n)")
for (n in 5:25)
lines(x, 12.5*n/factorial(n-1)*(x/8*n)^(n-1)*exp(-x/8*n), col="red")
```



## for -loop: fill a vector

```
v <- vector(mode="numeric", length=20) ; v
## [1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
for(i in 3:17) { v[i] <- (i+2)^2 } # What is now in v?

v # the code above was executed for each i
## [1] 0 0 25 36 49 64 81 100 121
## [10] 144 169 196 225 256 289 324 361 0
## [19] 0 0
```

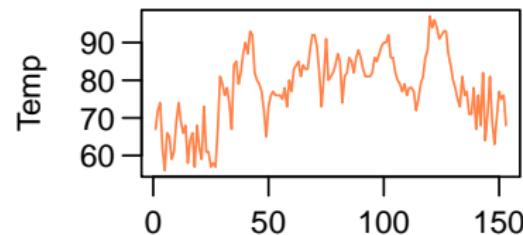
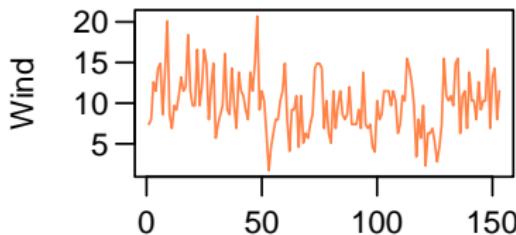
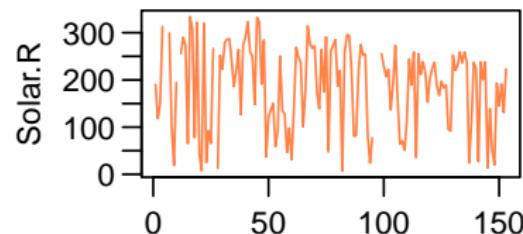
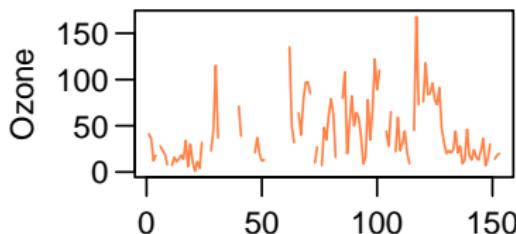
Which objects now exist in the global environment workspace?

-> `v` (as above) and `i` (with the last value, 17)

## for -loop: multiple graphics

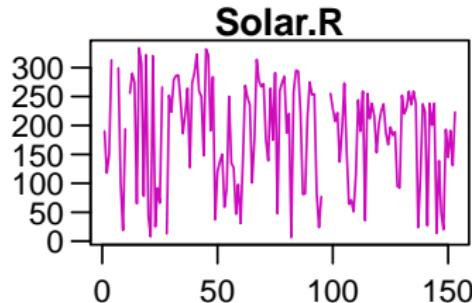
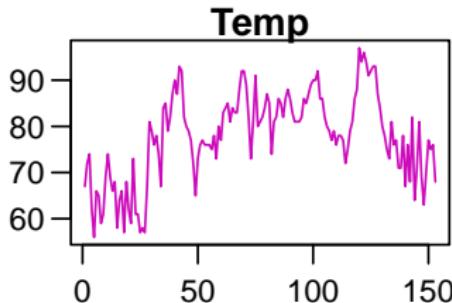
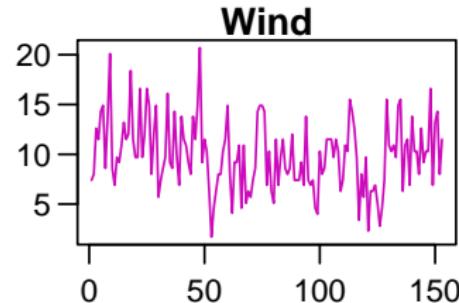
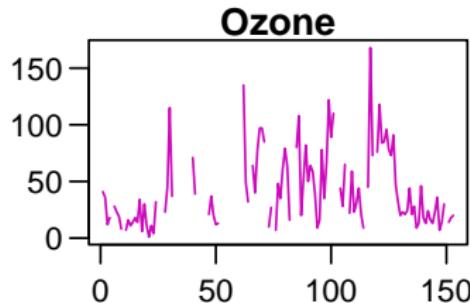
```
par(mfrow=c(2,2), mar=c(2,4,1,1), mgp=c(2.5,0.7,0), oma=c(0,0,2,0), las=1)
for(i in 1:4) plot(airquality[,i], type="l",
                     ylab=colnames(airquality)[i], col="sienna1")
mtext("Air quality in New York", line=0, outer=TRUE)
```

Air quality in New York



for -loop: the running variable can also be a character string

```
par(mfrow=c(2,2), mar=c(2,4,1,1), mgp=c(2.5,0.7,0), las=1)
for(cn in c("Ozone", "Wind", "Temp", "Solar.R"))
  plot(airquality[,cn], type="l", main=cn, ylab="", col=6)
```



## Avoid `for`-loops: vectorizing

- ▶ `for` loops are slow in R (as in any interpreted language)
- ▶ Great for graphics, rather avoid otherwise
- ▶ The best solution is vectorizing, but this does not always work
- ▶ Alternatively, use `lapply` (short code, easy to parallelize)

```
v <- vector(mode="numeric", length=20)
for(i in 3:17) { v[i] <- (i+2)^2 }
```

`v[3:17] <- (3:17+2)^2 # + and ^ are vector-capable`

Real example from my consulting:

```
for(i in c(1:nrow(DF))){  
  if(DF[i, "column1"] != "valid"){ # Where column1 is not  
    DF[i, "column2"] <- NA          # 'valid', set  
  }                                # column2 to NA  
}  
DF$column2[DF$column1 != "valid"] <- NA
```



## Avoid `for`-loops: `lapply`

```
# All CSV file names in the rawdata folder:  
files <- dir("rawdata/", pattern="*.csv", full.names=TRUE)  
  
# Read all files with a for loop:  
ldfs <- list() # create (initiate) empty list  
for(i in 1:length(files))  
    ldfs[[i]] <- read.csv(files[i], header=TRUE)  
  
# A lot nicer (although not faster computationally):  
ldfs <- lapply(X=files, FUN=read.csv, header=TRUE)  
  
# Progress bar with remaining / required time  
library("pbapply")  
ldfs <- pblapply(X=files, FUN=read.csv, header=TRUE)  
  
# Parallelized (use all CPU cores) - On Unix:  
ldfs <- pblapply(X=files, FUN=read.csv, header=TRUE, cl=8)  
# More complicated on Windows, use this for simple things:  
berryFunctions:::par_sapply(files, read.csv, simplify=FALSE, cl=8)
```

More ingenious stuff with `lapply`-loops and parallelization in the supplementary material.



## while loop: Repeat code as long as necessary

If it is unknown beforehand, how many iterations will be necessary:

Syntax: `while(condition){ doSomething }`

For example, if a folder name could be given with trailing slashes:

```
folder <- "path/to/project///"  
while(endsWith(folder, "/"))  
{  
  folder <- sub("/$", "", folder)  
}  
folder  
## [1] "path/to/project"
```

To break out of an infinite loop (if the condition is accidentally formulated so that it is never FALSE), press **ESC** while the mouse pointer is in the console, or click the **STOP** key

For this example there is also a solution without a loop:

```
folder <- sub("/+$", "", folder) ; "[/|\\"+$" # for / and \
```



## Real-life application of a `while` loop

This is how many coin flips it takes to get 'heads' 3 times in a row

```
set.seed(123) # Starting point for random number generator
sample(0:1, 9,T) # randomly draw 0 (tail) or 1 (heads)
## [1] 0 0 0 1 0 1 1 1 0
```

```
flips <- 0
heads <- 0 # Number of consecutive 'head' flips
set.seed(123) # Get the same "random" numbers every time
while(heads < 3) {
  if( sample(0:1,1) ) heads <- heads + 1 else heads <- 0
  flips <- flips + 1
}
flips
## [1] 8
```

Exemplary usage of `while` in `base R` and `packagePath`.



## Two special commands (also in `for` loops)

`next` : Jump to the next iteration

`break` : Exit the loop completely (cancel, abort)

```
i <- 0
while(i<5)
{
  i <- i + 1
  cat("i:",i,"\n")
}
## i: 1
## i: 2
## i: 3
## i: 4
## i: 5
```

```
i <- 0
while(i<5)
{
  i <- i + 1
  if(i==3) next
  cat("i:",i,"\n")
}
## i: 1
## i: 2
## i: 4
## i: 5
```

```
i <- 0
while(i<5)
{
  i <- i + 1
  if(i==3) break
  cat("i:",i,"\n")
}
## i: 1
## i: 2
```



### repeated execution of code:

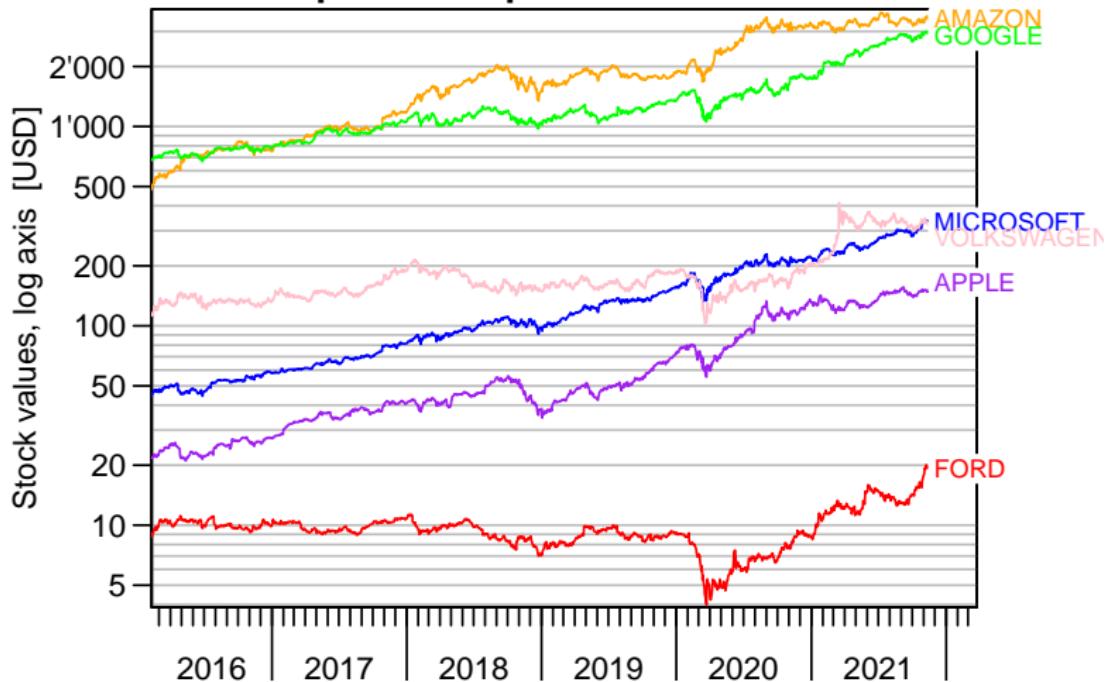
- ▶ `for(i in vec) doSomethingWith(i)`
- ▶ `for(c in charvec) {doSomethingWith(c)}`
- ▶ Vectorize, if possible
- ▶ Use `lapply` for elegant, extensible code
- ▶ Use `while` for loops with an initially unknown number of calls
- ▶ `next`, `break`

Report unclear tasks in the forum

Highlight the topics from this lesson in your RefCard



## Tech companies outperform car manufacturers



`seq_along(n)` is safer than `1:n` in `for` loops

```
doThis <- function(x) if(x<1) stop("x false: ",x) else x
thing <- 1:6
```

`for(i in 1:n)` is dangerous code:

```
for(i in 1:length(thing)) doThis(i)
```

Because here the same code fails:

```
thing <- which(letters=="4")
for(i in 1:length(thing)) doThis(i)
## Error in doThis(i): x false: 0
```

This is safer:

```
for(i in seq_along(thing)) doThis(i)
```

because:

```
1:length(thing)
## [1] 1 0
```

```
seq_along(thing)
## integer(0)
```



repeat : unconditional loop - endless - needs break

while(TRUE) {commands}

is possible with

repeat {commands}

```
i <- 0          # Initialize object mit 0
repeat          # Repeat the following block endlessly
{
  i <- i + 1      # overwrite i with new number
  if(i < 3) print(i) # Conditional output in the console
  if(i == 3) break  # Conditional termination of the loop
}
## [1] 1
## [1] 2
```

What is now the value of i ?

3, because in the last iteration i = 2 and +1 was added.



## for -> lapply: more ingenious things

```
files <- dir(pattern="*.csv")
ldfs <- lapply(files, read.csv)

names(ldfs) <- files
ldfs[[2]] # Content of the second file
str(ldfs, max.level=1) # str of only the top level
sapply(ldfs, nrow) # Number of rows per df
sapply(ldfs, "[", , j=3) # third column of each df
sapply(ldfs, "[", 5, ) # fifth line of dfs in each case
df <- do.call(rbind, ldfs) # if all dfs have n columns

# To read in many large files quickly:
library("data.table") # fread + rbindlist
ldfs <- lapply(X=files, FUN=fread, sep=",")
df <- rbindlist(ldfs)
```



## Run code in parallel on multiple CPU cores

```
library("parallel") # Comes with R, install.packages not needed
nc <- detectCores()-1 # Leave one core open to browse xkcd :
n_runs <- 1e6
r_mean <- function(x) mean(rnorm(n_runs))
m1 <- pblapply(X=1:200, FUN=r_mean) # 38 Sec on 1 CPU core

m2 <- pblapply(X=1:200, FUN=r_mean, cl=nc) # 9 Sec on 7 cores

# Windows needs more code than Unix:
cl <- makeCluster(nc)
clusterExport(cl, c("n_runs")) # objects needed in FUN
clusterEvalQ(cl, library("pack")) # Packages used in FUN
m3 <- pblapply(X=1:200, FUN=r_mean, cl=cl)
stopCluster(cl) ; rm(cl) ; gc()

berryFunctions::parallelCode() # Template for Windows
berryFunctions::par_sapply(X, FUN) # For simple things
```

Details e.g. at [stat701](#)

`pbapply::pblapply` uses `mclapply` on Unix, `parLapply` in Windows.

`clusterEvalQ` can be avoided: use `pack::fun()` instead of `fun()`.

