ESTP Course: The Use of R in Official Statistics

Alexander Kowarik, Bernhard Meindl
Vienna, April 2017

The Contractor is acting under a FWC concluded with the Commission.

ESTP Course: The Use of R in Official Statistics (Programme)

Alexander Kowarik, Bernhard Meindl
April 2017

Day 1 (4/4/2017)

Forenoon 9:30 - 12:30 (Coffee break approx. 11:00)

- Introduction and first steps with R.
- Help and information on the web.
- Working with Rstudio

Afternoon 14:00 - 17:00 (Coffee break approx. 15:30)

- Data types and basic syntax
- Vectorisation

Day 2 (5/4/2017)

Forenoon 9:30 - 12:30 (Coffee break approx. 11:00)

- Control structures
- Functions

Afternoon 14:00 - 17:00 (Coffee break approx. 15:30)

- Object orientation
- Basic data manipulation
- Data import-export

Day 3 (6/4/2017)

Forenoon 9:00 - 12:00 (Coffee break approx. 10:30)

- Graphics with R
- Packages `ggplot2` (and `ggvis`)

Afternoon 13:00 - 16:00 (Coffee break approx. 14:30)

- Dynamical reports using R markdown
- Advanced data manipulation (using `dplyr`)

Day 4 (7/4/2017)

Forenoon 9:00 - 12:00 (Coffee break approx. 10:30)

- Statistics with R
- Useful R-Packages for official statistics

Afternoon 13:00 - 16:00 (Coffee break approx. 14:30)

- Questions and Exercises
- Evaluation

Introduction to R

Alexander Kowarik, Bernhard Meindl

Aim

To gain knowledge in the basics of a modern *high-level* object-oriented statistical software **environment**:

- data manipulation
- data visualisation
- basics in object-orientation
- implementing new functionalities
- to see interesting applications with R

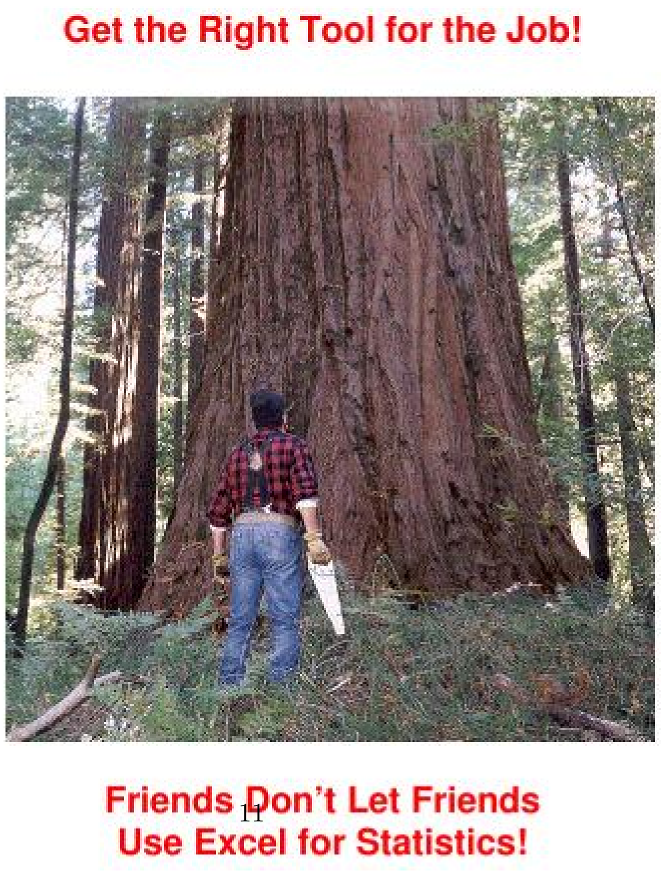
Statistical Computing

… is different to "Mathematical Computing'', because

- the use of different **data types** (more than *numeric*, *character*, and data in *rectangular form*)
- a lot of statistical methods are already available and are ready to be used
- the aim is to play with data in an **object-oriented programming language** (no *batch-mode* during the development of code).

no Excel!

“Don't do statistics in spreadsheets, especially Excel."


Get the Right Tool for the Job!
Friends Don’t Let Friends
Use Excel for Statistics!

Differences to other Software, e.g. SAS

Example: Calculate the rounded mean of the variable *hp* times 2.54 available in data set *mtcars*

In **R**:

```
data(mtcars)
round( 2.54 * mean( mtcars$hp ) )
```

```
[1] 373
```

Differences to other Software, e.g. SAS

In SAS:

```
DATA mtcars;
  set old;
  hp = hp * 2.54;
PROC means;
  var hp;
  output out=new2 mean=hp;
DATA new2;
  set new2;
  hp=round(hp);
PROC fsview;
run;
```

R

- Founder: Ross Ihaka and Robert Gentlemen 1995
- R (and SPLUS) is based on S
- S is a programming language, developed by John Chambers (Bell Laboratories). Bell Labs developed also Unix and C.
- Since 1997 international development
- Distributed from Vienna (R: **http://www.r-project.org**, resources: **http://cran.r-project.org**
- R is nowadays the most popular and most used software in the statistical world. It is also developed and used by major companies like Google, Pfister, Revolutions, .

R is free und open-source

- no licence costs (freeware)
- allowed to copy and reu-use code (free software)
- source code is available and can be modified (open source)

But:

- respect intellectual rights! (GPL-2)

R as a Mediator?

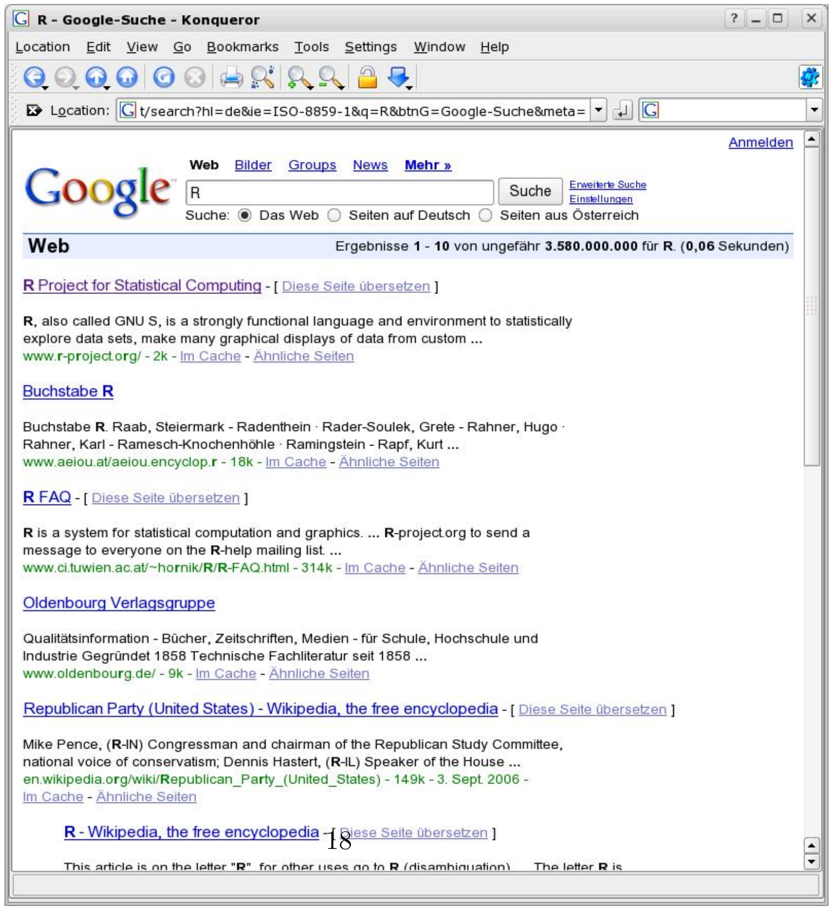
- Environment for interactive computing with data
- Users are also Developers
- high-quality graphics
- Exchange of code with others is easy. Modern methods are often exclusively developed in **R**
- Object-oriented programming
- Interfaces to many other software products (data exchange but also C, C++,Fortran, Java,... interfaces)
- Interfaces to multiple (relational) databases

Why not Java or C++?

- Interactive development and communication with data (avoid *batch files*) to write and execute progamms
- In that sense: **R** is similar to Matlab, Perl, Python, Ruby or Basic
- If calculation time plays a role:
  - **R** provides direct interfaces to C, C++, Java and many other languages.

Information about R

The CRAN Team was very proud of



R - Google-Suche - Konqueror

Location: t/search?hl=de&ie=ISO-8859-1&q=R&btnG=Google-Suche&meta=

Suche: (● Das Web) ○ Seiten auf Deutsch ○ Seiten aus Österreich

Web Ergebnisse **1** - **10** von ungefähr **3.580.000.000** für **R**. (**0,06** Sekunden)

R Project for Statistical Computing - [Diese Seite übersetzen]

R, also called GNU S, is a strongly functional language and environment to statistically explore data sets, make many graphical displays of data from custom ...
www.**r**-project.org/ - 2k - Im Cache - Ähnliche Seiten

Buchstabe **R**

Buchstabe **R**. Raab, Steiermark - Radenthein · Rader-Soulek, Grete - Rahner, Hugo · Rahner, Karl - Ramesch-Knochenhöhle · Ramingstein - Rapf, Kurt ...
www.aeiou.at/aeiou.encyclop.**r** - 18k - Im Cache - Ähnliche Seiten

R FAQ - [Diese Seite übersetzen]

R is a system for statistical computation and graphics. ... **R**-project.org to send a message to everyone on the **R**-help mailing list. ...
www.ci.tuwien.ac.at/~hornik/**R**/**R**-FAQ.html - 314k - Im Cache - Ähnliche Seiten

Oldenbourg Verlagsgruppe

Qualitätsinformation - Bücher, Zeitschriften, Medien - für Schule, Hochschule und Industrie Gegründet 1858 Technische Fachliteratur seit 1858 ...
www.oldenbourg.de/ - 9k - Im Cache - Ähnliche Seiten

Republican Party (United States) - Wikipedia, the free encyclopedia - [Diese Seite übersetzen]

Mike Pence, (**R**-IN) Congressman and chairman of the Republican Study Committee, national voice of conservatism; Dennis Hastert, (**R**-IL) Speaker of the House ...
en.wikipedia.org/wiki/**R**epublican_Party_(United_States) - 149k - 3. Sept. 2006 - Im Cache - Ähnliche Seiten

R - Wikipedia, the free encyclopedia - [Diese Seite übersetzen]

This article is on the letter "**R**", for other uses go to **R** (disambiguation). ... The letter **R** is

Information about R

- Homepage:
  - **http://www.r-project.org**
  - **http://cran.r-project.org**
- frequently asked questions (FAQ) lists on CRAN
- no need for buying a book (e.g. **http://adv-r.had.co.nz**)
- manuals und contributed manuals
- Task-views on CRAN
- help.start()
- Short code glossary on CRAN
- R-bloggers (**www.r-bloggers.com**)
- Quick-R (**www.statmethods.net**)

Let's have a look at the Word Wide Web

Books about R - the Springer series

- some Books

THE ART OF R PROGRAMMING
A TOUR OF STATISTICAL SOFTWARE DESIGN
NORMAN MATLOFF
R IN ACTION
Use R!
Alain F. Zuur
Elena N. Ieno
Erik H.W.G. Meesters
A Beginner's Guide to R
Using R for Data Management, Statistical Analysis, and Graphics
Nicholas J. Horton
Statistics and Computing
Robert A. Muenchen
R for SAS and SPSS Users
Second Edition
Springer
Clemens Reimann · Peter Filzmoser · Robert Garrett · Rudolf Dutter
Statistical Data Analysis Explained
Applied Environmental Statistics with R
WILEY
Springer Texts in Statistics
Simon J. Sheather
A Modern Approach to Regression with R
Springer
Software for Data Analysis
Programming with R
Springer
Elegant Graphics for Data Analysis
Springer
Paul Murrell

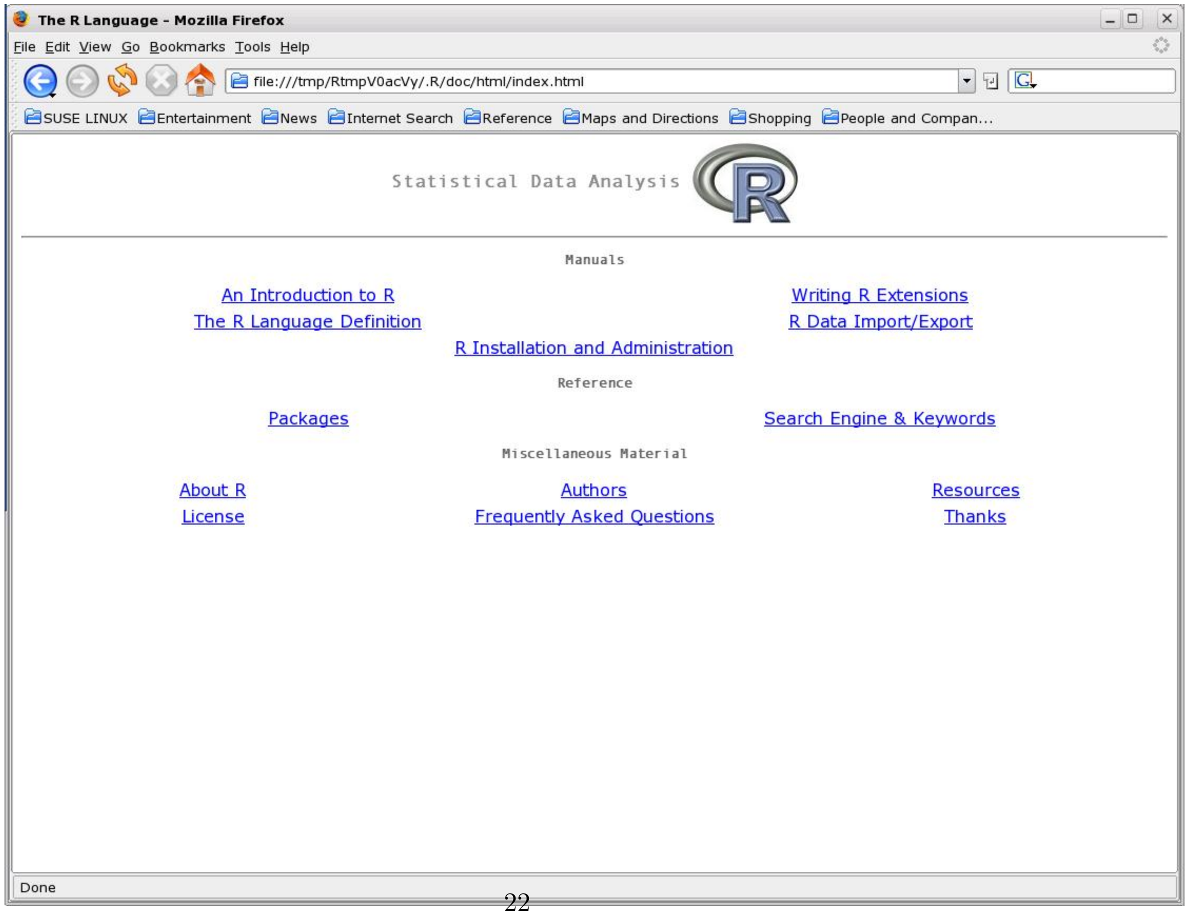
R as a simple calculator

Use R also as your daily calculator, some useful functions are

- +, -, *, / … addition | subtraction | multiplication | division …
- **exp()** … exponential function
- **log()** … logarithm (default: Base *e*)
- **sqrt()** … square root
- **sin()** … sinus-function
- **cos()** … cosinus-function
- **tan()** … tangens-function

help.start()

- the help-browser


The R Language - Mozilla Firefox

File Edit View Go Bookmarks Tools Help

file:///tmp/RtmpV0acVy/.R/doc/html/index.html

SUSE LINUX Entertainment News Internet Search Reference Maps and Directions Shopping People and Compan...

Statistical Data Analysis

Manuals

An Introduction to R | Writing R Extensions
The R Language Definition | R Data Import/Export
R Installation and Administration

Reference

Packages | Search Engine & Keywords

Miscellaneous Material

About R | Authors | Resources
License | Frequently Asked Questions | Thanks

Done

Libraries and Packages

- R comes with approx. 7000 *add-on* packages
- path of the installed packages:

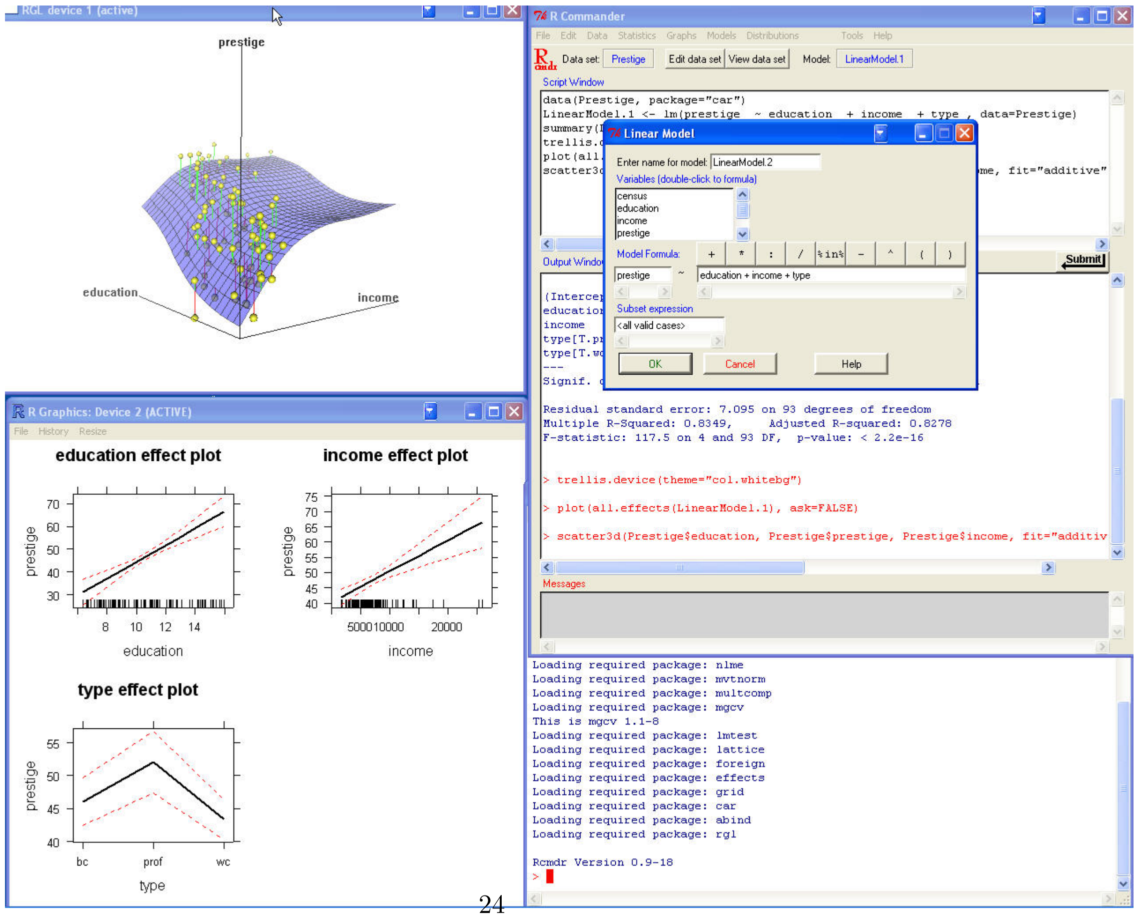
```
.libPaths()
```

```
[1] "/data/home/meindl/R/x86_64-suse-linux-gnu-library/3.3"
[2] "/usr/lib64/R/library"
```

- The basic installation includes the most important packages.
- Additional packages can be easy installed by, e.g., install.packages()

Point-and-Click Graphical User Interfaces

- A well-known GUI that allows reproducibility: the R Commander, **http://goo.gl/YLrOX**



```
data(Prestige, package="car")
LinearModel.1 <- lm(prestige  ~ education  + income  + type , data=Prestige)
summary(I
trellis.d
plot(all.
scatter3d                                                   me, fit="additive"
```

```
(Intercep
educatior
income
type[T.pr
type[T.wc
---
Signif. c

Residual standard error: 7.095 on 93 degrees of freedom
Multiple R-Squared: 0.8349,     Adjusted R-squared: 0.8278
F-statistic: 117.5 on 4 and 93 DF,  p-value: < 2.2e-16


> trellis.device(theme="col.whitebg")

> plot(all.effects(LinearModel.1), ask=FALSE)

> scatter3d(Prestige$education, Prestige$prestige, Prestige$income, fit="additiv
```

```
Loading required package: nlme
Loading required package: mvtnorm
Loading required package: multcomp
Loading required package: mgcv
This is mgcv 1.1-8
Loading required package: lmtest
Loading required package: lattice
Loading required package: foreign
Loading required package: effects
Loading required package: grid
Loading required package: car
Loading required package: abind
Loading required package: rgl

Rcmdr Version 0.9-18
> 
```

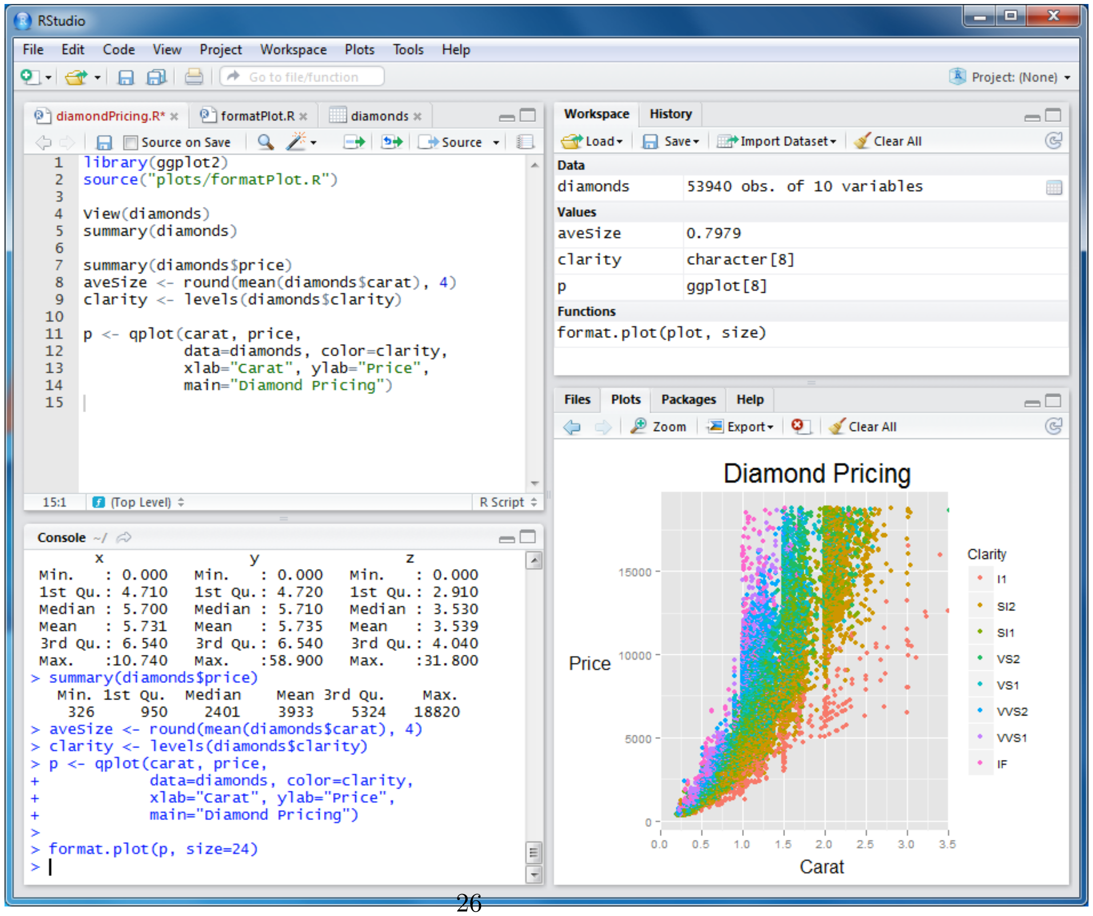
Script Editors and Programming Environments

Basic “rules" for working with R:

- write the code in a well-developed editor and communicate interactively with R
- the editor should allow syntax-highlighting, code-completion and interactive communication with R
- the editor should include tool for other useful software (github, svn, C++, Java, ...)
- source of R-functions with source('functionName.R') or even better, build packages!

Script Editors - RStudio

- **http://www.rstudio.org/**



```r
library(ggplot2)
source("plots/formatPlot.R")

View(diamonds)
summary(diamonds)

summary(diamonds$price)
aveSize <- round(mean(diamonds$carat), 4)
clarity <- levels(diamonds$clarity)

p <- qplot(carat, price,
           data=diamonds, color=clarity,
           xlab="Carat", ylab="Price",
           main="Diamond Pricing")

```

```
       x                y                z
 Min.   : 0.000   Min.   : 0.000   Min.   : 0.000
 1st Qu.: 4.710   1st Qu.: 4.720   1st Qu.: 2.910
 Median : 5.700   Median : 5.710   Median : 3.530
 Mean   : 5.731   Mean   : 5.735   Mean   : 3.539
 3rd Qu.: 6.540   3rd Qu.: 6.540   3rd Qu.: 4.040
 Max.   :10.740   Max.   :58.900   Max.   :31.800
> summary(diamonds$price)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
    326     950    2401    3933    5324   18820
> aveSize <- round(mean(diamonds$carat), 4)
> clarity <- levels(diamonds$clarity)
> p <- qplot(carat, price,
+            data=diamonds, color=clarity,
+            xlab="Carat", ylab="Price",
+            main="Diamond Pricing")
>
> format.plot(p, size=24)
> 
```

Data	
diamonds	53940 obs. of 10 variables
Values	
aveSize	0.7979
clarity	character[8]
p	ggplot[8]
Functions	
format.plot(plot, size)	

R-Studio - advantages

- designed for R
- working with project philosophy
- script editor communicate with R
- objects in the workspace are listed
- version control systems (svn, git) are supported
- dynamical reports supported
- many add-ons (eg Rmarkdown, C++ code, ggplot2, …)

Exercise 1

- Open RStudio
- In the script-frame assign **x <- 1**
- Send this line of code to the **R** console (left lower area in RStudio). DO NOT *copy/paste.*
- Look at your workspace (upper right frame in RStudio)
- type **x + x** in your console
- type **x + x** in your script-frame and send it to R (console)
- calculate the logarithm of 12

Excercise 2

- Create a new project within RStudio
- Save a (new) R-file in this project
- Create another project
- Switch between projects

Overview

- additional basics
- classes (and a bit of object-orientation)
- import/export facilites
- data manipulation
- graphics in R (graphics, ggplot2)
- dynamical reports
- statistics with R

Remark: **One can learn software only by actually using it**

–> We will do a lot of excercises

Data types in R

Alexander Kowarik, Bernhard Meindl

Eurostat

STATISTIK AUSTRIA

Die Informationsmanager

Overview / Objectives

- Get to know the most important data types
  - Numeric, character, logical
- to learn about the following data structures:
  - vectors / factors
  - matrices
  - lists
  - data frames
- indexing
- Special data types:
  - missing values, NULL-objects, NaN, - / + Inf

Vectors (1)

- Vectors are the simplest data structure in R
- = Vectors sequence of elements of the same type
  - numerical vectors
  - character vectors
  - logical vectors
- Query to a data type means for example:
  - is.numeric()
  - is.character()
  - is.logical()

Vectors (2)

- Vectors are often created with the function c()
- Creating a numeric vector

```
v.num <- c(1,3,5.9,7); v.num
```

```
[1] 1.0 3.0 5.9 7.0
```

```
is.numeric (v.num)
```

```
[1] TRUE
```

- Create a character vector

```
v.char <- c("one", "two", "three"); v.char
```

```
[1] "one"   "two"   "three"
```

```
is.character (v.char)
```

```
[1] TRUE
```

Vectors (3)

- logical vectors are often created indirectly from numerical / character vectors

```
v.log1 <- v.num > 3; v.log1
```

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

```
v.log2 <- v.char == "two"; v.log2
```

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

- Logical vectors can also be produced directly

```
v.log3 <- c(TRUE, FALSE, FALSE, TRUE); v.log3
```

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

Vectors (4)

- Warning many operations on vectors are performed element-wise
  - e.g. logical comparisons
  - arithmetic operations with vectors

```
v1 <- c(1,2,3)
v2 <- c(4,5,6)
v1 + v2
```

```
[1] 5 7 9
```

- common error source: if the length of two vectors does not match, the shorter one is repeated (*recycling*)

```
v1 <- c(1,2,3)
v2 <- c(4,5)
v1 + v2
```

```
[1] 5 7 7
```

Vectors (5)

- Vectors can store only elements of the same data type
- coercion : by specifying different types of encoding, R internally coerce to meaningful data types automatically

```
v1 <- c(100, TRUE, 20, FALSE); v1 # logical values are conv. to 0/1
```

```
[1] 100   1  20   0
```

```
is.numeric (v1)
```

```
[1] TRUE
```

```
v2 <- c(100, TRUE, "A", FALSE); v2 # 'lowest' common data type is string
```

```
[1] "100"   "TRUE"  "A"     "FALSE"
```

```
is.numeric (v2)
```

```
[1] FALSE
```

Vectors (6)

c() is the constructor for new vectors

```
v1 <- c(1,2,3); v2 <- c(4,5,6); v3 <- c(7,8,9)
v.gesamt <- c (v1, v2, v3); v.gesamt
```

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

- With length() you get the number of elements of a vector

```
length (v.gesamt)
```

```
[1] 9
```

- unique() reports all the unique elements of a vector

```
unique(c(1,1,1,2,2,2,3,3,3))
```

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

Sequences (1)

- Creating a vector by typing its values is boring
- We can use operators and functions to generate vectors
  - **:** number of vectors with numbers at intervals of 1
  - **seq()**: generalization of **:**
  - **rep()**: function to repeat vectors

Sequences (2)

- The **:** operator generates a sequence of numbers at intervals of 1
  - forward

```
forward <- 1:5; forward
```

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

- backwards

```
backward <- 5:1; backward
```

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

- non-integer starting value

```
decimal <- 2.5:10; decimal
```

```
[1] 2.5 3.5 4.5 5.5 6.5 7.5 8.5 9.5
```

Sequences (3)

- Function **seq()** can be used to produce sequences in a general way
- important function arguments
  - **from**: seed
  - **to**: maximal final value
  - **by**: increment
  - **length**: desired number of elements

```
seq1 <- seq(from = 1, to = 10, by = 2); seq1
```

```
[1] 1 3 5 7 9
```

```
seq2 <- seq(from = 1, to = 5, length = 10); seq2
```

```
 [1] 1.000000 1.444444 1.888889 2.333333 2.777778 3.222222 3.666667
 [8] 4.111111 4.555556 5.000000
```

Tasks / Exercises

Time for practical training! :)

Please continue to work on Exercises 1x).

Repetition (1)

- Using the **rep()** function, vectors can be repeated
- Important function arguments
  - **x**: vector to be repeated
  - **times**: Number of repetitions
  - **each**: How often will every single element of *x* be repeated

```
rep1 <- rep(1:5, times=2); rep1
```

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

```
rep2 <- rep(1:5, each=2); rep2
```

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

Repetition (2)

- **rep()** can be used to repeat vectors of arbitrary data type
  - Character vector

```
rep(c("one", "two", "three"), each=3)
```

```
[1] "one"   "one"   "one"   "two"   "two"   "two"   "three" "three" "three"
```

- Logical vector

```
rep(c(TRUE, FALSE, TRUE), times=2)
```

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

- The argument x can be a vector again

```
rep(c("one","two","three"), times=c(3,5,2))
```

```
 [1] "one"   "one"   "one"   "two"   "two"   "two"   "two"   "two"  
 [9] "three" "three"
```

Access to vectors / Indexing (1)

- Often it is necessary to subset vectors
- The selection is made using the [] operator
- Selection can be done in 3 different ways
    - **positive**: a vector of positive integers that specifies the position of the desired elements
    - **negative**: a vector with negative integers indicating the position of the non-required elements
    - **logical**: a logic vector in which the elements are to be the selected (**TRUE**), and those who are not selected (**FALSE**).

Access to vectors / Indexing (2)

- positive indexing

```
v <- 1:10; v
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

```
v[3] # the third element
```

```
[1] 3
```

```
v[c(1, length(v))] # the first and last element
```

```
[1]  1 10
```

```
v[seq(from=1, to=length(v), by=2)] # all odd indices
```

```
[1] 1 3 5 7 9
```

Access to vectors / Indexing (3)

- negative indexing

```r
v <- 1:10; v
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

```r
v[-c(1,3)] # everything but the first and third element
```

```
[1]  2  4  5  6  7  8  9 10
```

```r
v[-c(2:8)] # all except the elements at position two to eight
```

```
[1]  1  9 10
```

```r
v[-seq(from=1, to=length(v), by=2)] # only values at even index positions
```

```
[1]  2  4  6  8 10
```

Access to vectors / Indexing (4)

- logical indexing

```
v <- 1:10; v
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

```
y <- rep (c(TRUE, FALSE), length=10); y
```

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

```
v[y] # v at odd positions
```

```
[1] 1 3 5 7 9
```

Access to vectors / Indexing (5)

- the ! operator (*not*) switches the values of a logical vector
- you can make use of it in indexing

```
y; !y # TRUE FALSE and vice versa
```

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

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

- *v* at odd positions

```
v[y]
```

```
[1] 1 3 5 7 9
```

- *v* at the even positions

```
v[!y]
```

```
[1]  2  4  6  8 10
```

Access to vectors / Indexing (6)

- a logical expression can be written directly in []
- additional logical operators can be used
  - &: logical *and*
  - |: logical *or*
  - !: *negation*
- & and | compare (element-wise) two or more logical expressions
  - & is **TRUE**: all elements compared are **TRUE**
  - | is **TRUE**: at least one element is **TRUE**

Access to vectors / Indexing (7)

- logical conjunction

```
x <- 1:10;
log1 <- x < 5; log1
```

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

```
log2 <- x >= 3; log2
```

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

- logical *and* (&) and logical *or* (|)

```
x[log1 & log2]
```

```
[1] 3 4
```

```
x[log1 | log2]
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

Tasks / Exercises

Time for practical training! :)

Please continue to work on Exercises 2x), 3x) and 4x).

Matrices (1)

- a matrix is a generalization of a vector
- matrices can have only elements of the same data type
- so there are:
    - numerical matrices
    - character matrices
    - logical matrices
- a matrix can be generated with the **matrix()** function/constructor

Constructing matrices (1)

- important function arguments from **matrix()**
  - *data*: a data vector
  - *nrow*: Number of lines
  - *ncol*: Number of columns
  - *byrow*: the matrix should be standard in columns (default) or line by line, filled

```
matrix(data = 1:8, nrow=2, ncol=4) # filled columns-wise
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8
```

```
matrix(data = 1:8, nrow=2, ncol=4, byrow=TRUE) # line by line
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
```

Constructing matrices (2)

- a matrix can also be created by setting the attribute *dim* to a vector
- the first element is the number of rows, the second item to the number of columns

```
x <- 1:8; is.matrix(x)
```

```
[1] FALSE
```

```
dim(x) <- c(2,4); x
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8
```

```
is.matrix(x)
```

```
[1] TRUE
```

Constructing matrices (3)

- a matrix of vectors can be created by row / column-wise concatenation
- line by line with **rbind()**. Column by column with **cbind()**

```
rbind(1:4,5:8); cbind(1:2,3:4,5:6)
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
```

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

- Be careful on *recycling*, the shorter vector is repeated

```
rbind(1:4, 1:3)
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    1    2    3    1
```

Constructing matrices (4)

- with **rbind()** and **cbind()** matrices or vectors or matrices and vectors can be combined

```
m1 <- matrix(1:8, nrow=2); m1
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8
```

```
m2 <- matrix(9:10, nrow=2); m2
```

```
     [,1]
[1,]    9
[2,]   10
```

```
cbind(m1, m2, c(-1,-1))
```

```
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    3    5    7    9   -1
[2,]    2    4    6    8   10   -1
```

Indexing of matrices (1)

Individual elements can be accessed with the [] -operator

- Syntax: [*indexRows*, *indexColums*]
- Indexing for the row index (*indexRows*) and for the column index (*indexColums*) is analogous to the indexing of a vector
    - positive indexing
    - negative indexing
    - logical indexing
- the type of indexing may be different for row index and column index
- is an index empty, all elements are selected

Indexing of matrices (2)

- Matrix indexing

```
mat <- matrix(1:8, nrow=2, byrow=TRUE); mat
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
```

```
mat[2, 3] # directly, second line, third column
```

```
[1] 7
```

```
mat[-2, 3] # mixed negative and positive indexing
```

```
[1] 3
```

```
mat[-2,] # empty column index
```

```
[1] 1 2 3 4
```

Indexing of matrices (3)

- more matrix indexing

```
mat <- matrix(1:8, nrow=2, byrow=TRUE); mat
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
```

```
mat[c(TRUE, FALSE),] # first line with logical indexing the rows
```

```
[1] 1 2 3 4
```

```
mat[c(TRUE, FALSE),, drop = FALSE] # as above, returns a matrix!
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
```

```
mat[2, mat[1,]>2]  # second line; only columns, the values in the 1st line are > 2
```

```
[1] 7 8
```

Tasks / Exercises

Time for practical training! :)

Please continue to work on Exercises 5x) and 6x).

Lists (1)

- a list in **R** is a **ordered** collection of objects
- each object is part of the list
- the data types of the individual list elements can be different
  - vectors
  - matrices
  - lists (lists can be *recursive*!)
  - factors or data frame (to be discussed later)
- the dimension of each list item can be different
- lists can be used to group and summarize various objects within a single object.

Lists (2)

- with the function list(), a new list is generated

```
mylist <- list(v1=1:5, v2=10:20, mat=matrix(1:8, nrow=4), l1=list(a="A", b="B"))
mylist
```

```
$v1
[1] 1 2 3 4 5

$v2
 [1] 10 11 12 13 14 15 16 17 18 19 20

$mat
     [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    3    7
[4,]    4    8

$l1
$l1$a
[1] "A"

$l1$b
[1] "B"
```

Indexing lists (1)

- We have (at least) three ways to access elements of a list
  - the [] -operator
  - with the operator [[]]
  - with the $ -operator and the name of a list item
- with str()), you can view the structure of a list
- with names() you get the names of the list elements (highest level)

```
names(mylist)
```

```
[1] "v1"  "v2"  "mat" "l1"
```

Indexing lists (2)

- The result of indexing with [] is again a list
- Within [] the position of the desired list items are specified
- Positive, negative and logical indexing is possible

```
str(mylist)
```

```
List of 4
 $ v1 : int [1:5] 1 2 3 4 5
 $ v2 : int [1:11] 10 11 12 13 14 15 16 17 18 19 ...
 $ mat: int [1:4, 1:2] 1 2 3 4 5 6 7 8
 $ l1 :List of 2
  ..$ a: chr "A"
  ..$ b: chr "B"
```

```
res <- mylist[c(1,3)]; str(res)
```

```
List of 2
 $ v1 : int [1:5] 1 2 3 4 5
 $ mat: int [1:4, 1:2] 1 2 3 4 5 6 7 8
```

Indexing lists (3)

- Indexing with [[]] is used to accurately extract a list item
- The result is an object with the data type that has the desired list item

```
str(list)
```

```
function (...)
```

```
res1 <- mylist[[3]]; str(res1) # result is a matrix
```

```
 int [1:4, 1:2] 1 2 3 4 5 6 7 8
```

```
res2 <- mylist[3]; str(res2) # indexing with []: result is a list
```

```
List of 1
 $ mat: int [1:4, 1:2] 1 2 3 4 5 6 7 8
```

Indexing lists (4)

- Indexing with $ is used to accurately extract a list item (analogous to [[]])
- After the operator follows the name (not the position) of the desired list item
- The result is an object with the data type that has the desired list item

```
str (mylist)
```

```
List of 4
 $ v1 : int [1:5] 1 2 3 4 5
 $ v2 : int [1:11] 10 11 12 13 14 15 16 17 18 19 ...
 $ mat: int [1:4, 1:2] 1 2 3 4 5 6 7 8
 $ l1 :List of 2
  ..$ a: chr "A"
  ..$ b: chr "B"
```

```
res1 <- mylist$v2; str(res1) # result is a vector
```

```
 int [1:11] 10 11 12 13 14 15 16 17 18 19 ...
```

Indexing lists (5)

- Indexing types can be combined arbitrarily

```
str(mylist)
```

```
List of 4
 $ v1 : int [1:5] 1 2 3 4 5
 $ v2 : int [1:11] 10 11 12 13 14 15 16 17 18 19 ...
 $ mat: int [1:4, 1:2] 1 2 3 4 5 6 7 8
 $ l1 :List of 2
  ..$ a: chr "A"
  ..$ b: chr "B"
```

```
mylist[[1]][1] # first list element, first vector element
```

```
[1] 1
```

```
mylist[[1]][-1] # first list element, all but first vector
```

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

```
mylist$l1[[2]] # second list element of list 'l1'
```

```
[1] "B"
```

Tasks / Exercises

Time for practical training! :)

Please continue to work on Exercises 7x).

Factors (1)

- Factors in **R** is an important data type
- Are used to represent notional or ordinal data
  - *unordered factors* for nominally scaled data
  - *orderd factors* for ordinal scaled data
- Factors may be seen as special vectors
- Factors are internally
  - coded integers from 1 to n (# of occurrences)
  - each number is associated with a name (label)

Factors (2)

- Why should / can numeric or character variables be used as factors?
  - Factors are properly used in statistical modeling (correct number of degrees of freedom)
  - often different implementation of graphics for factors vs. numerical / character vectors
  - efficient storage of character vectors
- Factors have a more complex data structure; Factors include:
  - a numerically coded data vector
  - Labels for each level

Factors (3)

- Factors can be generated in **R** with the functions
  - **factor()**: for unordered factors
  - **ordered()**: for ordered factors
- must be specified as input only a numeric / character vector
- optional function arguments
  - *levels*: which characteristics of the input vector to be used as a factor levels
  - *labels*: description of Levels
  - *exclude*: which characteristics of the input vector to be interpreted as *missing values*.

Factors (4)

- produce factors with defaults of a character vector

```
gender <- c("m", "m", "w", "w", "w", "m", "w", "m", "m", "m", "w", "m")
gender.f1 <- factor(gender); gender.f1
```

```
 [1] m m w w w m w m m m w m
Levels: m w
```

- Specify order of the labels

```
gender.f2 <- factor(gender, levels=c("w", "m")); gender.f2 # other sequence
```

```
 [1] m m w w w m w m m m w m
Levels: w m
```

- defining custom labels

```
gender.f3 <- factor(gender, levels=c("w", "m"), labels=c("woman", "man")); gender.f3
```

```
 [1] man   man   woman woman woman man   woman man   man   man   woman
[12] man
Levels: woman man
```

Factors (5)

- The names of the characteristics of a factor can be changed with **levels()**:

```r
size <- factor(c(2, 3, 1, 1, 1, 2, 3, 3), levels=c(1, 2, 3),
  labels=c("small", "middle", "large"))
levels(size)
```

```
[1] "small"  "middle" "large"
```

- Allocation of new labels

```r
levels(size) <- c("s", "m", "l"); size
```

```
[1] m l s s s m l l
Levels: s m l
```

```r
levels(size)
```

```
[1] "s" "m" "l"
```

Factors (6)

- With function **levels()** it is also possible to combine levels
- simultaneous renaming the level is possible

```
size
```

```
[1] m l s s s m l l
Levels: s m l
```

```
levels (size) <- c("small", "small", "large") # + new labels
size
```

```
[1] small large small small small small large large
Levels: small large
```

Factors (7)

- So far no order of the levels of a factor
- you can use **ordered()** (alternatively function argument *ordered* in function **factor()**)

```
grades <- c(2,1,4,5,5,2,4,1,1,4,5,1)
labs <- c("non-sufficient", "satisfactory", "satisfactory", "good", "very good")
grades.f1 <- ordered(grades, levels=5:1, labels=labs); grades.f1
```

```
 [1] good           very good      satisfactory   non-sufficient
 [5] non-sufficient good           satisfactory   very good
 [9] very good      satisfactory   non-sufficient very good
5 Levels: non-sufficient < satisfactory < satisfactory < ... < very good
```

- Generating an ordered factor with **factor()**

```
grades.f2 = factor (grades, levels=5:1, labels=labs, order=TRUE)
identical(grades.f1, grades.f2)
```

```
[1] TRUE
```

- levels are ordered in output of **print()**

Factors (8)

- Factors can be converted to a numeric vector with **as.numeric()**

```
size
```

```
[1] small large small small small small large large
Levels: small large
```

```
as.numeric(size)
```

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

- Factors can be converted to a character vector with **as.character()**
- The value of the elements are the corresponding labels

```
as.character(size)
```

```
[1] "small" "large" "small" "small" "small" "small" "large" "large"
```

Indexing of factors (1)

- Indexing of factors is analogous to index vectors with the [] -operator
  - Positive / negative indexing
  - Logical indexing
- logical indexing by checking the levels of the factor (query with levels())
- the result of indexing is again a factor

```
size
```

```
[1] small large small small small small large large
Levels: small large
```

```
size[c(1,4,7)] # positive indexing
```

```
[1] small small large
Levels: small large
```

Indexing of factors (2)

- negative and logical indexing

```
size
```

```
[1] small large small small small small large large
Levels: small large
```

```
size[-c(1:4)] # negative indexing, element from the 5
```

```
[1] small small large large
Levels: small large
```

```
levels(grades.f1) # school notes, ordered factor
```

```
[1] "non-sufficient" "satisfactory"   "satisfactory"   "good"
[5] "very good"
```

```
grades.f1[grades.f1 == "non-sufficient" | grades.f1 == "very good"]
```

```
[1] very good      non-sufficient non-sufficient very good
[5] very good      non-sufficient very good
5 Levels: non-sufficient < satisfactory < satisfactory < ... < very good
```

Tasks / Exercises

Time for practical training! :)

Please continue to work on Exercises 8x) and 9x).

Data frames (1)

- Data frames (class *data.frame*) are an important data type in **R**
- Data frames may be considered as:
  - generalization of matrices
  - lists with some restrictions
- Data Frames are "special lists" (class *data.frame*)
- corresponds to the well-known rectangle data format from other software packages (*Excel*, *SPSS*) with
  - *rows* correspond to observation units
  - *columns*: variables

Data frames (2)

- Relationship between Data frames and matrices / lists
- A *data.frame* is like a *matrix* in which ...
  - column vectors / factors can be of different types (numeric, character, logical)
  - all vectors must have the same number of elements (same length)
- A *data.frame* is like a *list* in which ...
  - all list elements are vector / factors
  - all list elements have the same number of elements (equal length)
- data from *external sources* to be read are often stored as Data frames.

Data frames (3)

- Data frames are usually created by reading data into **R**
- Data frames can be constructed by the function **data.frame()**
- as input, at least one vector / factor is needed

```
name <- c("Bernhard", "Matthias", "Angelika")
gender <- c("m", "m", "w")
size <- c(185, 182, 165)

df1 <- data.frame(name, gender, size)
df1
```

```
      name gender size
1 Bernhard      m  185
2 Matthias      m  182
3 Angelika      w  165
```

Data frames (4)

- Vectors / factors can also be constructed directly

```
df2 <- data.frame(
 name=c("Bernhard", "Matthias", "Angelika"),
 gender=c("m", "m", "w"),
 size=c(185, 182, 165)
)
df2
```

```
      name gender size
1 Bernhard      m  185
2 Matthias      m  182
3 Angelika      w  165
```

```
identical(df1, df2) # all the same
```

```
[1] TRUE
```

Data frames (5)

- When **constructing** Data frames
  - numerical vectors are stored unchanged
  - character vectors are (by default) converted into factors; this case corresponds to the levels being the (unique) characteristics of the input

```
str(df1)
```

```
'data.frame':	3 obs. of  3 variables:
 $ name  : Factor w/ 3 levels "Angelika","Bernhard",..: 2 3 1
 $ gender: Factor w/ 2 levels "m","w": 1 1 2
 $ size  : num  185 182 165
```

- This behavior can be changed by setting the *stringsAsFactors* parameter
  - *stringsAsFactors=TRUE*: automatic conversion of character vectors to factors
  - *stringsAsFactors=FALSE*: no automatic conversion

Data frames (6)

- default behavior: option *stringsAsFactors*

```
name <- c("Bernhard", "Matthias", "Angelika");
gender <- c("m", "m", "w"); size <- c (185, 182, 165)
str(data.frame(name, gender, size, stringsAsFactors=TRUE)) # default behavior
```

```
'data.frame':	3 obs. of  3 variables:
 $ name  : Factor w/ 3 levels "Angelika","Bernhard",..: 2 3 1
 $ gender: Factor w/ 2 levels "m","w": 1 1 2
 $ size  : num  185 182 165
```

- No automatic conversion of character vectors

```
str(data.frame (name, gender, size, stringsAsFactors=FALSE)) # no conversion
```

```
'data.frame':	3 obs. of  3 variables:
 $ name  : chr  "Bernhard" "Matthias" "Angelika"
 $ gender: chr  "m" "m" "w"
 $ size  : num  185 182 165
```

- Is *stringsAsFactors=FALSE*: possible recoding of character variables to factors later using **factor()** or **ordered()**.

Indexing of Data frames (1)

- A lot of possibilities exists to subset a Data frame (-> *data Management part*)
- syntax: [*index row*, *index columns*] (like matrices)
- positive, negative and logical indexing is possible
- the type of indexing may be different for row index and column index
- if empty, all elements are selected (rows or columns)
- access to individual columns with the $ -operator (like lists)
- an alternative: function **subset()**

Indexing of Data frames (2)

- Indexing of data frames using []

```
df1
```

```
      name gender size
1 Bernhard      m  185
2 Matthias      m  182
3 Angelika      w  165
```

```
df1[c(1,3), 1:3] # first and third line, all columns
```

```
      name gender size
1 Bernhard      m  185
3 Angelika      w  165
```

```
df1[,-3] # all lines, not the third column
```

```
      name gender
1 Bernhard      m
2 Matthias      m
3 Angelika      w
```

Indexing of Data frames (3)

- Indexing of a data frame by $

```
df1
```

```
      name gender size
1 Bernhard      m  185
2 Matthias      m  182
3 Angelika      w  165
```

```
df1$name # access to variable 'name'
```

```
[1] Bernhard Matthias Angelika
Levels: Angelika Bernhard Matthias
```

- logical indexing

```
df1[df1$size <= 180, ] # all lines, where applicable: size <= 180
```

```
      name gender size
3 Angelika      w  165
```

Indexing of Data frames (4)

- Indexing of data Frames using **subset()**

```
df1
```

```
      name gender size
1 Bernhard      m  185
2 Matthias      m  182
3 Angelika      w  165
```

```
subset(x = df1, subset = gender == "m")
```

```
      name gender size
1 Bernhard      m  185
2 Matthias      m  182
```

- Optional arguments of **subset ()**

```
# Subset of df1, where applicable: gender == "m", without variable 'gender'
subset(x = df1, subset = gender == "m", select = -gender)
```

```
      name size
1 Bernhard  185
2 Matthias  182
```

Indexing of Data frames (5)

- Indexing to individual columns returns a vector / factor with the appropriate type (*numeric*, *character*, *logical*)

```
name <- df1$name; name
```

```
[1] Bernhard Matthias Angelika
Levels: Angelika Bernhard Matthias
```

```
size <- df1$size; size
```

```
[1] 185 182 165
```

- should the result be a *data.frame*, use data.frame() again

```
data.frame(name2 = df1$name[c(1,3)]) # with indexing!
```

```
     name2
1 Bernhard
2 Angelika
```

Working with Data frames (1)

- Data frames are very common in **R**
- a few helpful functions that can be used in conjunction with data frames are:
  - **dim()**: the dimension (number of rows and columns)
  - **nrow()**: number of lines
  - **ncol()**: number of columns
  - **head()**first (default 6) rows of a data frame
  - **tail()**: last (default 6) rows of a data frame
  - **rownames()**: row Labels
  - **colnames()**: columns / variable names

Working with Data frames (2)

- Some useful functions

```
df1
```

```
       name gender size
1 Bernhard      m  185
2 Matthias      m  182
3 Angelika      w  165
```

```
dim(df1) # vector with 2 elements
```

```
[1] 3 3
```

```
nrow(df1) # number of rows
```

```
[1] 3
```

```
ncol(df1) # number of columns
```

```
[1] 3
```

Working with Data frames (3)

- Some useful functions

```
df1
```

```
      name gender size
1 Bernhard      m  185
2 Matthias      m  182
3 Angelika      w  165
```

```
rownames(df1)
```

```
[1] "1" "2" "3"
```

```
colnames(df1)
```

```
[1] "name"   "gender" "size"
```

```
tail(df1, n = 1) # last observation
```

```
      name gender size
3 Angelika      w  165
```

Tasks / Exercises

Time for practical training! :)

Please continue to work on Exercises 10x) and 11x).

Special symbols (1)

- In practice, data is almost never complete
- in **R**, other special symbols are also present
- missing values are often called **NA** (**N**ot **A**vailable) encodes
- special symbol (**NaN**) (**N**ot **a N**umber) for missing values resulting from calculations impossible
- **NULL**: often the result of calculations / expressions whose result is undefined.
- **Inf** / **-Inf**: special symbols (minus) infinity

Special symbols - NA (1)

- NA's may be caused by conversion, e.g.

```
v1 <- c(1:2, "a", 3:4, "b"); v1 # automatically stored as character-vector
```

```
[1] "1" "2" "a" "3" "4" "b"
```

```
v2 <- as.numeric(v1); v2 # not all elements can be converted -> NA
```

```
[1]  1  2 NA  3  4 NA
```

- which element is NA using is.na()

```
is.na(v2) # returns logical vector
```

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

- put NA's:

```
v2[1] <- NA; v2
```

```
[1] NA  2 NA  3  4 NA
```

Special symbols - NaN (1)

- NaN is often the result of non-valid calculations

```
v1 <- sqrt(c(-2, 0, 2)); v1 # square root of negative numbers is not defined!
```

```
[1]       NaN 0.000000 1.414214
```

```
v2 <- log(c(-1,1)); v2 # logarithmus naturalis not defined for neg. values!
```

```
[1] NaN   0
```

- query with is.nan()

```
is.nan(v1) # of logical vector with TRUE at the positions where v1 == NaN applies
```

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

```
which(is.nan(v2)) # the indices in which v2 == NaN applies
```

```
[1] 1
```

Special symbols - NULL (1)

- **NULL** is often the result of expressions whose result is undefined

```
v1 <- c(); v1
```

```
NULL
```

- query **NULL** elements with **is.null()**

```
is.null(v1) # v1 is an empty vector, which is initialized to NULL
```

```
[1] TRUE
```

Special symbols - Inf and -Inf (1)

- Inf and -Inf are often the result of expressions whose result is undefined

```
v1 <- log(-1:1); v1
```

```
[1]  NaN -Inf    0
```

```
v2 <- 1/0; v2
```

```
[1] Inf
```

- Query whether an element of a vector finite or non-finite with is.finite () or is.infinite()

```
which(is.finite(v1)) # of positions at which there are finite values v1
```

```
[1] 3
```

```
is.infinite(v2) # division by 0 is not finite
```

```
[1] TRUE
```

Tasks / Exercises

Time for practical training! :)

Please continue to work on Exercises 12x).

Summary (1)

- to get familiar with basic **data types**
  - *numeric*: numbers
  - *character*: alphanumeric characters / strings
  - *boolean*: logical values TRUE / FALSE
- Introduction of significant **data structures**
  - *vectors*: elements of the same data type
  - *factors*: structure of nominal / ordinal scaled data
  - *matrices*: rectangular, same data type
  - *lists*: contexts of different objects
  - *data.frame*: 'flexible' rectangle data format

Summary (2)

- construct sequences with **seq()**
- replications using **rep()**
- indexing of the individual data structures with **[] [[]]** or **$**
    - Positive, negative and logical indexing
- to get familiar with helpful functions like:
    - **c()**, **is.numeric()**, **is.character()**, **is.logical()**, **length()**, **unique()**, **matrix()**, **rbind()**, **cbind()**, **list()**, **factor()**, **ordered()**, **levels()**, **data.frame()**, **dim()**, **head()**, **tail()**, **str()**, **subset()**, **names()**, **rownames()**, **colnames()**

Summary (3)

- Getting familiar with special symbols:
  - NA: missing values; query with is.na()
  - NaN: impossible arithmetic expressions; query with is.nan()
  - Inf /-Inf: +/- infinity; query with is.finite() / is.infinite ()

Functions in R

Alexander Kowarik, Bernhard Meindl

Overview/Objectives

- Learn about important and commonly used existing functions from **R**
  - Mathematical functions
  - Statistical functions
  - Operations for dealing with character vectors
- Writing your own functions
  - Basic syntax
  - Local versus global variables (*Scoping*)
- Control structures

Functions (1)

- Typical: application of a function on one or more vectors
- Syntax for calling a function

```
res1 <- name_of_function(v1) # an input argument
res2 <- name_of_function(v1, v2) # two input arguments
```

- Functions often have additional function arguments with default values
- You get access to all function arguments with **args()**

```
args(sample)
```

```
function (x, size, replace = FALSE, prob = NULL)
NULL
```

- Help for a function can be found with

```
?sample # alternatively ?"sample"
```

Functions (2)

- Functions with empty input are possible but rare
- Usually: element-wise application of the function

```
x <- 1:5; y <- 6:10; z <- -1:3; x + y + z # "+" is a function
```

```
[1]  6  9 12 15 18
```

```
exp(x=x) # application of the exp-function on each element of x
```

```
[1]   2.718282   7.389056  20.085537  54.598150 148.413159
```

- non-sense computations: R usually gives a *warning*

```
log(z)
```

```
Warning in log(z): NaNs produced
```

```
[1]       NaN      -Inf 0.0000000 0.6931472 1.0986123
```

Functions (3)

- When applying a function to more than one vector, the vectors must have the same length
- If not: –> *Recycling* of the shorter vector

```
x <- 1:6; y <- 1:2; x + y # y is repeated 3 times
```

```
[1] 2 4 4 6 6 8
```

- Some functions return only one value

```
sum(x=1:6) # sum of the integers 1-6
```

```
[1] 21
```

```
prod(x=1:6) # product of integers 1-6
```

```
[1] 720
```

Mathematical Functions (1)

- The most important mathematical functions and operators are available:
  - Basic arithmetic: +, -, *, /
  - Modulo operator (remainder from division): %%

```
1:20 %% 3 # rest when dividing by 3
```

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

  - Absolute value with abs(), square root with sqrt()

```
abs(-5:5)
```

```
[1] 5 4 3 2 1 0 1 2 3 4 5
```

```
sqrt(-5:5)
```

```
[1]      NaN      NaN      NaN      NaN      NaN 0.000000 1.000000
[8] 1.414214 1.732051 2.000000 2.236068
```

Mathematical Functions (2)

- Further important mathematical functions:
  - Logarithm with log(), exponential function with exp()

```
log(x=1:5) # by default logarithm naturalis with base=exp(1)
```

```
[1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379
```

```
log(x=1:5, base=2) # or analoguously: log2(x=1:5)
```

```
[1] 0.000000 1.000000 1.584963 2.000000 2.321928
```

```
exp(1:5)
```

```
[1]   2.718282   7.389056  20.085537  54.598150 148.413159
```

  - Trigonometric functions: sin(), cos(), tan() (?Trig)

```
cos(seq(from=0, to=90, by=10))
```

```
[1]  1.0000000 -0.8390715  0.4080821  0.1542514 -0.6669381  0.9649660
[7] -0.9524130  0.6333192 -0.1103872 -0.4480736
```

Functions for Rounding (1)

- Various functions to round non-integer vectors are available in **R**:
  - **round()**: Rounding vectors to desired number of decimal places (argument *digits*)

```
x <- exp(1:5); x
```

```
[1]   2.718282   7.389056  20.085537  54.598150 148.413159
```

```
round(x, digits=3)
```

```
[1]   2.718   7.389  20.086  54.598 148.413
```

```
round(x, digits=2)
```

```
[1]   2.72   7.39  20.09  54.60 148.41
```

```
round(x, digits=1)
```

```
[1]   2.7   7.4  20.1  54.6 148.4
```

Functions for Rounding (2)

- If *digits=0* (the default), then rounding to integer

```
x <- exp(1:5); x
```

```
[1]   2.718282   7.389056  20.085537  54.598150 148.413159
```

```
round(x) # digits=0
```

```
[1]   3   7  20  55 148
```

- Special case: rounding to the next **even** number

```
round(c(1.5,2.5,3.5,4.5), digits=0)
```

```
[1] 2 2 4 4
```

Functions for Rounding (3)

- Rounding possible to multiples of 10 by a negative value for *digits*

```
x <- seq(from=-100, to=100, length=20); x
```

```
 [1] -100.000000  -89.473684  -78.947368  -68.421053  -57.894737
 [6]  -47.368421  -36.842105  -26.315789  -15.789474   -5.263158
[11]    5.263158   15.789474   26.315789   36.842105   47.368421
[16]   57.894737   68.421053   78.947368   89.473684  100.000000
```

- To multiples of 10 and 100

```
round(x, digits=-1) # multiples of 10
```

```
 [1] -100  -90  -80  -70  -60  -50  -40  -30  -20  -10   10   20   30   40
[15]   50   60   70   80   90  100
```

```
round(x, digits=-2) # multiples of 100
```

```
 [1] -100 -100 -100 -100 -100    0    0    0    0    0    0    0    0    0
[15]    0  100  100  100  100  100
```

Functions for Rounding (4)

- Further functions for rounding
  - **floor(x)**: rounding to the next integer less than *x*
  - **ceiling(x)**: rounding to the next integer greater than *x*
  - **trunc(x)**: rounding to the next integer towards 0

```
floor(c(-123.123, 123.123))
```

```
[1] -124  123
```

```
ceiling(c(-123.123, 123.123))
```

```
[1] -123  124
```

```
trunc(c(-123.123, 123.123))
```

```
[1] -123  123
```

Tasks / Exercises

Time for practical training! :)

Please continue to work on Exercises 1x) and 2x).

Probability functions (1)

- 4 categories of built-in probability functions:
  - rXY: generating random numbers
  - dXY: value of the density function at a certain point
  - pXY: value of the distribution function at a certain point
  - qXY: quantiles (inverse distribution function!)
- XY stands for the probability distribution, for example,
  - unif: uniform distribution
  - norm: standard normal distribution
  - binom: binomial distribution
  - t: Student t-distribution
  - gamma: Gamma distribution
  - pois: Poisson distribution

Probability functions (2)

- E.g. normal distribution, XY=**norm**

- Default values to the standard normal distribution

  - **rnorm()**: generate standard normally distributed random numbers

```
rnumb <- rnorm(n=10); rnumb # standard normal distribution
```

```
[1]  0.33536743  0.05513452  0.88828660  0.28122583  0.74250195
[6] -0.39125695  1.46182775 -1.10254270 -0.59118512  0.48419882
```

  - **dnorm(q)**: value of the density function of a normal distribution

```
dnorm(c(-2, 0, 2)) # standard normal distribution, symmetric
```

```
[1] 0.05399097 0.39894228 0.05399097
```

Probability functions (3)

- More information on normal distribution
  - **pnorm()**: value of the distribution function at a certain value of the random variable

```
pnorm(0) # probability that a standard normally distributed random number <= 0
```

```
[1] 0.5
```

- By setting the function argument *lower.tail=FALSE* you get the opposite probability

```
pnorm(0,lower.tail=F) # probability that a standard normally distributed random number > 0
```

```
[1] 0.5
```

Probability functions (4)

- Yet more on normal distribution:
  - **qnorm()**: Calculates for a given probability the corresponding quantile
  - helpful e.g. for calculation of z-values

```
qnorm(0.95, mean=0, sd=1)
```

```
[1] 1.644854
```

```
qnorm(0.975, mean=0, sd=1)
```

```
[1] 1.959964
```

- Interpretation: at x=1.64485 the distribution function of the standard normal distribution is equal to 0.95, at x=1.95996 its value is 0.975.

Probability functions (5)

- Similar procedure for other distributions
- E.g. density function of a binomial distribution: ?dbinom

```
# Probability for x, given a binomial distribution with n=20 and prob=0.8
dbinom(x=c(10,15,20), size=20, prob=0.8)
```

```
[1] 0.002031414 0.174559522 0.011529215
```

- Random numbers from a binomial distribution rbinom?

```
# 10 random numbers from a binomial distribution with n=20 and prob=0.8
rbinom(n=10, size=20, prob=0.8)
```

```
 [1] 18 15 11 17 20 17 17 14 19 16
```

Functions for data analysis (1)

- A variety of functions for (descriptive) statistical analysis implemented in **R**:
  - Measures of location: **mean()**, **median()**, **quantile()**
  - Measures of dispersion/association: **sd()**, **IQR()**, **var()**, **cor()**, **cov()**
  - Other useful functions: **min()**, **max()**, **range()**, **pmin()**, **pmax()**, **summary()**
- Frequently, an important function argument is *na.rm*
  - If *na.rm=TRUE*: missing values are ignored in calculations

Functions for data analysis (2)

- Arithmetic mean with **mean()**

```
x <- rnorm(5); x
```

```
[1]  1.6700110 -1.2293067  0.4903000 -0.8196558 -0.9033072
```

```
y <- c(NA, x); y
```

```
[1]         NA  1.6700110 -1.2293067  0.4903000 -0.8196558 -0.9033072
```

```
c(mean(x), mean(y)) # na.rm=FALSE is the default value
```

```
[1] -0.1583918         NA
```

```
mean(y, na.rm=TRUE) # the missing value is ignored
```

```
[1] -0.1583918
```

```
identical(mean(x), mean(y, na.rm=TRUE)) # same result
```

```
[1] TRUE
```

Functions for data analysis (3)

- Median and other quantiles with **median()** or **quantile()**

```
x <- runif(12); x # 12 uniformly distributed random numbers between 0 and 1
```

```
 [1] 0.44517752 0.42247296 0.49303146 0.07859673 0.53838826 0.24780883
 [7] 0.63488438 0.64589409 0.94603666 0.86903389 0.35607793 0.39024126
```

```
median(x)
```

```
[1] 0.4691045
```

```
quantile(x, 0.5) == median(x) # median corresponds to the 0.5 quantile
```

```
 50%
TRUE
```

```
quantile(x, c(0.2, 0.5, 0.8)) # simultaneous calculation of several quantiles
```

```
      20%       50%       80%
0.3629106 0.4691045 0.6436921
```

- Also **median()** and **quantile()** have the argument *na.rm*

Functions for data analysis (4)

- Standard deviation and variance with sd() or var()

```
x <- c(NA, runif(10)); x
```

```
 [1]         NA 0.81071815 0.80636774 0.76507860 0.37284502 0.41134609
 [7] 0.01598997 0.96760011 0.16216713 0.17288889 0.45954427
```

```
sd(x) # NA because of missing value
```

```
[1] NA
```

```
sd(x, na.rm=TRUE) # standard deviation without the missing value
```

```
[1] 0.3266121
```

```
sqrt(var(x, na.rm=TRUE)) == sd(x, na.rm=TRUE) # sd = square root of the variance
```

```
[1] TRUE
```

- Also sd() and var() have an argument *na.rm*

Functions for data analysis (5)

- Interquartile range as a robust alternative to the standard deviation with **IQR()**

```
x <- c(100, runif(10)); x
```

```
 [1] 100.0000000   0.3130625   0.4869478   0.5533436   0.5706249
 [6]   0.9487287   0.1651601   0.3291793   0.4736841   0.1195373
[11]   0.4579937
```

```
sd(x) # an outlier affects this estimator
```

```
[1] 30.01875
```

```
IQR(x) # IQR as the difference between 0.75 and the 0.25 quantile
```

```
[1] 0.2408634
```

```
IQR(x) == diff(quantile(x, c(0.25, 0.75))) # check result
```

```
 75%
TRUE
```

Functions for data analysis (6)

- Correlation and covariance with **cor()** and **cov()**

```
x <- runif(10); y <- rnorm(10); x; y
```

```
 [1] 0.18817187 0.79449283 0.12044702 0.07713736 0.41590585 0.01195501
 [7] 0.02776343 0.32407393 0.66838852 0.60502378
```

```
 [1] -0.8777005  1.6898804  2.6526438 -0.7458879 -0.5699749 -0.9694182
 [7] -0.5045833  0.8034303  0.5719746  0.9332804
```

```
cov(x,y); sum((x-mean(x))*(y-mean (y)))/(length(x)-1)
```

```
[1] 0.1577068
```

```
[1] 0.1577068
```

```
cor(x,y, method="pearson"); cov(x,y) / (sd(x)* sd(y))
```

```
[1] 0.447934
```

```
[1] 0.447934
```

Functions for data analysis (7)

- also cor() and cov() have an argument *na.rm*
- also matrices or data frames can be used as input

```
df <- data.frame(x=rnorm(10), y=rnorm(10, mean=2, sd=5), z=runif(10))

cor(df, method="kendall") # correlation matrix by Kendall
```

```
            x           y           z
x  1.00000000 -0.02222222  0.37777778
y -0.02222222  1.00000000 -0.02222222
z  0.37777778 -0.02222222  1.00000000
```

```
cov(df) # covariance matrix of the three variables x, y and z
```

```
            x           y          z
x  1.66382095 -0.11516504 0.06904195
y -0.11516504 20.91340881 0.08076466
z  0.06904195  0.08076466 0.10161822
```

```
c(var(df$x), var(df$y), var(df$z)) # diagonal elements
```

```
[1]  1.6638210 20.9134088  0.1016182
```

Functions for data analysis (8)

- Other useful functions are: **min()**, **max()**, **range()**

```
x <- c(rnorm(10), NA)
c(min(x), min(x, na.rm=TRUE)) # minimum with and without consideration of NA
```

```
[1]        NA -1.164033
```

```
c(max(x), max(x, na.rm=TRUE)) # maximum with and without consideration of NA
```

```
[1]       NA 2.243294
```

```
range(x, na.rm=TRUE) # range of the vector
```

```
[1] -1.164033  2.243294
```

- parallel maxima and minima with **pmin()** and **pmax()**

```
pmax(x=1:5, y=5:1); pmin(x=1:5, y=5:1) # element-wise max/min
```

```
[1] 5 4 3 4 5
```

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

Functions for data analysis (9)

- Summary of a data object with **summary()**
- Different output depending on the type of input

```
charv <- rep(c ("A","B","C"), each=3); fac <- factor(charv)
summary(rnorm(10)) # min, max, quartiles and mean values for numeric vector
```

```
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max. 
-1.37300 -1.10100 -0.60570 -0.30420  0.02486  1.58600 
```

```
summary(charv) # generic output for character vector
```

```
   Length     Class      Mode 
        9 character character 
```

```
summary(fac) # tabulation for factor
```

```
A B C 
3 3 3 
```

Tasks / Exercises

Time for practical training! :)

Please continue to work on Exercises 3x) and 4x).

Functions for strings (1)

- Special functions for dealing with strings
- These functions are often *vectorized*
- A list of important functions:
  - **nchar()**: number of characters
  - **tolower()**, **toupper()**: converting to lowercase or uppercase
  - **paste()**: connecting strings
  - **substring()**: extracting parts of a string
  - **strsplit()**: splitting a string into several parts

Functions for strings (2)

- more functions for strings
  - sub(), gsub(): replacement within strings
  - match(): matching a string with a vector
  - grep(): searching for patterns in a character vector
- External (useful) add-on packages for manipulating strings:
  - Packages stringr or stringi

Functions for strings (3)

- nchar() calculates the number of characters/letters

```
data(mtcars); strvec <- rownames(mtcars); strvec
```

```
 [1] "Mazda RX4"           "Mazda RX4 Wag"       "Datsun 710"
 [4] "Hornet 4 Drive"      "Hornet Sportabout"   "Valiant"
 [7] "Duster 360"          "Merc 240D"           "Merc 230"
[10] "Merc 280"            "Merc 280C"           "Merc 450SE"
[13] "Merc 450SL"          "Merc 450SLC"         "Cadillac Fleetwood"
[16] "Lincoln Continental" "Chrysler Imperial"   "Fiat 128"
[19] "Honda Civic"         "Toyota Corolla"      "Toyota Corona"
[22] "Dodge Challenger"    "AMC Javelin"         "Camaro Z28"
[25] "Pontiac Firebird"    "Fiat X1-9"           "Porsche 914-2"
[28] "Lotus Europa"        "Ford Pantera L"      "Ferrari Dino"
[31] "Maserati Bora"       "Volvo 142E"
```

```
nchar(strvec)
```

```
 [1]  9 13 10 14 17  7 10  9  8  8  9 10 10 11 18 19 17  8 11 14 13 16 11
[24] 10 16  9 13 12 14 12 13 10
```

- nchar() is vectorized
- *Blanks* are counted

Functions for strings (4)

- **tolower()** and **toupper()** convert strings to lowercase or uppercase

```
head(strvec) # original
```

```
[1] "Mazda RX4"         "Mazda RX4 Wag"     "Datsun 710"
[4] "Hornet 4 Drive"    "Hornet Sportabout" "Valiant"
```

```
head(tolower(strvec)) # conversion into lowercase letters
```

```
[1] "mazda rx4"         "mazda rx4 wag"     "datsun 710"
[4] "hornet 4 drive"    "hornet sportabout" "valiant"
```

```
head(toupper(strvec)) # conversion into uppercase letters
```

```
[1] "MAZDA RX4"         "MAZDA RX4 WAG"     "DATSUN 710"
[4] "HORNET 4 DRIVE"    "HORNET SPORTABOUT" "VALIANT"
```

- The functions are applied simultaneously to all elements of a vector

Functions for strings (5)

- Strings can be connected with **paste()**
- Numerical vectors are automatically converted into characters

```
paste("result of a random drawing:", sample(1:20,1))
```

```
[1] "result of a random drawing: 13"
```

- Use: e.g. for generating variable names

```
c(paste("v",1:5, sep=""), paste("x",1:3, sep="_"))
```

```
[1] "v1"  "v2"  "v3"  "v4"  "v5"  "x_1" "x_2" "x_3"
```

- If the function argument *collapse* is specified, the result is a character vector of length 1

```
paste("v",1:5, sep="",collapse=".")
```

```
[1] "v1.v2.v3.v4.v5"
```

Functions for strings (6)

- With **substring()** you can extract parts of a string

```
head(strvec)
```

```
[1] "Mazda RX4"         "Mazda RX4 Wag"     "Datsun 710"
[4] "Hornet 4 Drive"    "Hornet Sportabout" "Valiant"
```

```
substring(strvec, first=1, last=3) # first three characters of each element
```

```
 [1] "Maz" "Maz" "Dat" "Hor" "Hor" "Val" "Dus" "Mer" "Mer" "Mer" "Mer"
[12] "Mer" "Mer" "Mer" "Cad" "Lin" "Chr" "Fia" "Hon" "Toy" "Toy" "Dod"
[23] "AMC" "Cam" "Pon" "Fia" "Por" "Lot" "For" "Fer" "Mas" "Vol"
```

- can also be used to replace

```
substring(strvec, first=1, last=3) <- "xxx"
head(strvec)
```

```
[1] "xxxda RX4"         "xxxda RX4 Wag"     "xxxsun 710"
[4] "xxxnet 4 Drive"    "xxxnet Sportabout" "xxxiant"
```

Functions for Strings (7)

- strsplit() can split strings

```
v <- "we participate in the R course"
res <- strsplit(v, " "); res
```

```
[[1]]
[1] "we"          "participate" "in"          "the"         "R"          
[6] "course"     
```

- The result is a list. With unlist() we obtain a vector which usually can be indexed:

```
v <- unlist(res); v
```

```
[1] "we"          "participate" "in"          "the"         "R"          
[6] "course"     
```

```
v[length(v):1] # revert vector
```

```
[1] "course"      "R"           "the"         "in"          "participate"
[6] "we"         
```

Functions for strings (8)

- With **sub()**, **gsub()** you can replace parts within strings

```
s <- "European people are great people!"
sub(pattern="great", replacement="kind", x=s)
```

```
[1] "European people are kind people!"
```

- **sub()** replaces only the first occurrence, **gsub()** replaces all

```
s <- "European people are great people!"
sub(pattern="people", replacement="cars", x=s) # only the first replaced
```

```
[1] "European cars are great people!"
```

```
gsub(pattern="people", replacement="cars", x=s) # all replaced
```

```
[1] "European cars are great cars!"
```

Functions for strings (9)

- match() checks if a string exists in a vector (exact matching)

```
s <- c("aa", "bb", "aa", "aa", "dd", "yy")
match("by", table=s, nomatch=-1) # is 'by' contained in s?
```

```
[1] -1
```

```
match(x="yy", table=s) # yy is contained (at position 6)
```

```
[1] 6
```

```
# Check for each element of s if it is equal to 'bb'
match(x=s, table="bb", nomatch=-1)
```

```
[1] -1  1 -1 -1 -1 -1
```

- match() is thus also vectorized

Functions for strings (10)

- **grep()** searches for a pattern in a character vector
- returned are the positions of matches or a vector of length 0

```
s <- c ("aa", "bb", "aa", "aa", "dd", "yy")
length(grep(pattern="aax", x=s)) # no match, result vector has length 0
```

```
[1] 0
```

```
grep(pattern="dd", x=s) # exact match
```

```
[1] 5
```

```
grep(pattern="b", x=s) # partial matching
```

```
[1] 2
```

```
grep(pattern="a", x=s) # partial matching, multiple matches
```

```
[1] 1 3 4
```

Tasks / Exercises

Time for practical training! :)

Please continue to work on Exercises 5x).

Control structures (1)

- R offers the standard control structures of other programming languages
- if-statements: if, else, ifelse
  - when using if, the else is optional

```
x <- runif(1); x
```

```
[1] 0.6982489
```

```
if (x<0) {
  cat("x is negative!\n") # \n is a line break
} else {
  cat("x is positive!\n")
}
```

```
x is positive!
```

```
ifelse (x>0, 1, 2) # 1 if condition is true, otherwise 2
```

```
[1] 1
```

Control structures (2)

- loop constructs can often be circumvented by vectorization
- They exist anyway! Syntax

```
x <- runif(3)
for (i in 1:length(x)) {
  cat ("x at the point", i,"=",x[i],"\n")
}
```

```
x at the point 1 = 0.3258558
x at the point 2 = 0.5170702
x at the point 3 = 0.9173609
```

- any arbitrary R-code can appear within the curly brackets
- loop variable *i* (in this case) can be referenced in the loop

Control structures (3)

- There are also **while** loops

```
x <- runif(3); i <- 1
while (i <= length(x)) {
  cat("x at the point",i,"=",x[i],"\n")
  i <- i + 1 # add to loop variable, otherwise break condition is never satisfied
}
```

```
x at the point 1 = 0.7574479
x at the point 2 = 0.8991252
x at the point 3 = 0.6066138
```

- Note that the break condition is true at some point
- Otherwise: infinite loop
- Same result by avoiding loop using vectorization

```
cat(paste("x at the point",1:length(x),":",x,collapse="\n"))
```

```
x at the point 1 : 0.757447920972481
x at the point 2 : 0.899125213036314
x at the point 3 : 0.606613763840869
```

Own functions (1)

- Functions are an essential element in the **R** language
- Allow to separate repetitive code
- General syntax:

```
my_first_function <- function(input1, input2) {
  # R-code is within the curly brackets
  res <- input1+input2
  # then the result is returned
  return(res)
}
# Call
my_first_function(input1=1:5, input2=5:1)
```

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

- each function can return more than one (but an arbitrarily complex) object
- either in **return()** or the last expression

Own functions (2)

- Empty argument list is possible
- Definition of default values for individual functional arguments is possible
- Default values do not have to be specified
- Order of argument names for function call does not matter
- **Warning:** local variables are created and used within functions
- Use of undefined variables in a function:
  - –> probably use of *global variables* (scoping)

Own functions (3)

- A first (poor) function:

```
f1 <- function(v1, v2) {
  return(sum(v1+v2))
}
f1(v1=1:3, v2=2:5)
```

```
[1] 21
```

- What if *v1* and *v2* have different length?
- What if *v1* or *v2* are not numeric?
- More meaningful default values possible?

Own functions (4)

```r
# better
f2 <- function(v1=0, v2=1) {
  if (length(v1) != length(v2)) {
    return (NA) # return NA if different number of elements
  }
  if (!is.numeric(v1) | !is.numeric(v2)) {
    return(NA) # return NA if non-numeric vectors
  }
  return(sum(v1+v2))
}
```

- Call

```r
f2(v2=4:7, v1=1:3) # different length, sequence of arguments!
```

```
[1] NA
```

```r
f2(v1=1:3, v2=c("a", "b", "c")) # not all inputs are numeric
```

```
[1] NA
```

```r
f2() # use the default values
```

```
[1] 1
```

Own functions (5)

- Local versus global variables (scoping)

```r
x <- 10
fn1 <- function(y=3) {
  y + x # value of this statement is returned
}
fn2 <- function(y=3, x=5) {
  y + x # value of this statement is returned
}
c(fn1(), fn2()) # call with default values
```

```
[1] 13  8
```

- no local variable *x* exists in *fn1()*
  - -> global *x* (with value 10) is used
- local variable *x* in *fn2()* is used
- frequent source of error when creating own functions

Tasks / Exercises

Time for practical training! :)

Please continue to work on Exercises 6x) and 7x).

Summary (1)

- Existing **R** functions were presented.
  - Mathematical functions: **exp()**, **log()**, **sum()**, **prod()**, **+**, **-**, *****,**/**, **%%**, **abs()**, **sqrt()**, **sin()**, **cos()**, **tan()**, **round()**, **floor()**, **ceiling()**, **trunc()**
  - Probability functions:
    - **rXY()**: generating random numbers
    - **dXY()**: value of a density function
    - **pXY()**: value of the distribution function
    - **qXY()**: quantiles (inverse distribution function)
    - *XY* is e.g. **norm** for the normal distribution

Summary (2)

- Furthermore, we discussed:
    - functions for data analysis: **mean(), median(), quantile(), sd(), IQR(), var(), cor(), cov(), min(), max(), range(), pmin(), pmax(), summary()**
    - functions for strings: **nchar(), tolower(), toupper(), paste(), substring(), strsplit(), sub(), gsub(), grep()**
- The *concept of vectorization* was explained
- The *concept of recycling* was presented

Summary (3)

- It has been shown how to create own functions
  - Necessary syntax has been shown
  - We pointed at the *scoping* problem
- Important control structures were presented:
  - if, else, ifelse
- Two main loop constructs were presented:
  - for, while

Tables in R

Alexander Kowarik, Bernhard Meindl

Overview / Objectives

- Creating tables in **R**
  - Frequency tables
  - Contingency tables
- Plenty of possibilities ...
  - introduce some possible ways
- Tables with relative frequencies
- Adding marginal sums
- Calculation with tables

Tables - Warm-up

- Tables are a simple way to aggregate data
  - **Frequency tables:** absolute frequencies for each occurrence of a variable
  - **Contingency tables:** absolute frequencies for every possible combination of several variables
- **Relative frequencies:** proportion of occurrence(s) of every combination on the number of units

Frequency tables (1)

Load and view the **airquality** data:

```
data(airquality)
airquality$Oz2 <- cut(airquality$Ozone, c(-Inf, 80, Inf), labels=c("<= 80", ">80"))
head(airquality)
```

```
  Ozone Solar.R Wind Temp Month Day   Oz2
1    41     190  7.4   67     5   1 <= 80
2    36     118  8.0   72     5   2 <= 80
3    12     149 12.6   74     5   3 <= 80
4    18     313 11.5   62     5   4 <= 80
5    NA      NA 14.3   56     5   5  <NA>
6    28      NA 14.9   66     5   6 <= 80
```

- Daily measurements of air quality in New York from May to September 1973
- Also solar radiation, wind speed and temperature are recorded
- Measurements of ozone are not complete

Frequency tables (2)

- Count all occurrences of a variable by **table()**

```
tab <- table (airquality$Month); tab # number of measurements per month
```

```

 5  6  7  8  9 
31 30 31 31 30 
```

```
class(tab)
```

```
[1] "table"
```

- Factors: use of labels for labeling

```
is.factor(airquality$Oz2)
```

```
[1] TRUE
```

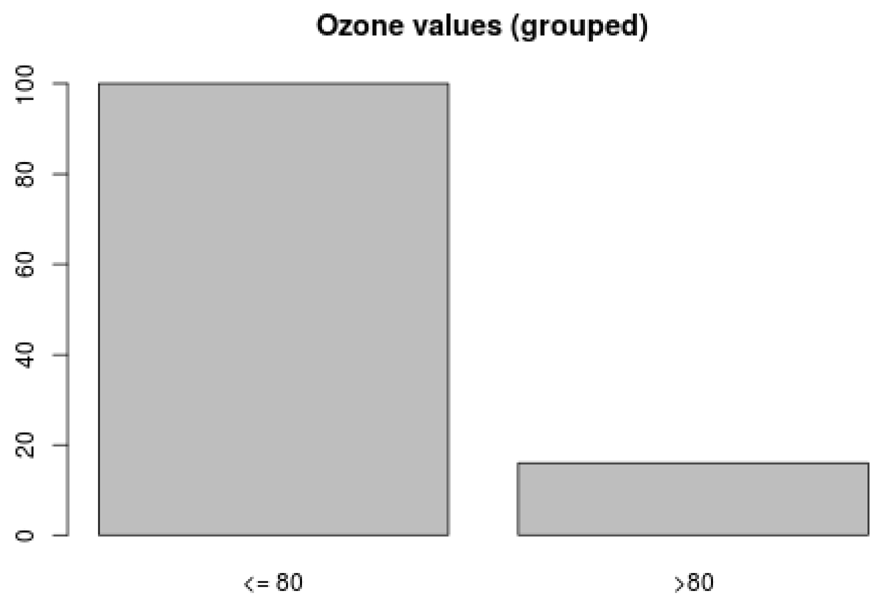
```
table(airquality$Oz2)
```

```

<= 80   >80 
  100    16 
```

Frequency tables (3)

- Tables can be used as input for **barplot()**

```
barplot(table(airquality$Oz2), main="Ozone values (grouped)")
```



Frequency tables (4)

- Arguments of the function **table()**

```
args(table)
```

```
function (..., exclude = if (useNA == "no") c(NA, NaN), useNA = c("no",
    "ifany", "always"), dnn = list.names(...), deparse.level = 1)
NULL
```

- **useNA**: handling missing values
  - *useNA="no"*: missing values are ignored
  - *useNA="ifany"*: NA values are only shown as a separate category if they exist
  - *useNA="always"*: always an additional category for missing values
- **dnn**: optional caption of the table

Frequency tables (5)

- Missing values (**NA**) are ignored by default and are not shown
- By changing the option *useNA* this can be adjusted

```
table(airquality$Oz2)
```

```

<= 80   >80
  100    16
```

```
table(airquality$Oz2, useNA="ifany")
```

```

<= 80   >80  <NA>
  100    16    37
```

Contingency tables (1)

- Currently the tabulation is only for one variable
- Contingency tables are created by using several variables within **table()**

```
table(airquality$Oz2, airquality$Month)
```

```
        
        5  6  7  8  9
  <= 80 25  9 20 19 27
  >80    1  0  6  7  2
```

- Analogously, but with variable names as *labels*

```
table(airquality[,c("Oz2","Month")])
```

```
       Month
Oz2      5  6  7  8  9
  <= 80 25  9 20 19 27
  >80    1  0  6  7  2
```

Contingency tables (2)

- Use of function argument *useNA* analogous to the one-dimensional case

```
# Column for NA only for variable 'Oz2'
table(airquality[,c("Oz2","Month")], useNA="ifany")
```

```
       Month
Oz2      5  6  7  8  9
  <= 80 25  9 20 19 27
  >80    1  0  6  7  2
  <NA>   5 21  5  5  1
```

```
# Columns for NA for both variables
table(airquality[,c("Oz2","Month")], useNA="always")
```

```
       Month
Oz2      5  6  7  8  9 <NA>
  <= 80 25  9 20 19 27    0
  >80    1  0  6  7  2    0
  <NA>   5 21  5  5  1    0
```

Contingency tables (3)

- Extension for additional variables easily possible
- Example: 3-dimensional table

```
# first table: Temp <= 65: months by ozone grous
# second table: Temp > 65: months by ozone grous
table(airquality$Month, airquality$Oz2, airquality$Temp <= 65, useNA="ifany")
```

```
, ,  = FALSE


   <= 80 >80 <NA>
  5    14   1    1
  6     8   0   21
  7    20   6    5
  8    19   7    5
  9    25   2    1

, ,  = TRUE


   <= 80 >80 <NA>
  5    11   0    4
  6     1   0    0
  7     0   0    0
  8     0   0    0
  9     2   0    0
```

Relative frequencies (1)

- **prop.table()** can be used to create tables with relative frequencies
- Necessary inputs:
  - a table (created with **table()**)
  - an optional index that determines the margin for calculating relative frequencies (rows, columns)
- *margin=NULL*: relative to the total frequency
- *margin=1*: relative to the row sums
- *margin=2*: relative to the column sums

Relative frequencies (2)

```
tab <- table(airquality$Oz2, airquality$Month, useNA="ifany")
prop.table(tab, margin=NULL) # tab / sum(tab)
```

```
                  5           6           7           8           9
  <= 80 0.163398693 0.058823529 0.130718954 0.124183007 0.176470588
  >80   0.006535948 0.000000000 0.039215686 0.045751634 0.013071895
  <NA>  0.032679739 0.137254902 0.032679739 0.032679739 0.006535948
```

```
prop.table(tab, margin=1) # tab / rowSums(tab)
```

```
                 5          6          7          8          9
  <= 80 0.25000000 0.09000000 0.20000000 0.19000000 0.27000000
  >80   0.06250000 0.00000000 0.37500000 0.43750000 0.12500000
  <NA>  0.13513514 0.56756757 0.13513514 0.13513514 0.02702703
```

```
prop.table(tab, margin=2) # tab / colSums(tab)
```

```
                 5          6          7          8          9
  <= 80 0.80645161 0.30000000 0.64516129 0.61290323 0.90000000
  >80   0.03225806 0.00000000 0.19354839 0.22580645 0.06666667
  <NA>  0.16129032 0.70000000 0.16129032 0.16129032 0.03333333
```

Marginal sums (1)

- Marginal sums can be calculated with the function **margin.table()**
- Inputs are analogous to **prop.table()**
  - a table (created with **table()**)
  - an optional index that determines the margin for calculating relative frequencies (rows, columns)
- *margin=NULL*: total frequency
- *margin=1*: row sums
- *margin=2*: column sums

Marginal sums (2)

```
tab
```

```

        5  6  7  8  9
  <= 80 25  9 20 19 27
  >80    1  0  6  7  2
  <NA>   5 21  5  5  1
```

```
margin.table(tab, margin=NULL)
```

```
[1] 153
```

```
margin.table(tab, margin=1)
```

```

<= 80   >80  <NA>
  100    16    37
```

```
margin.table(tab, margin=2)
```

```

 5  6  7  8  9
31 30 31 31 30
```

Marginal sums (3)

- The function addmargins() adds marginal sums to a table.

```
tab <- table(airquality$Ozone > 80, airquality$Month, useNA = "ifany")
ptab <- prop.table(tab)
addmargins(tab)
```

```
         5   6   7   8   9 Sum
  FALSE 25   9  20  19  27 100
  TRUE   1   0   6   7   2  16
  <NA>   5  21   5   5   1  37
  Sum   31  30  31  31  30 153
```

```
addmargins(ptab)
```

```
                  5           6           7           8           9
  FALSE 0.163398693 0.058823529 0.130718954 0.124183007 0.176470588
  TRUE  0.006535948 0.000000000 0.039215686 0.045751634 0.013071895
  <NA>  0.032679739 0.137254902 0.032679739 0.032679739 0.006535948
  Sum   0.202614379 0.196078431 0.202614379 0.202614379 0.196078431

                Sum
  FALSE 0.653594771
  TRUE  0.104575163
  <NA>  0.241830065
  Sum   1.000000000
```

Marginal sums (4)

- addmargins() can add any arbitrary function

```
tab <- table(airquality$Ozone > 80, airquality$Month, useNA = "ifany")
addmargins(tab, FUN=median)
```

```
Margins computed over dimensions
in the following order:
1:
2:
```

```
         5  6  7  8  9 median
  FALSE 25  9 20 19 27     20
  TRUE   1  0  6  7  2      2
  <NA>   5 21  5  5  1      5
  median 5  9  6  7  2      6
```

More functions for tabulation (1)

- **ftable()** can be used to generate hierarchically *flat* tables

```
tab <- table(airquality$Oz2, airquality$Month, airquality$Temp<=65)
ftable(tab)
```

```
          FALSE  TRUE

<= 80 5      14    11
      6       8     1
      7      20     0
      8      19     0
      9      25     2
>80   5       1     0
      6       0     0
      7       6     0
      8       7     0
      9       2     0
```

- **xtable()** from package **xtable** is useful to generate tables for use in LaTeX or html.

More functions for tabulation (2)

- **xtabs()** can be used to create tables using *formula interface*
- powerful and flexible, but difficult with missing values

```
tab <- xtabs(~ Oz2 + Month, data=airquality)
```

```
class(tab)
```

```
[1] "xtabs" "table"
```

- functions **addmargins()**, **prop.table()**, or **margin.table()** can be applied to the resulting tables:

```
addmargins(tab)
```

```
       Month
Oz2       5   6   7   8   9 Sum
  <= 80  25   9  20  19  27 100
  >80     1   0   6   7   2  16
  Sum    26   9  26  26  29 116
```

Tasks / Exercises

Time for practical training! :)

Please continue to work on Exercises 1x) and 2x).

Summary (1)

- Frequency tables and contingency tables can be created easily in **R**
- Main function: **table()**
- The function argument *useNA* controls how to deal with missing values
  - *useNA="no"* (default): missing values are ignored
  - *useNA="ifany"*: separate column for missing values (only if they occur)
  - *useNA="always"*: column for missing values is always shown

Summary (2)

- Tables with (conditional) relative frequencies can be generated with **prop.table()**
- Marginal sums can be generated with **margin.table()**
- Marginal sums can be added to a table with **addmargins()**
- More features to create tables (advanced):
  - **ftable()**
  - **xtabs()**

Import / Export

Alexander Kowarik, Bernhard Meindl

Overview/Objectives

- Read and write data from various sources
    - R data format
    - CSV files
    - Excel files
    - Databases (DB2, MySQL)

R as a mediator

- Often a myriad of different statistical programs are used in business enterprises, such as SAS, Stata,EViews, Octave, SPSS, and R.
- Good "communication" between the programs is necessary.
- Dealing with binary formats from SPSS or Excel ...
- R offers very good interfaces to other software.
- Simple text files are most convenient to read, but also XML files and many other file formats can be imported (exported).
- For large amounts of data, a connection to databases is recommended. In the following we list the most popular methods for data import and export.

Data existing already in R

- Many different data sets are already existing in **R**
  - Access by means of the function **data**
  - Example: **data(mtcars)**
  - Or: **data(solder, package="rpart")**
- Very handy for practice; often used in the examples.

R data format (1)

- R has its own binary data format.
- Usually, the extension .RData is used.
- By default, the data are also compressed.
- For saving the data, the function save() is used.

```
save(mtcars, file="mtcarsTest.RData")
```

- Several objects can be stored at once:

```
save(mtcars, iris, file="mtcarsIrisTest.RData")
```

R data format (2)

- To save all objects from the workspace:

```r
ls() # lists all objects that will be contained in 'RSession.RData'
save.image(file="RSesssion.RData")
```

- To load data, the function load used.
- load automatically detects whether the file is compressed and uses decompression.

```r
load(file="mtcarsTest.RData")
```

R paths

- *Note:* paths in the Unix style can also be used in Windows
- *Example:*
    - C:/path/to/xy.RData
    - or C:\\path\\to\\xy.RData
    - but NOT C:\path\to\xy.RData
- The function file.path() can create platform independent paths.

Data from text files or CSV files

- To import rectangular data, frequently **read.table()** is used
- Highly detailed possibilities for parameter choices, see help with **?read.table**

```
read.table(file, header=FALSE, sep="", quote = "\"'", dec=".", row.names, col.names, as.is=FALSE,
na.strings="NA" , colClasses=NA, nrows =-1, skip = 0, check.names = TRUE, fill =!blank.lines.skip, strip.white
= FALSE, blank.lines.skip = TRUE, comment.char="#" )
```

- **read.table()** is extremely flexible
- reasonable default values exist

Data from text files or CSV files (2)

- Important *arguments* of **read.table()** are:
  - *col.names*: optional vector of variable names
  - *row.names*: optional vector of row labels, often the first column
  - *header*: Does the first line contain the variable names?
  - *sep*: Which column separator? Often ; or ,

Data from text files or CSV files (3)

- **skip**: from which row of data start to read?
- **nrows**: how many rows of data to be read?
- **na.strings**: how are missing values encoded?
- **as.is**: the default converts all variables to factors.

- *Special cases:* **read.csv()**, **read.csv2()**, **read.delim()** or **read.delim2()**

- Additional information: *R Data Import/Export Manual*

Other functions for data import

- readLines() reads line by line
- scan() is a workhorse for read.table()
- readBin() used to read binaries byte by byte
- read.fwf() for data with a fixed format (e.g. host files)
- read.csv() (English) and read.csv2() (German) for .csv files
- *Furthermore*: many functions for reading databases or URL's exist in different R packages (e.g. readr)

URL’s

- Direct importing from URL's is possible

```
dat <- read.csv2("http://data.statistik.gv.at/data/OGD_f1197_Bev_Jahresdurchschn_1.csv")
head(dat)
```

```
   C.A10.0 C.B00.0 C.C11.0 F.ISIS.568
1 A10-1982   B00-1   C11-1     129951
2 A10-1982   B00-1   C11-2     140571
3 A10-1982   B00-1   Total     270522
4 A10-1982   B00-2   C11-1     258738
5 A10-1982   B00-2   C11-2     279179
6 A10-1982   B00-2   Total     537917
```

Excel

- There are two important packages: xlsx and XLConnect
  - xlsx: The functions read.xlsx() and write.xlsx() can read “beautiful” Excel files with rectangular data in an entire worksheet
  - XLConnect: greater flexibility in reading and writing
- Warning: formulas, links to other Excel files, hidden table parts, etc. can lead to unpredictable problems accessing an Excel file.
- *Database connectivity* or at least csv files are to be preferred if possible.

Excel (2)

- Package **xlsx**

```
write.xlsx(mtcars, file="text.xlsx")
read.xlsx(file="text.xlsx", sheetIndex=1)
```

- Package **XLConnect**

```
demoExcelFile <- system.file("demoFiles/mtcars.xlsx", package="XLConnect")
wb <- loadWorkbook(demoExcelFile)
data <- readTable(wb, sheet="mtcars_table", table="MtcarsTable")
```

Databases

- *ODBC* (Open Database Connectivity) or *JDBC* (Java Database Connectivity) support popular databases and formats:
  - R packages **RODBC** and **RJDBC**
  - *ODBC* or *JDBC* drivers must be installed on the system.
- Additionally, there are packages for individual database systems:
  - MySQL (Package **RMySQL**)
  - SQLite (Package **RSQLite**)
  - Oracle (Package **ROracle**)
  - ...

Databases

- read from mysql-database

```r
require(RMySQL)
con <- dbConnect(RMySQL::MySQL(), host="osrp01",
  user="validUser", password=readline("passwd:\n"))
dat <- dbReadTable(con, "myTable")
dbDisconnect(con)
```

- read from DB2 using *RODBC*

```r
require(RODBC)
odbcDataSources()
con <- odbcConnect("ATSTZDB2", uid="filz$", pwd="pwd")
sqlTables(con) # show existing tables
dim(a)
head(a[a$TABLE_SCHEM=="UBR",])
# run a query
dat1 <- sqlQuery(con,
  "SELECT * from UBR.THVVV where YEAR='2012' FETCH FIRST 100 ROWS ONLY;", errors = TRUE)
dbDisconnect(con)
```

Export (1)

- **cat()** allows to write output to the console or into a text file
- To the console:

```r
a <- 1:10
cat(a, "\n", sep = "-")
```

```
1-2-3-4-5-6-7-8-9-10-
```

- Redirect output to a file:

```r
cat(1:10, "\n", sep = "-",file = "log.txt")
```

- *Note*: **sink()** allows to redirect all output (e.g. into a text file)

Export (2)

- **write.table()** is the counterpart to **read.table()**
- The arguments are similar and there is again **write.csv()** and **write.csv2()**

```
write.table(x, file = "", append = FALSE, quote = TRUE,sep = " ", eol = "\n", na = "NA", dec = ".", row.names =
TRUE, col.names = TRUE)
```

Tasks / Exercises

Time for practical training! :)

Please continue to work on Exercises 1x) to 5x).

Summary (1)

- R is very flexible, thus it is often used as *Mediator*
- Data can be easily imported from and exported to in various formats
- R has its own binary file format
- We can read from and write to (relational) databases
- Many additional packages to get data in and out are available

Mini-introduction graphics

Alexander Kowarik, Bernhard Meindl

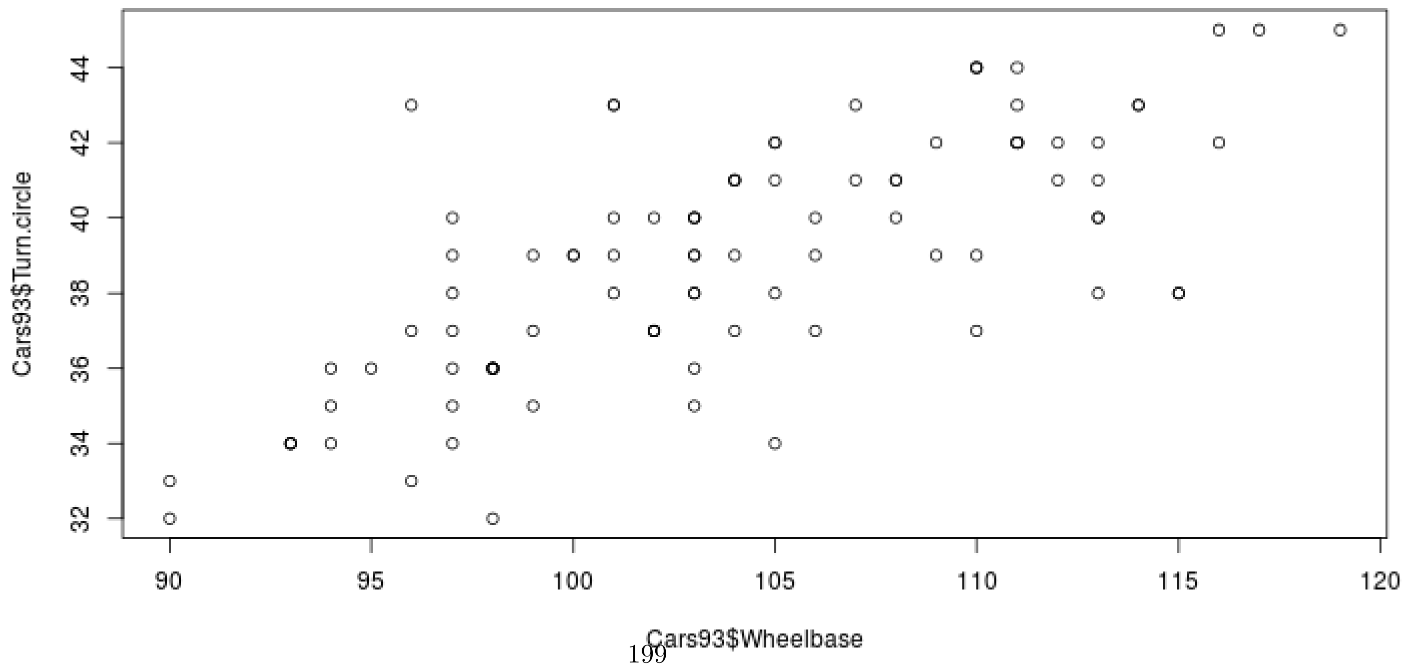
Overview / Objectives

- Scatterplot
- Histogram
- Boxplot

Scatterplot

```
plot(x, y)
```

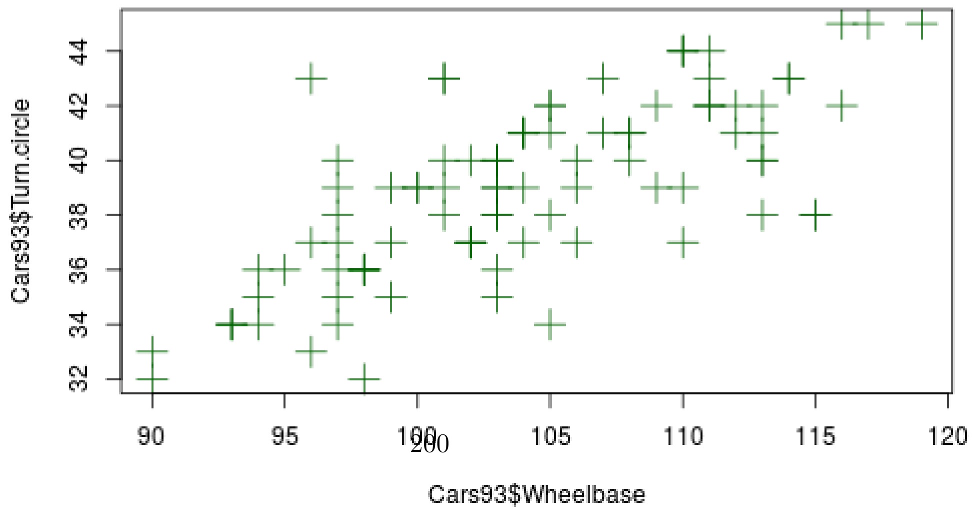
```
data(Cars93, package="MASS")
plot(Cars93$Wheelbase, Cars93$Turn.circle)
```



Scatterplot (2)

- Parameters:
  - Plotsymbol: *pch*
  - Color: *col*
  - Size of plot symbols: *cex*

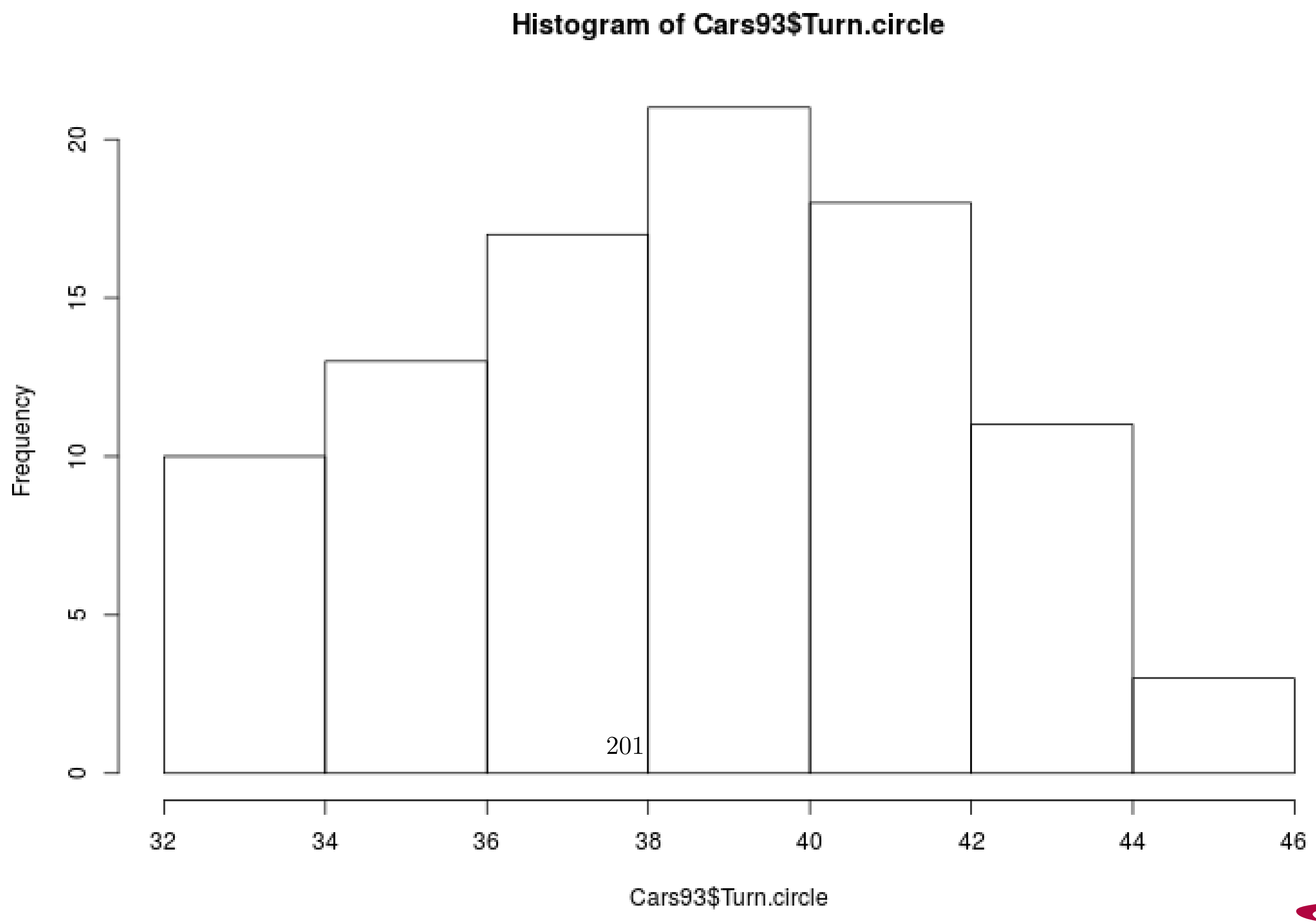
```
plot(Cars93$Turn.circle ~ Cars93$Wheelbase, pch=3, col="darkgreen", cex=2)
```



Histogram

- With absolute frequencies

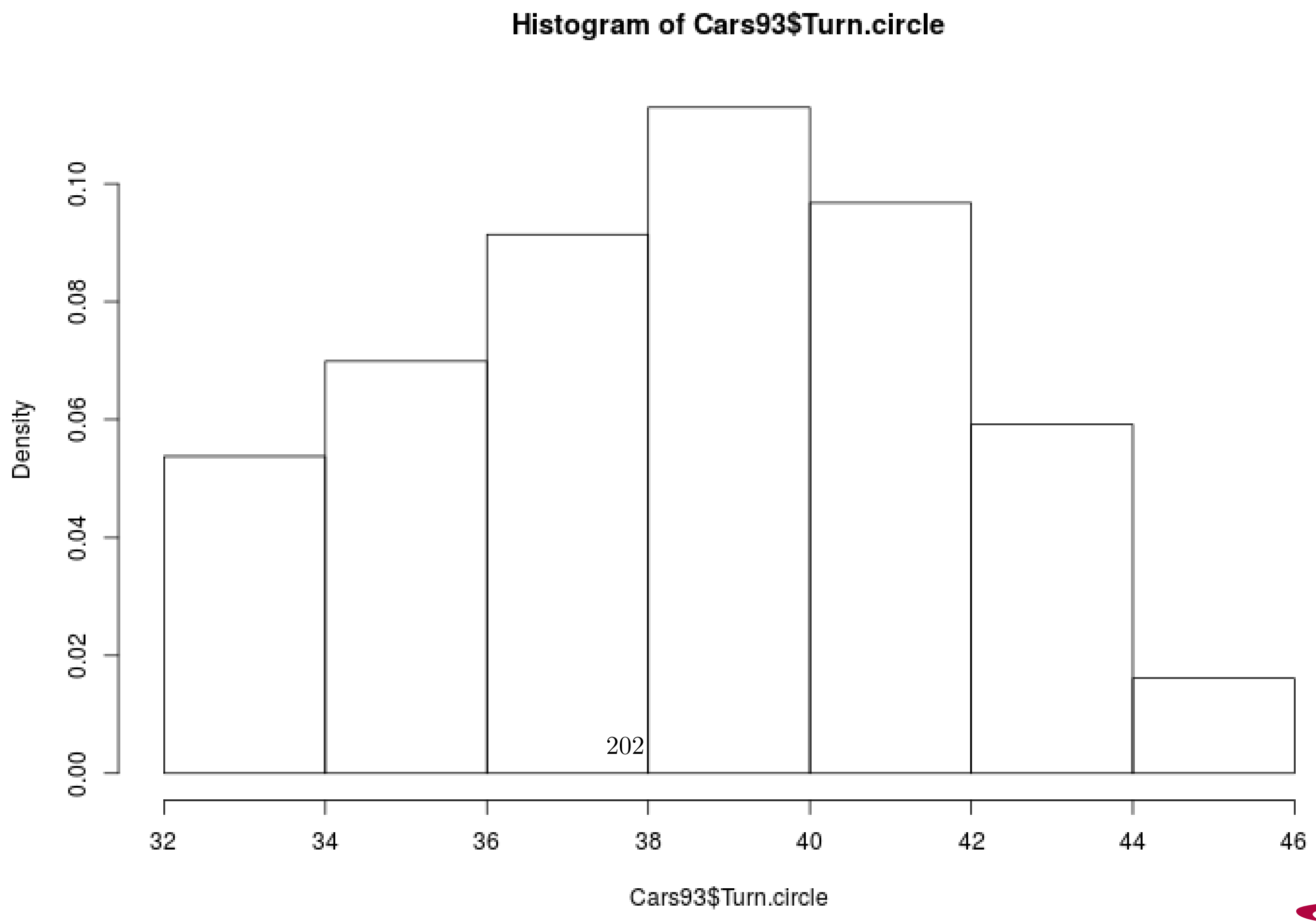
```
hist(Cars93$Turn.circle)
```



Histogram (2)

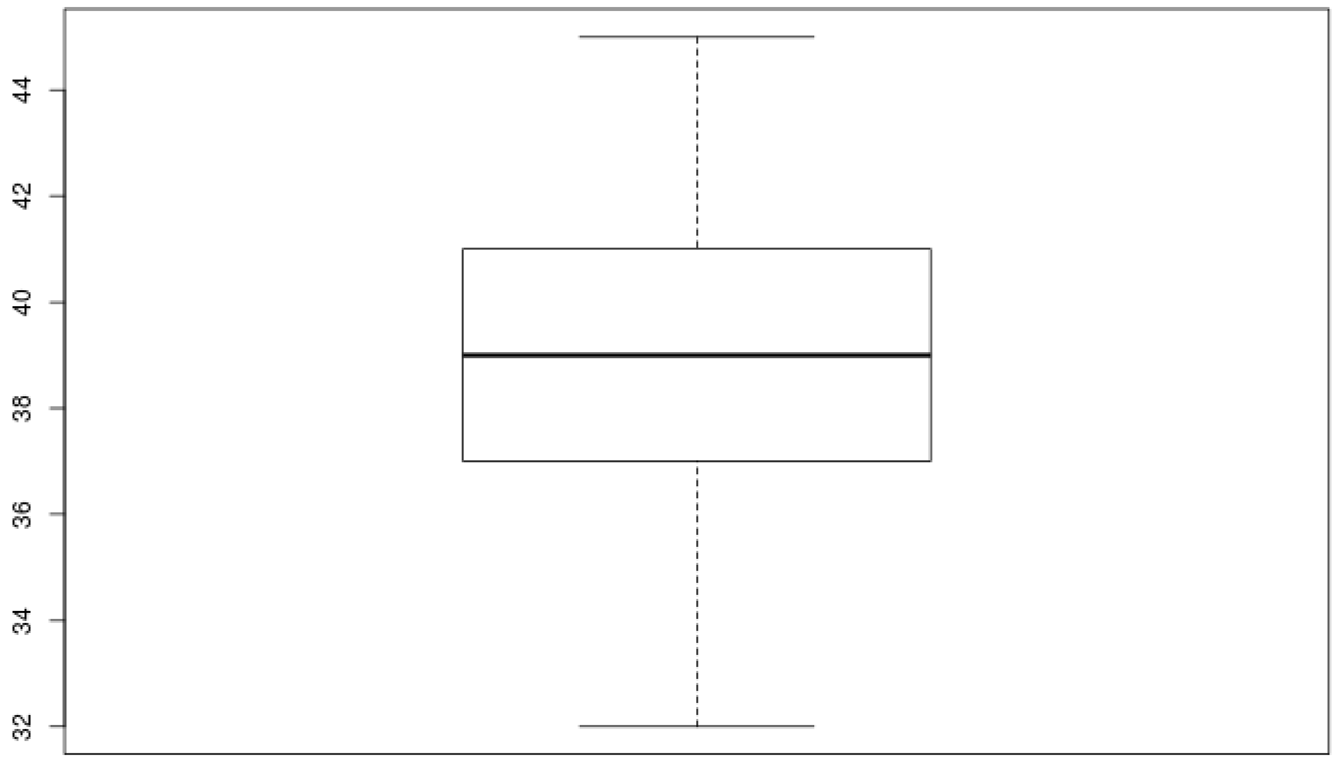
- With relative frequencies

```
hist(Cars93$Turn.circle, probability=TRUE)
```



Boxplot

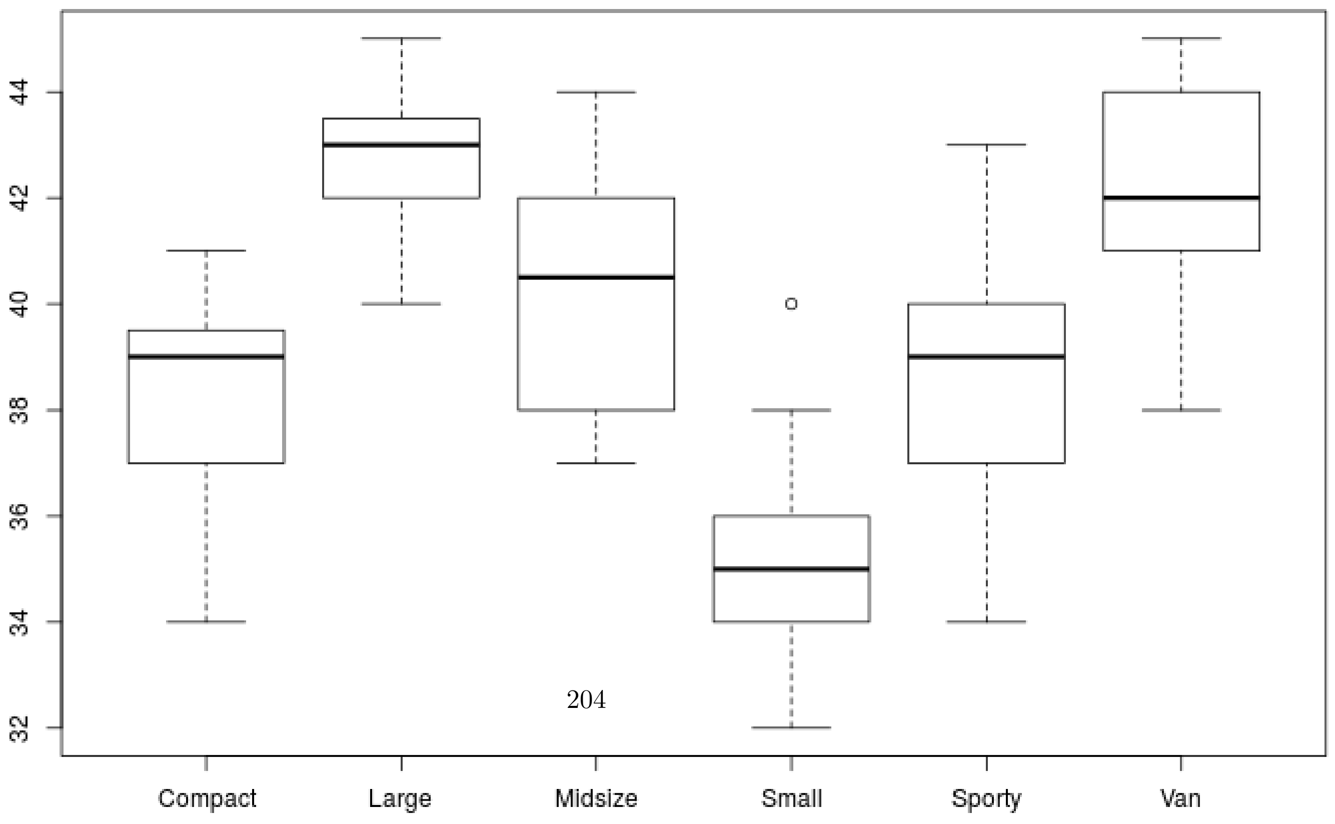
```
boxplot(Cars93$Turn.circle)
```



Boxplot (2)

- By group (factor variable!)

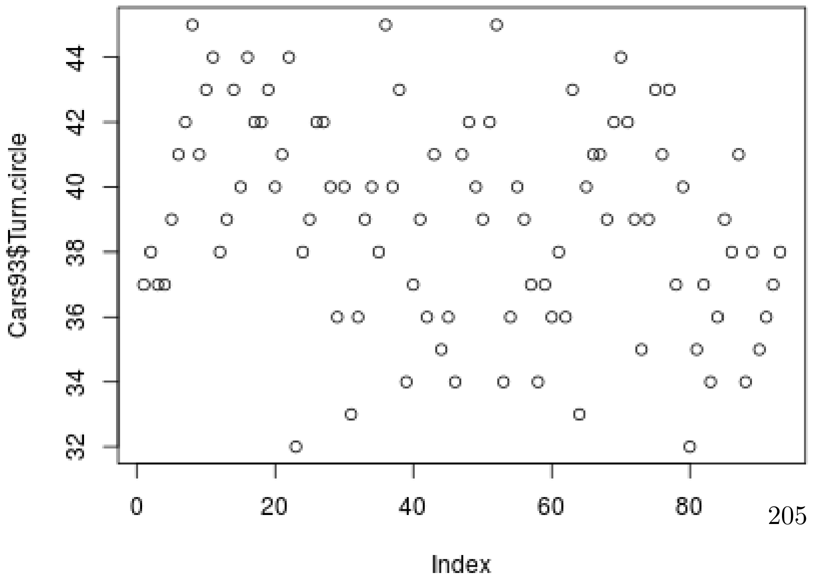
```
boxplot(Cars93$Turn.circle ~ Cars93$Type)
```

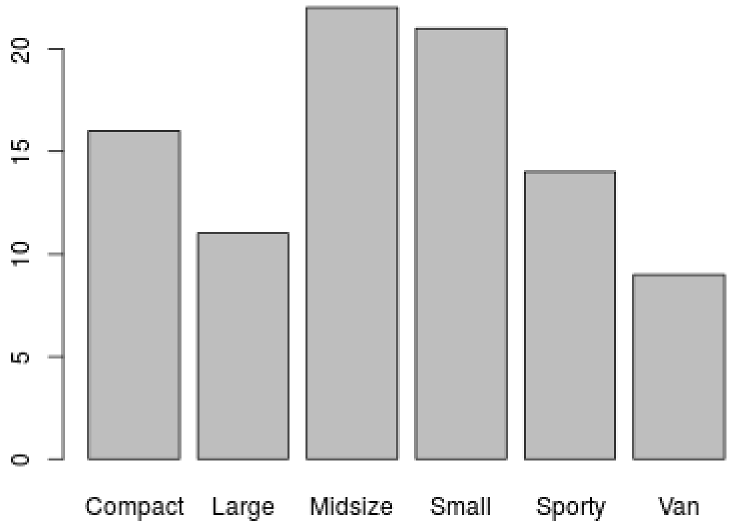


Plot methods

- Function **plot()** returns results depending on the input
- **par(mfrow = c (Z, S))** is an easy way to integrate multiple plots in one plot (Z = lines in the plot, S = columns in the plot)

```
par(mfrow = c(1,2))
plot(Cars93$Turn.circle); plot(Cars93$Type)
```





Tasks / Exercises

Time for practical training! :)

Please continue to work on Exercises 1x) and 2x).

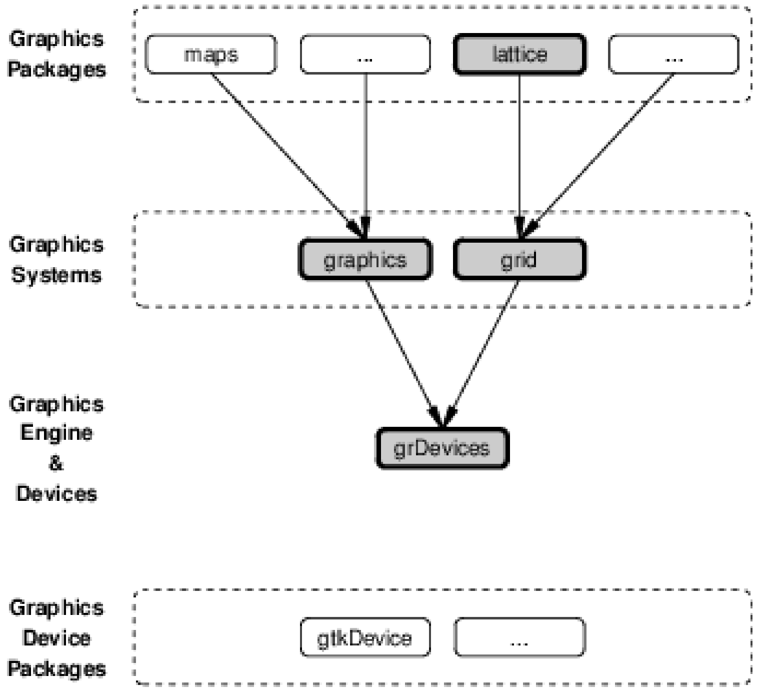
The package graphics

Alexander Kowarik, Bernhard Meindl

Overview/Aim

- Gain knowledge of basic R Graphics
  - different output formats
  - the traditional graphics system
  - adaptation of standard graphics
- **Note:** other packages such as **gglot2**, **ggvis**, **lattice** oder **grid** are not covered now

Basics of R Graphics



Types of R graphics functions

- **High-level** functions create complete graphs (eg. **plot()**)
- **Low-level** functions add output to existing graphs (eg. **points()**)
- **Interactive** functions allow interaction with graphs (eg. **identify()**)

Typically: multiple functions are combined to create a plot

Different output formats (1)

- Each **graphics device** can be thought of as (abstract) **sheet of paper**
- Draw with **many** pens in many colors
- **No eraser** (except in **ggplot2**)
- Multiple devices can simultaneously be open
- Only in one (the 'active') Graphic Device can be drawn
- A device is (almost completely) hidden from the user

Different output formats (2)

- No difference if we plot on the screen or e.g. into a PDF
- the current state of a device can be stored and copied to other devices
- Common devices include: **X11()**, **pdf()**, **postscript()**, **png()**, **jpg()** or **svg()**
- Example: save plot into a **pdf**:

```
data(mtcars)
pdf(file="myPlot.pdf")
plot(mpg, hp)
dev.off()
```

Different output formats (3)

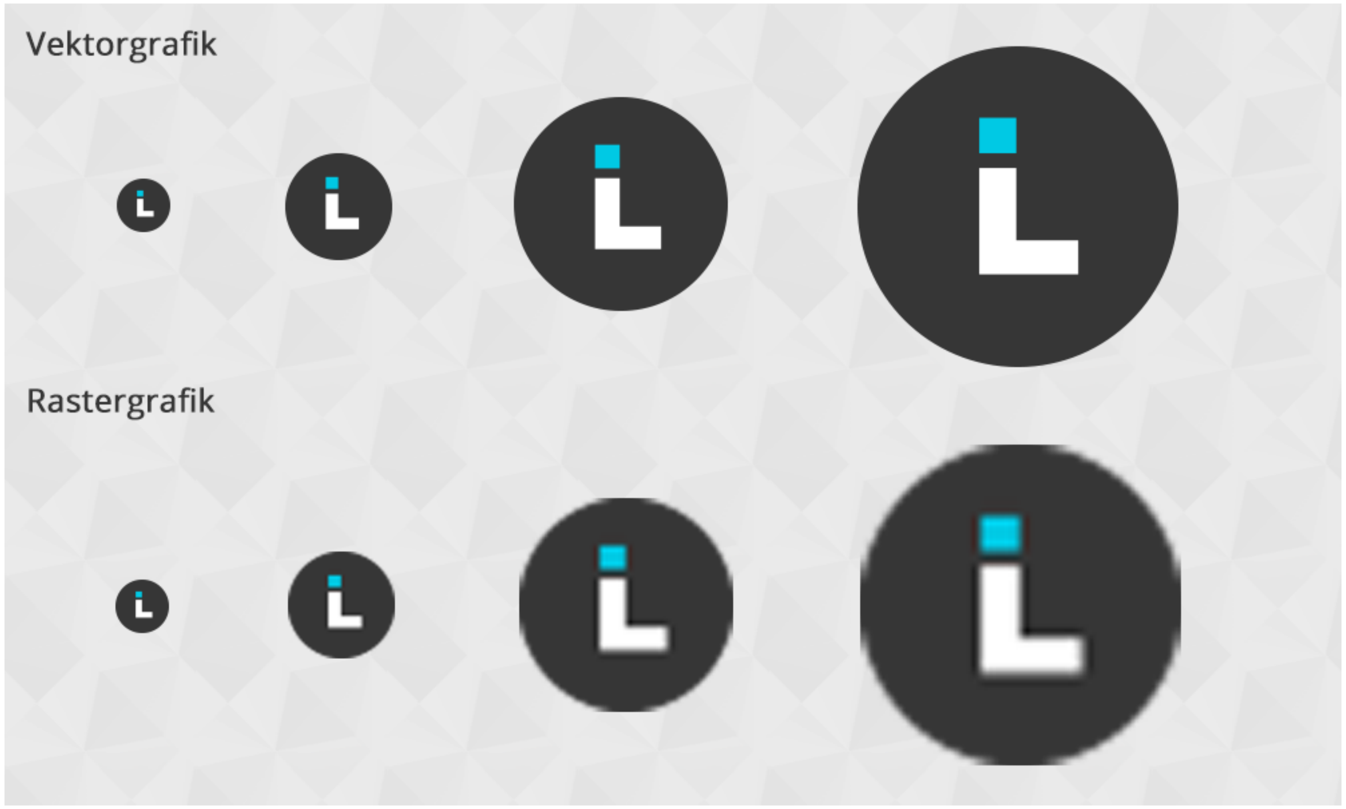
- Screen Devices:
  - **X11()**: X Windows window
  - **Windows()**: Microsoft Windows window
- File Devices (uva.):
  - **postscript()**: PostScript format
  - **pdf()**: PDF format
  - **jpeg()**: JPEG bitmap format
  - **svg()**: Scalable Vector Graphics
  - **cairo()**: Cairo-based graphics device - own graphic library to generate PDF, PostScript, SVG, or bitmap output (PNG, JPEG, TIFF), and X11.
- Use function arguments like **width**, **height**, **quality**, ….

Which output format should be used? (1)

- **X11** for displaying the image (automatically with RStudio)
- **pdf** (or postscript) for line graphics
- **png** (or jpg) for pixel graphics or graphics with many data points

Which output format should be used? (2)

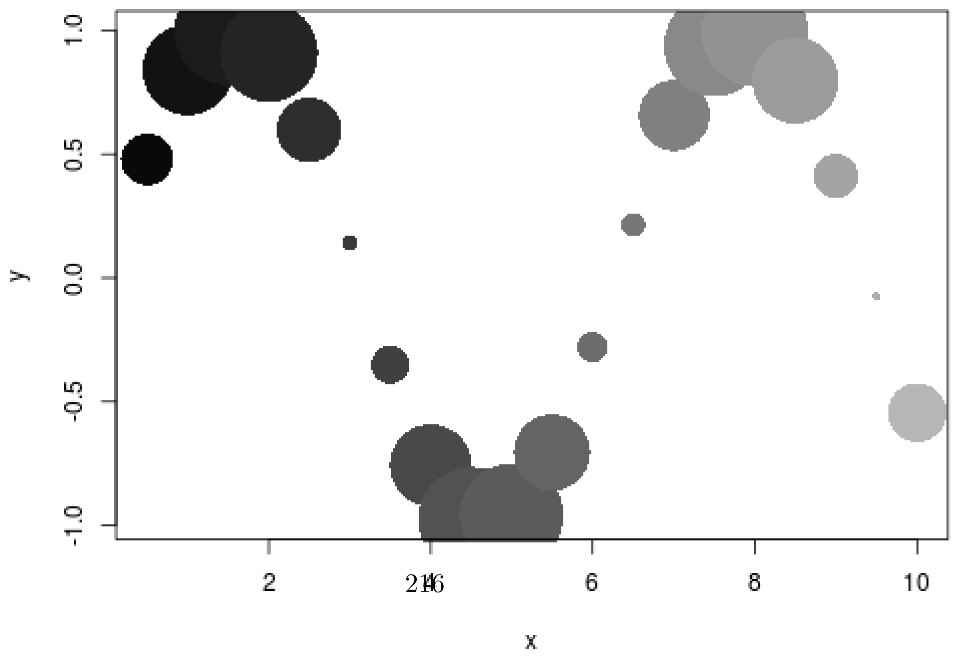
- svg has advantages in the browser (scalable, responsive important in web design!)



Package graphics

- Is the traditional graphics system.
- Warm-up example using a high-level graphic function:

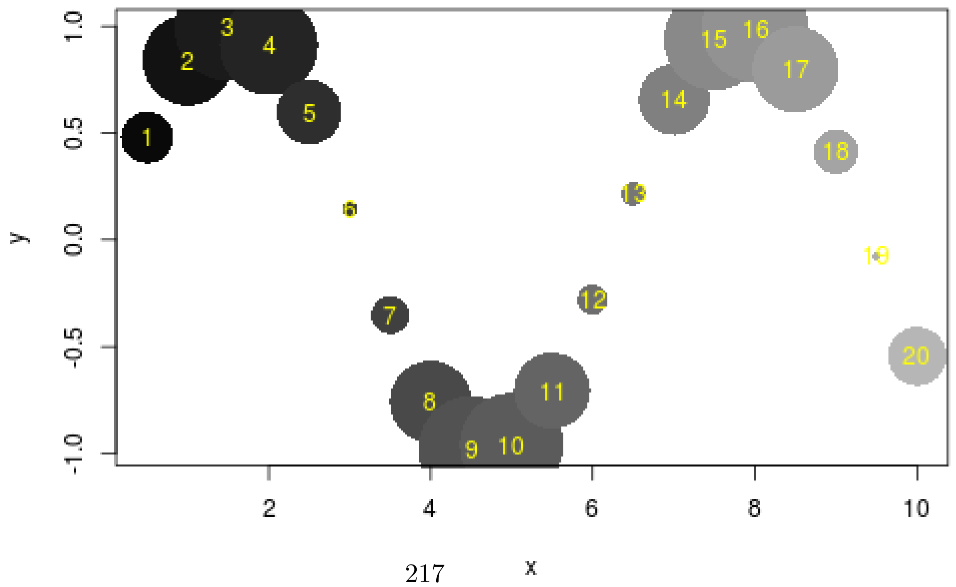
```
x <- 1:20/2; y <- sin(x)
plot(x, y, pch=16, cex=10*abs(y), col=grey(x/14))
```



Package graphics

Adding low-level graphics:

```
plot(x, y, pch=16, cex=10*abs(y), col=grey(x/14))
text(x,y, 1:20, col="yellow")
```

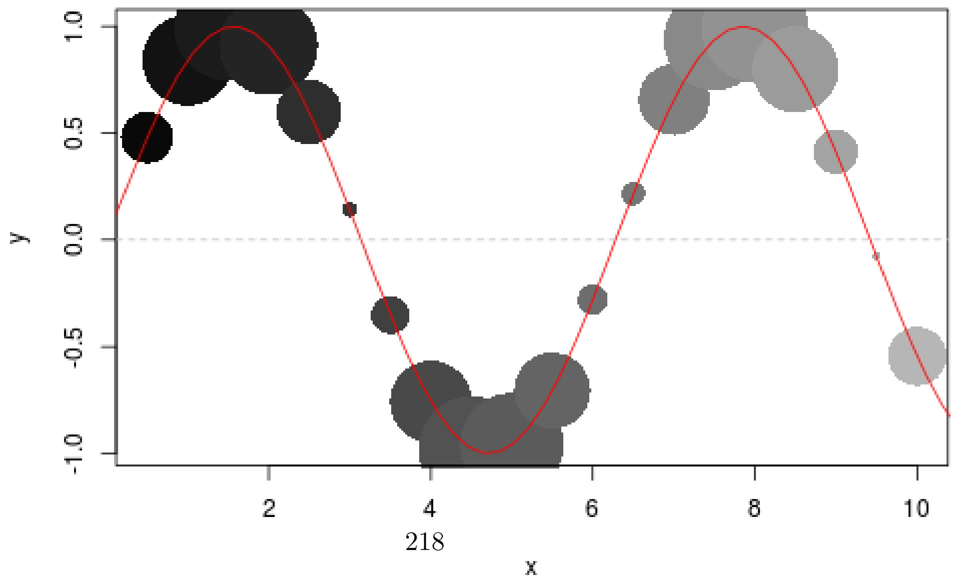


Eurostat

Package graphics

Adding low-level graphics:

```
plot(x, y, pch=16, cex=10*abs(y), col=grey(x/14))
curve(sin, -2*pi, 4*pi, add=TRUE, col="red")
abline(h=0, lty=2, col="grey")
```



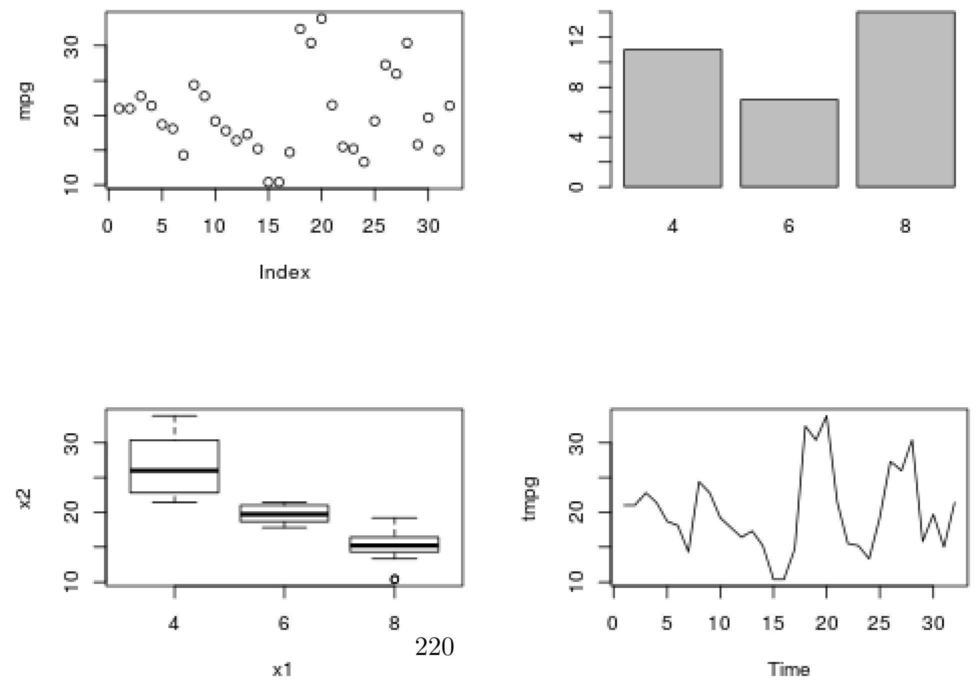
Generic Functions

Example **plot()**

- Is a **generic function**
- Function overloading and **method dispatch**
- Shows different output depending on the **class** of the object to be plotted

Generic Functions

```
par(mfrow=c(2,2))
mpg <- mtcars$mpg
cyl<- factor(mtcars$cyl)
df <- data.frame(x1=cyl, x2=mpg)
tmpg <- ts(mpg)
plot(mpg); plot(cyl); plot(df); plot(tmpg)
```



Generic Functions

Which plot methods are currently available?

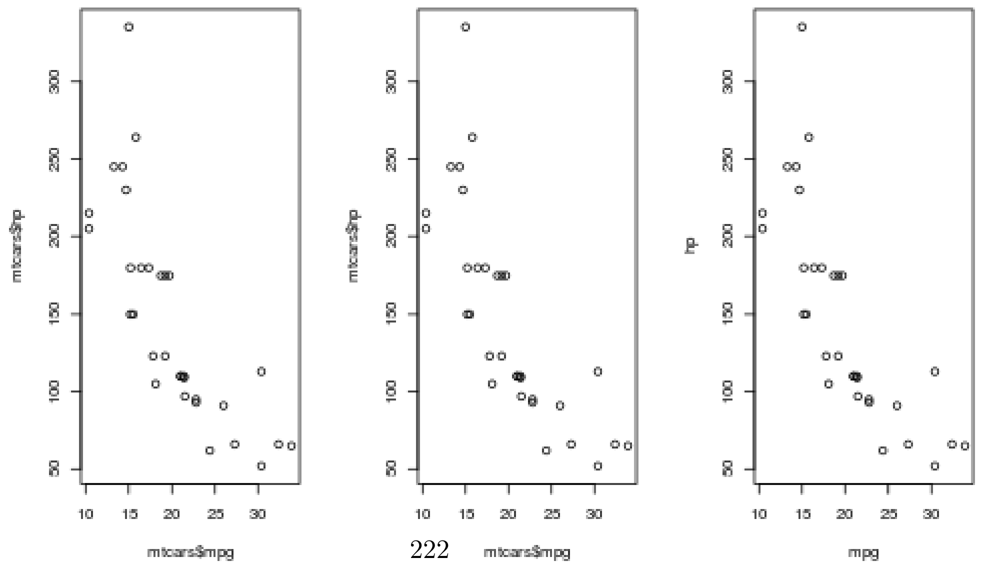
```
methods(plot)
```

```
 [1] plot.acf*           plot.data.frame*    plot.decomposed.ts*
 [4] plot.default        plot.dendrogram*    plot.density*
 [7] plot.ecdf           plot.factor*        plot.formula*
[10] plot.function       plot.hclust*        plot.histogram*
[13] plot.HoltWinters*   plot.isoreg*        plot.lm*
[16] plot.medpolish*     plot.mlm*           plot.ppr*
[19] plot.prcomp*        plot.princomp*      plot.profile.nls*
[22] plot.raster*        plot.spec*          plot.stepfun
[25] plot.stl*           plot.table*         plot.ts
[28] plot.tskernel*      plot.TukeyHSD*
see '?methods' for accessing help and source code
```

Generic Functions

Subsequent calls produce (almost) equivalent results

```
par(mfrow=c(1,3))
plot(x=mtcars$mpg, y=mtcars$hp)
plot(mtcars$mpg, mtcars$hp)
plot(hp ~ mpg, data=mtcars)
```



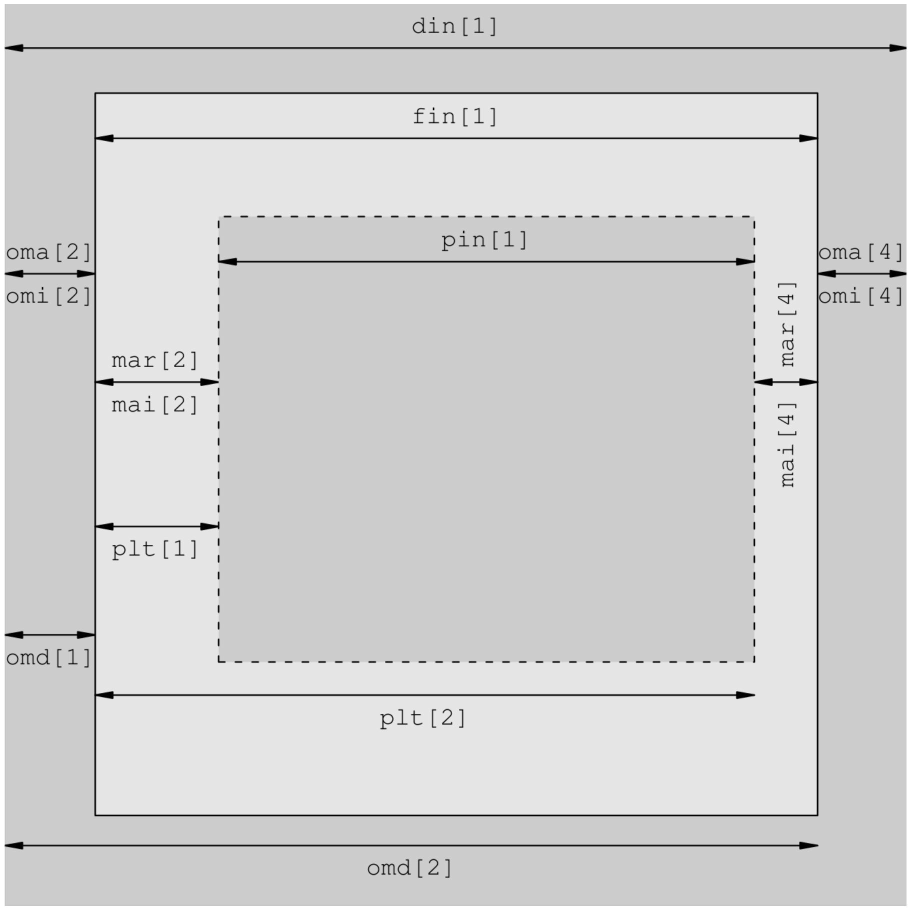
```
par(mfrow=c(1,1))
```

Control of graphics parameters

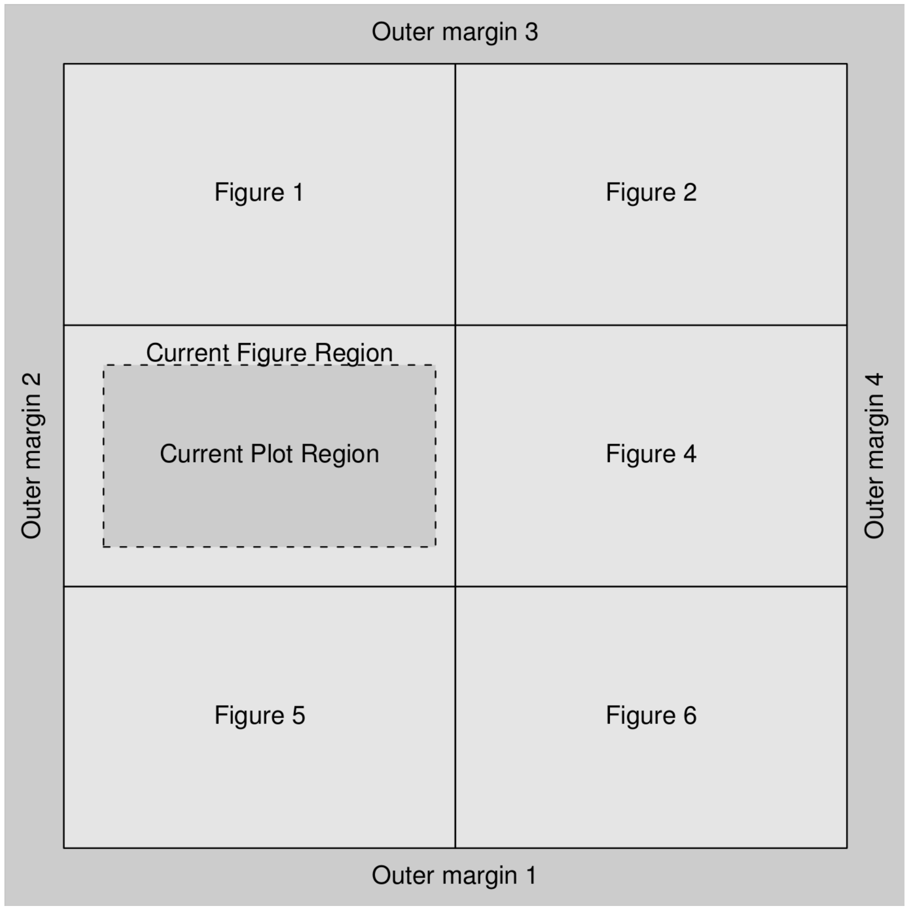
Customizing graphics and change the default output is almost always necessary:

- High-level plot functions do not always produce the desired final result
- Functionality for fine tuning of graphics is necessary (colors, icons, fonts, line widths, ...)
- You basically need information about the plot regions and coordinate system to place output of low-level functions
- Multiple graphs on a page.

Control of graphics parameters



Control of graphics parameters



Control of graphics parameters

- **Graphical parameters** are the key to change the appearance of graphics
- Including, for example ...
  - Colors
  - Fonts
  - Linetypes
  - Axis definitions
- All open devices have their own independent list of graphics parameters
- Most parameters can directly specified in high- or low-level plotting functions
- All graphic parameters can be set via function **par**

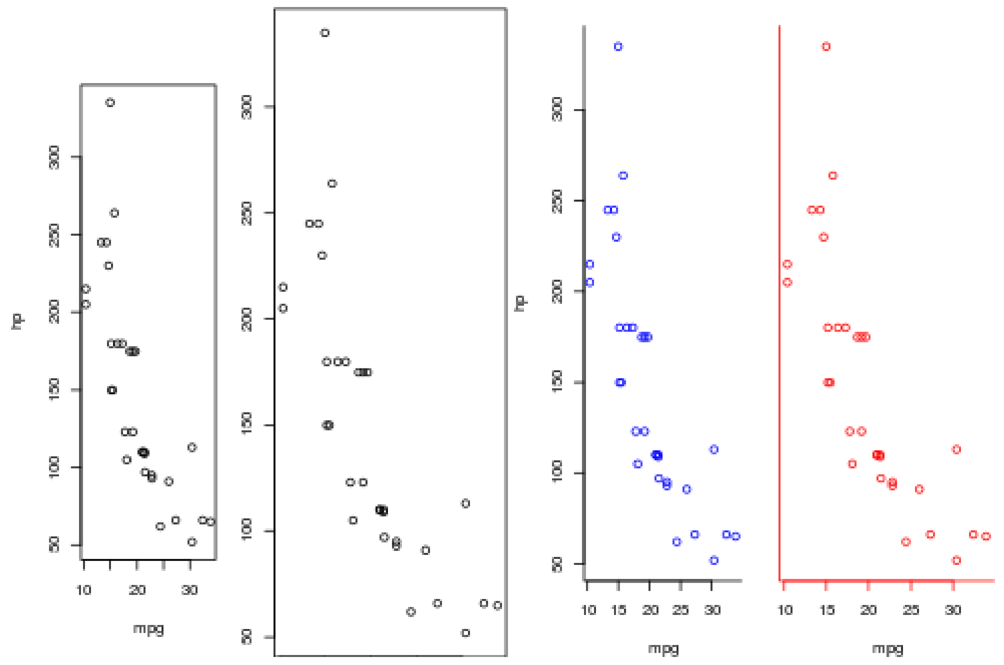
List of parameters: par()

```r
par()[1:8] # ?par
```

```
$xlog
[1] FALSE

$ylog
[1] FALSE

$adj
[1] 0.5

$ann
[1] TRUE

$ask
[1] FALSE

$bg
[1] "white"

$bty
[1] "o"

$cex
[1] 1
```

par() - Change the margins

```
par(mfrow=c(1,4))
plot(mtcars[,c("mpg","hp")])
par(mar=c(0,1,0.1,0.1))
plot(mtcars[,c("mpg","hp")])
par(mar=c(4,4,1,1), bty="l")
plot(mtcars[,c("mpg","hp")], col="blue")
par(mar=c(4,1,1,1), bty="l", yaxt="n", fg="red", ylab="")
plot(mtcars[,c("mpg","hp")])
```



Eurostat

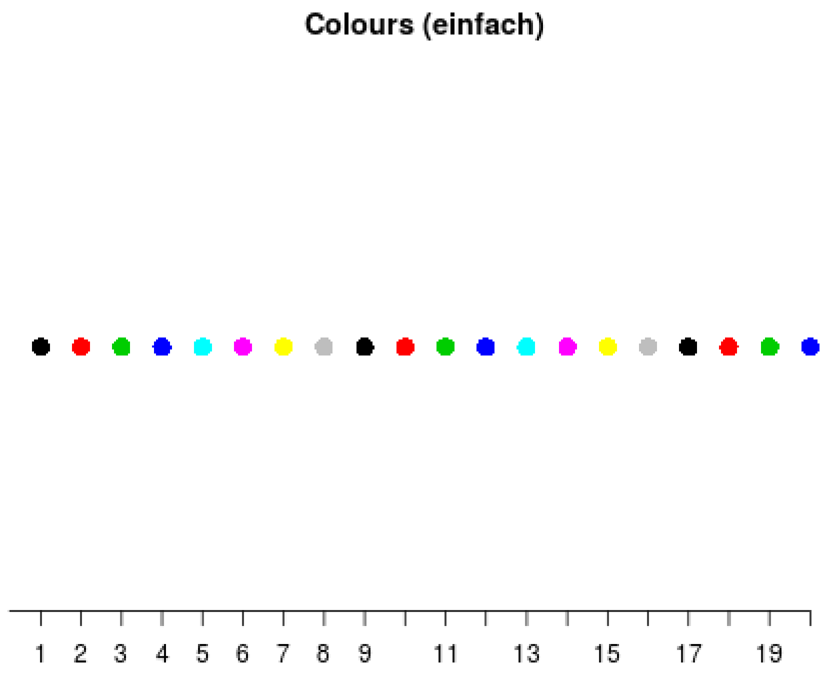
Control of colors

Many roads lead to Rome:

- In R you can default address colours by name via colors().
- rgb() to mix red-green-blue. A better alternative is hsv()
- Pre-defined set of palettes with rainbow colors and many others, eg, ?rainbow
- Predefined set of palettes with palette(). Better alternative are available in the RColorBrewer package.

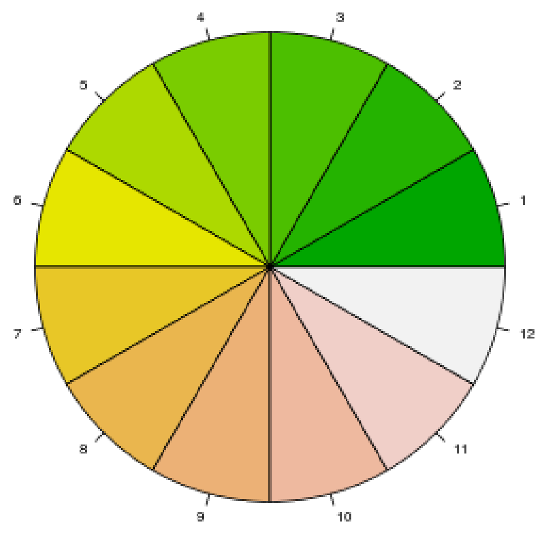
Colors

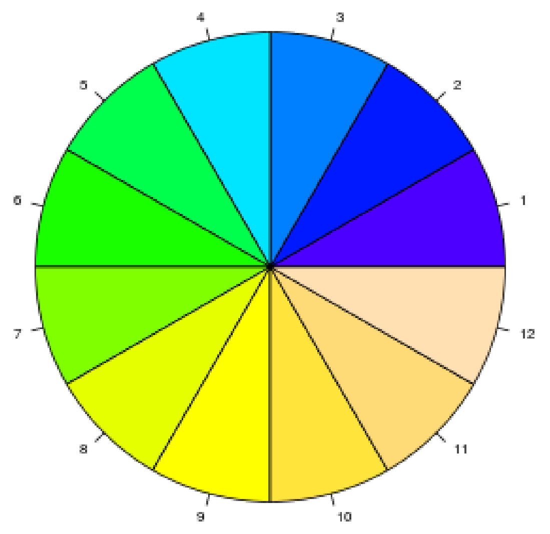
```r
plot(1:20,rep(0,20), pch=16, cex=1.7, col=1:20, xlab="", ylab="", axes=FALSE, main="Colours (einfach)")
axis(side=1, at=0:20, lab=0:20, cex=0.8)
```

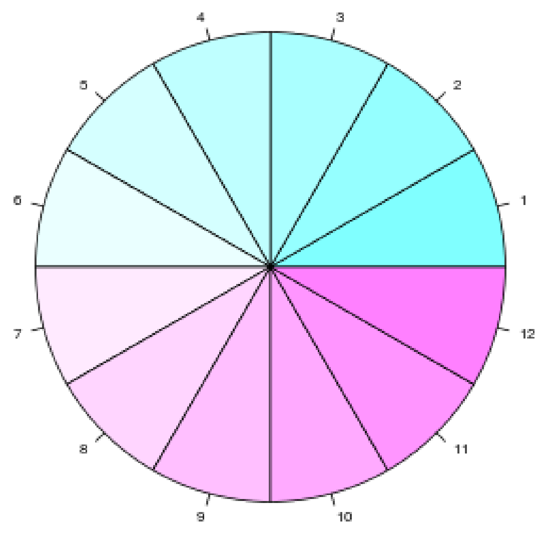


Colors

```r
par(mfrow=c(1,3), mar=c(0,0,0,0))
pie(rep(1,12), col=terrain.colors(12))
pie(rep(1,12), col=topo.colors(12))
pie(rep(1,12), col=cm.colors(12))
```

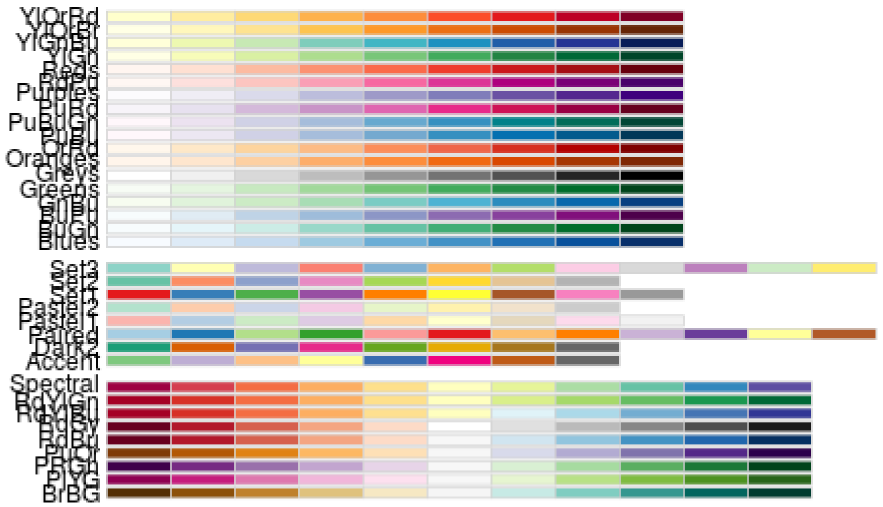






Palettes with RColorBrewer

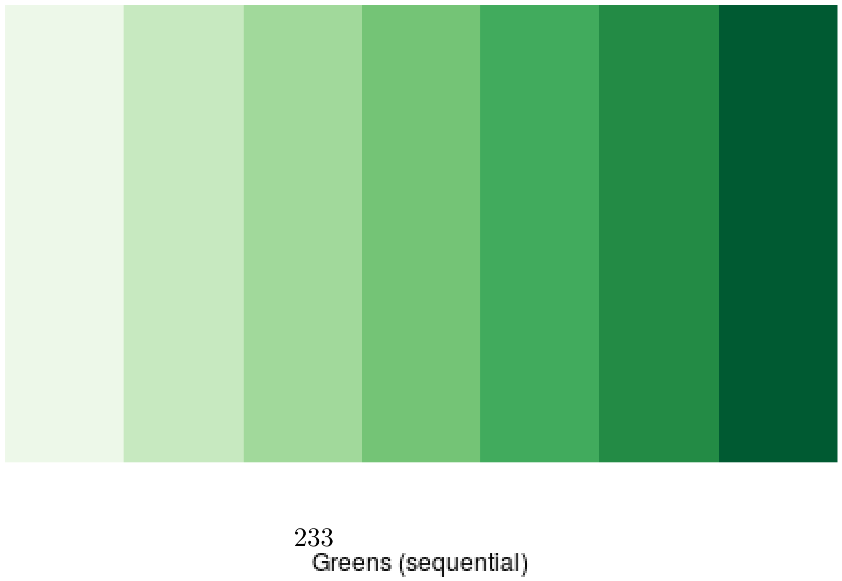
```r
require(RColorBrewer)
display.brewer.all()
```



Palettes with RColorBrewer

Range with brewer.pal()

```
mypalette <- brewer.pal(7, "Greens")
image(1:7, 1, as.matrix(1:7), col=mypalette, xlab="Greens (sequential)",
  ylab="", xaxt="n", yaxt="n", bty="n")
```

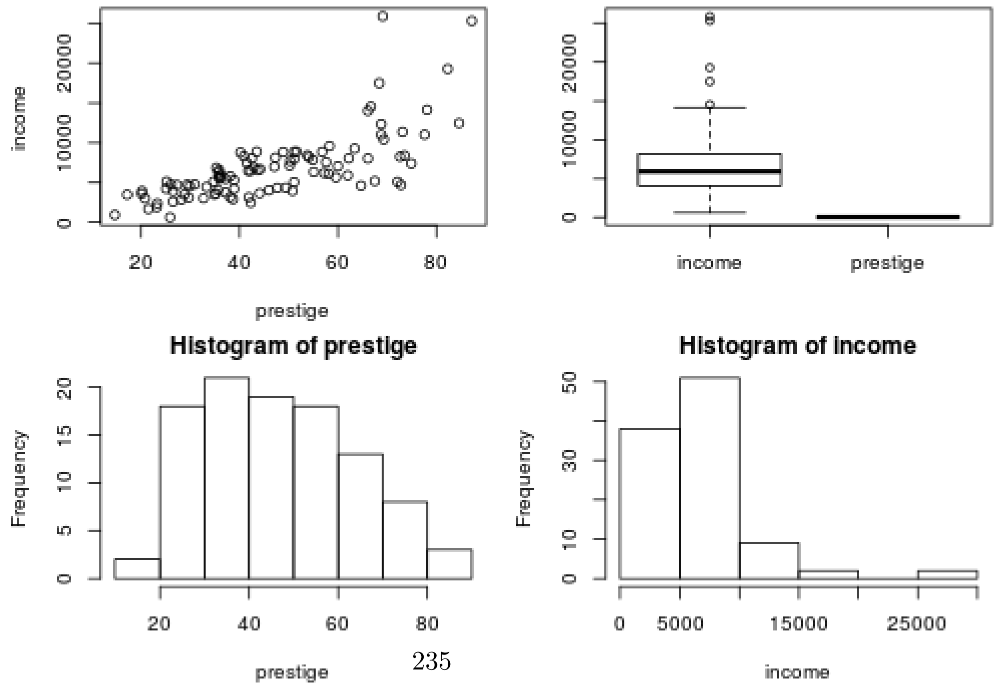


Multiple plots with package graphics

- The easiest with mfrow: par(mfrow=c (2,2))
- Better: layout(), for example,

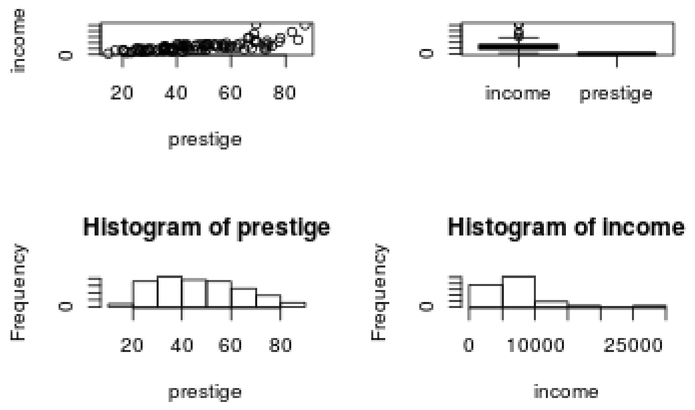
Changing parameters with par()

```
pf <- function() {
  data(Prestige, package="car"); attach(Prestige)
  plot(income ~ prestige); boxplot(Prestige[,c("income","prestige")])
  hist(prestige); hist(income)
  detach(Prestige)
}
par(mfrow=c(2,2), mar=c(4,4,2,1)); pf()
```



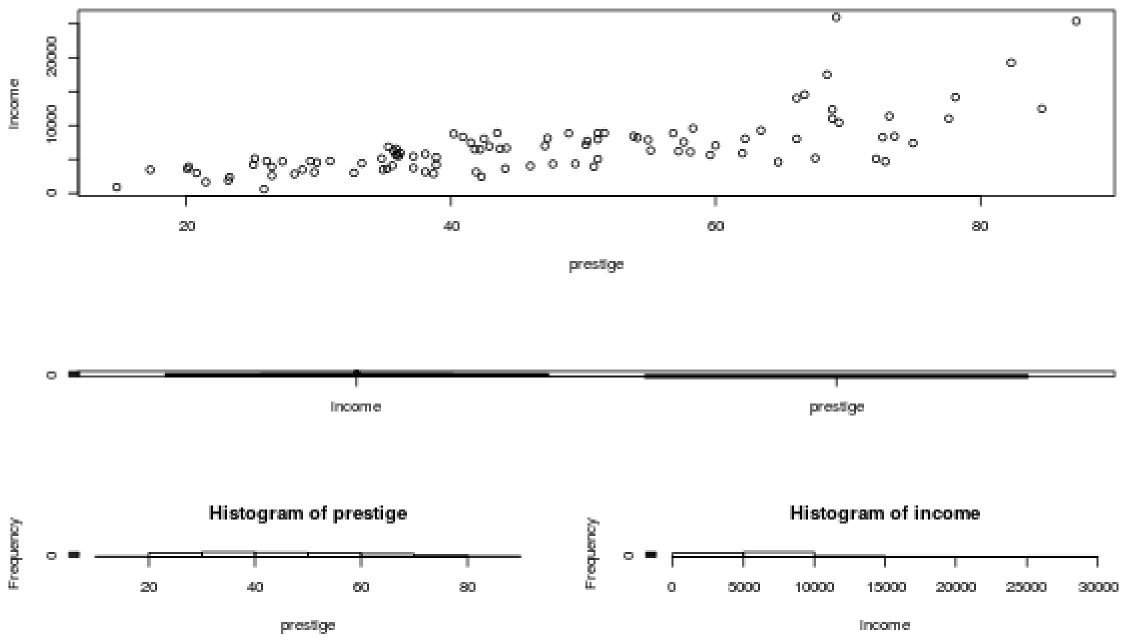
Changing parameters with par()

```
par(mfrow=c(2,2), omi=c(1,1,1,1))
pf()
```



Using layout() (1)

```
layout(matrix(c(1,1,1,1,2,2,3,4), 4, 2, byrow = TRUE))
pf()
```



Using layout() (2)

- plotting schedule:

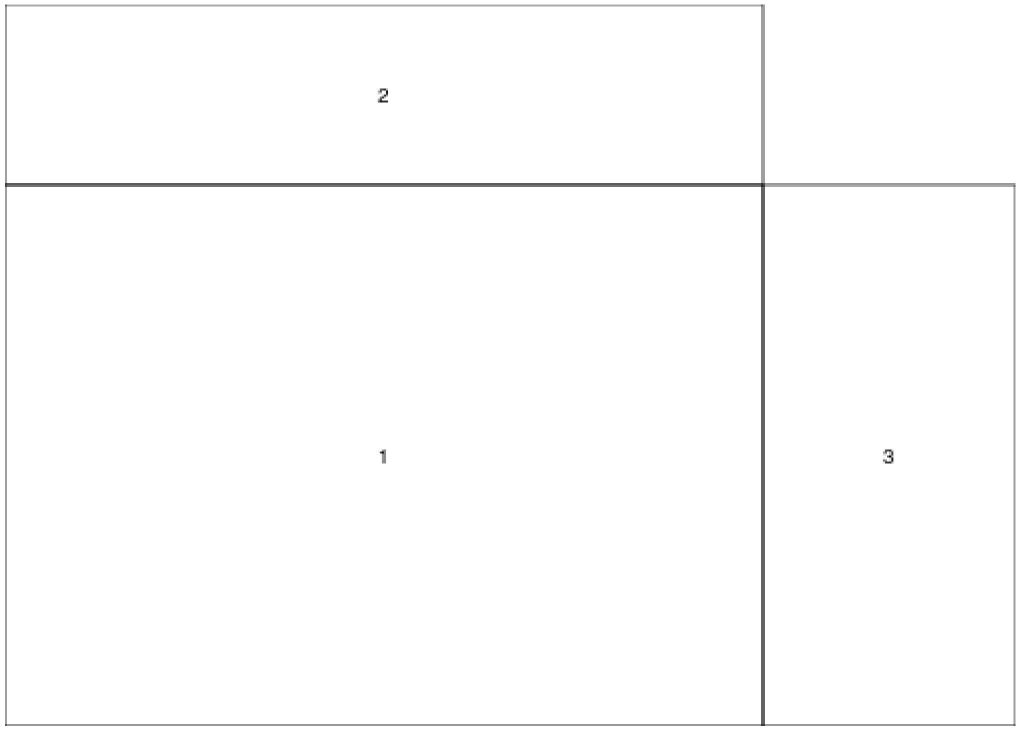
```
m <- matrix(c(2, 0, 1, 3), 2, 2, byrow = TRUE)
m
```

```
     [,1] [,2]
[1,]    2    0
[2,]    1    3
```

Using layout() (3)

- **Layout with different sizes:**

```
nf <- layout(m, widths = c(3,1), heights = c(1, 3))
layout.show(nf)
```

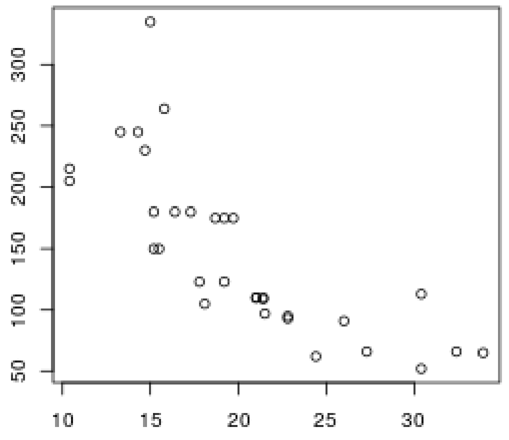


Using layout() (4)

```r
## min und max in both axis
xmin <- min(mtcars$mpg); xmax <- max(mtcars$mpg)
ymin <- min(mtcars$hp); ymax <- max(mtcars$hp)

## calculate histograms
xhist <- hist(mtcars$mpg, breaks=15, plot=FALSE)
yhist <- hist(mtcars$hp, breaks=15, plot=FALSE)

## maximum count
top <- max(c(xhist$counts, yhist$counts))
xrange <-  c(xmin,xmax)
yrange <- c(ymin, ymax)
```

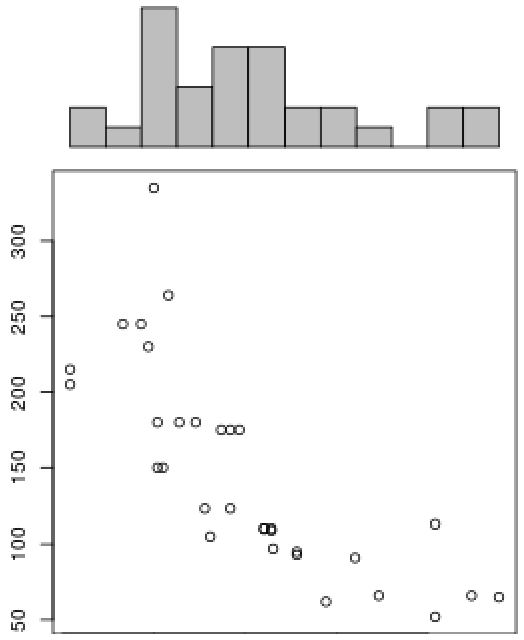
Using layout() (5)

```
layout(m, c(3,1), c(1, 3), TRUE)
## first plot:
plot(mtcars[,c("mpg","hp")], xlim=xrange, ylim=yrange, xlab="", ylab="")
```



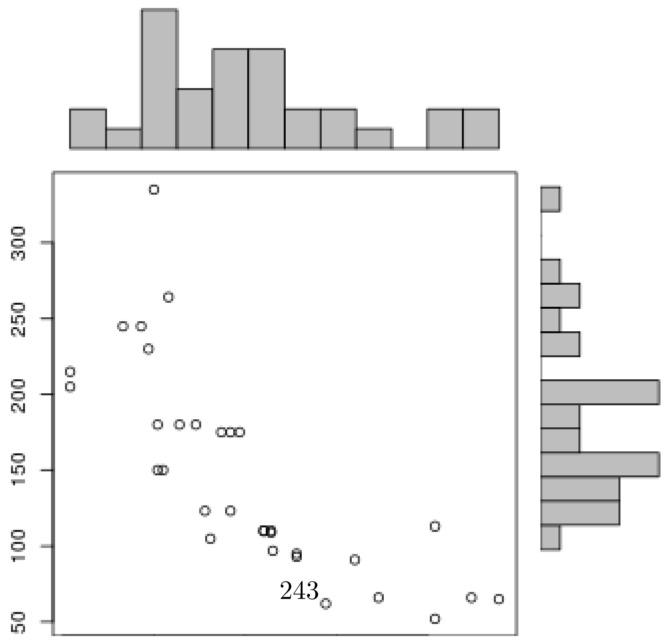
Using layout() (6)

```
layout(m, c(3,1), c(1, 3), TRUE)
par(mar=c(0,0,1,1))
plot(mtcars[,c("mpg","hp")], xlim=xrange, ylim=yrange, xlab="", ylab="")
## plus second plot:
par(mar=c(0,0,1,1))
barplot(xhist$counts, axes=FALSE, ylim=c(0, top),
    space=0)
```



Using layout() (7)

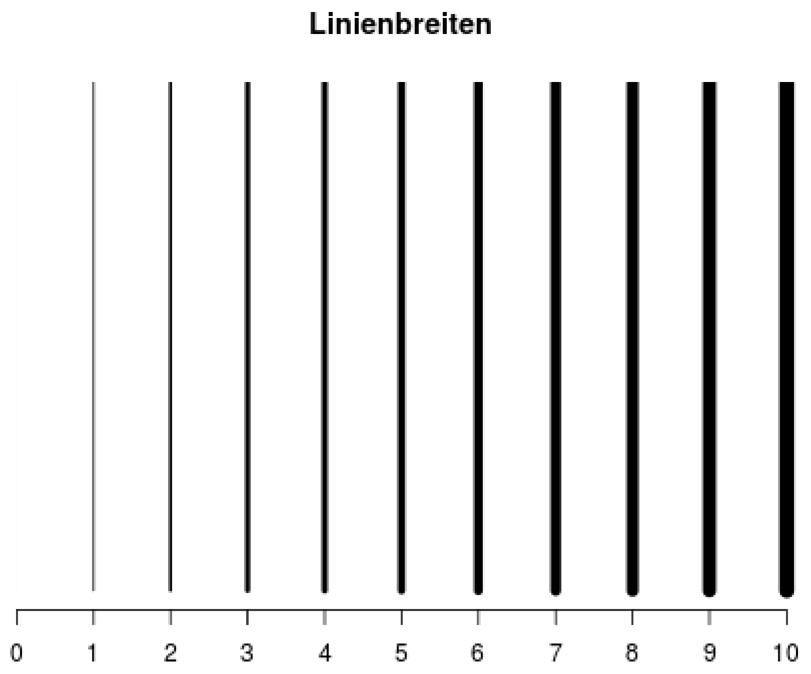
```
layout(m, c(3,1), c(1, 3), TRUE)
par(mar=c(0,0,1,1))
plot(mtcars[,c("mpg","hp")], xlim=xrange, ylim=yrange, xlab="", ylab="")
par(mar=c(0,0,1,1))
barplot(xhist$counts, axes=FALSE, ylim=c(0, top),
    space=0)
## plus third plot:
par(mar=c(3,0,1,1))
barplot(yhist$counts, axes=FALSE, xlim=c(0, top),
    space=0, horiz=TRUE)
```



Eurostat

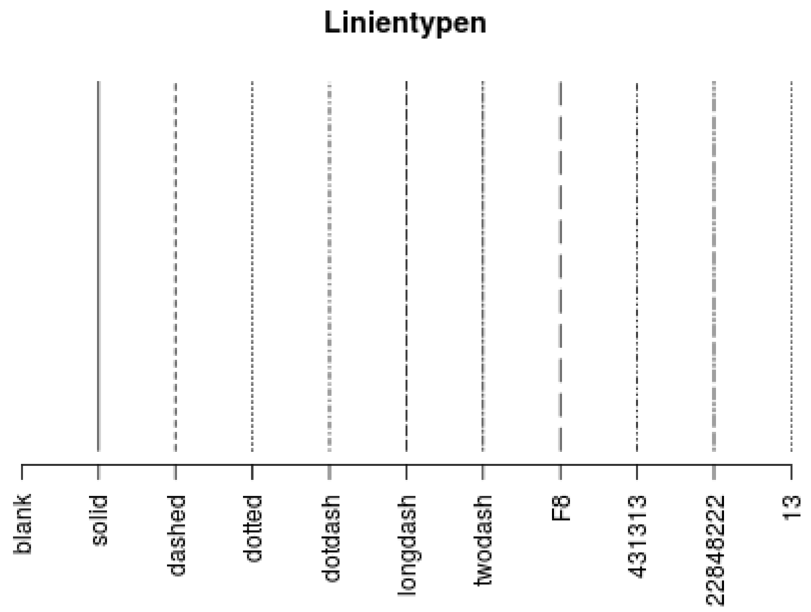
Modifying line widths

```
plot.new()
SEQ <- 0:10
segments(x0=SEQ/10, y0=0, x1=SEQ/10, y1=1.5, lwd=SEQ)
axis(side=1, at=SEQ/10, lab=SEQ)
title("Linienbreiten")
```



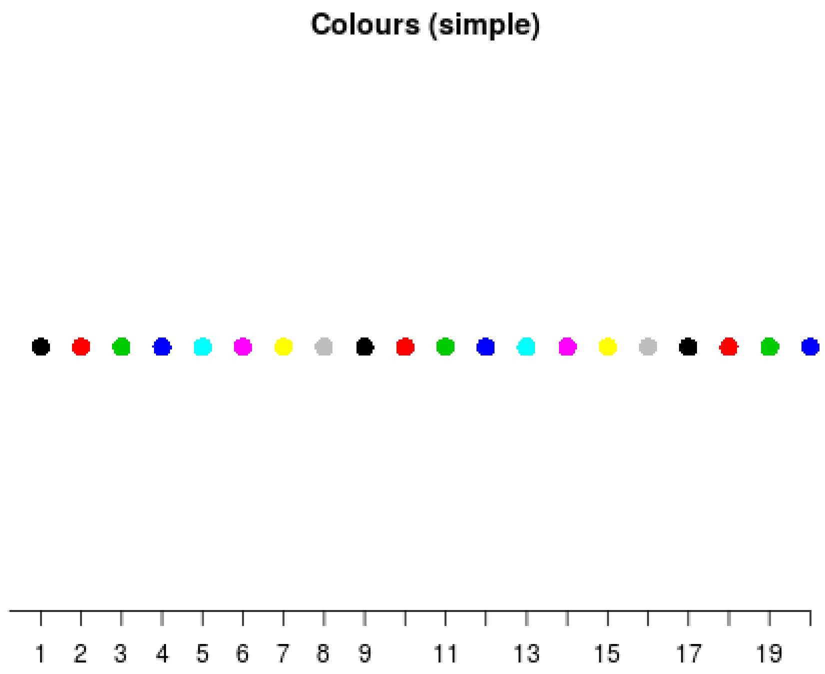
Modifying line types

```r
plot.new()
ltyvec <- c("blank","solid","dashed","dotted","dotdash",
"longdash","twodash","F8","431313","22848222","13")
segments(SEQ/10,0,SEQ/10,1.5, lty=ltyvec)
axis(side=1, at=SEQ/10, lab=ltyvec, las=3)
title("Linientypen")
```



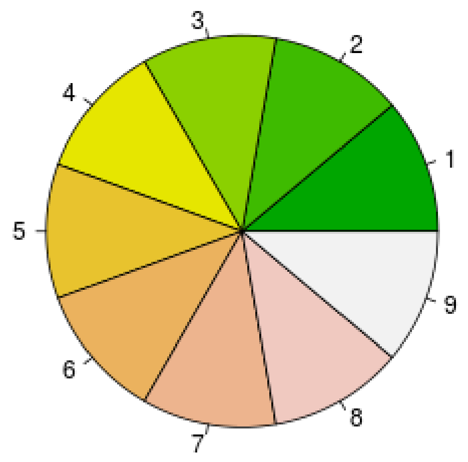
Colors (recapitulation)

```r
plot(1:20,rep(0,20), pch=16, cex=1.7, col=1:20, xlab="", ylab="", axes=FALSE, main="Colours (simple)")
axis(side=1, at=0:20, lab=0:20, cex=0.8)
```



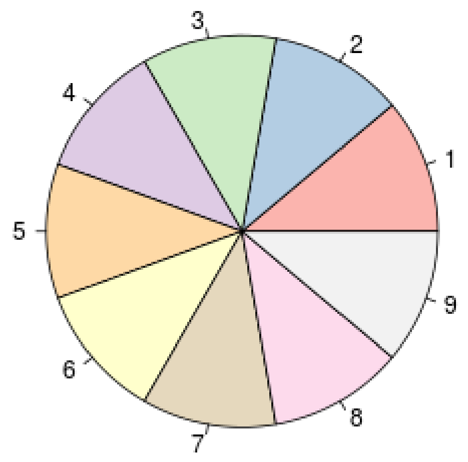
Colors (recapitulation)

```
pie(rep (1,9), col = terrain.colors(9))
```



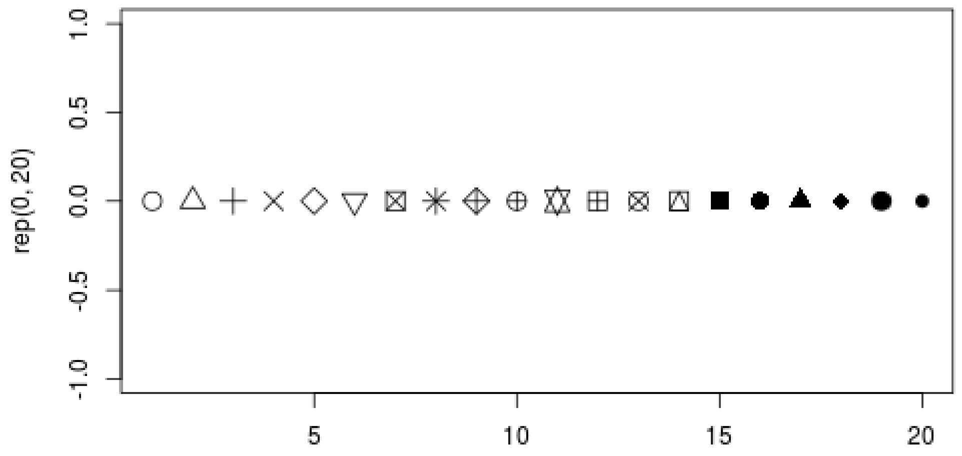
Colors (recapitulation RColorBrewer)

```
pie (rep (1,9), col = brewer.pal (9, "Pastel1"))
```



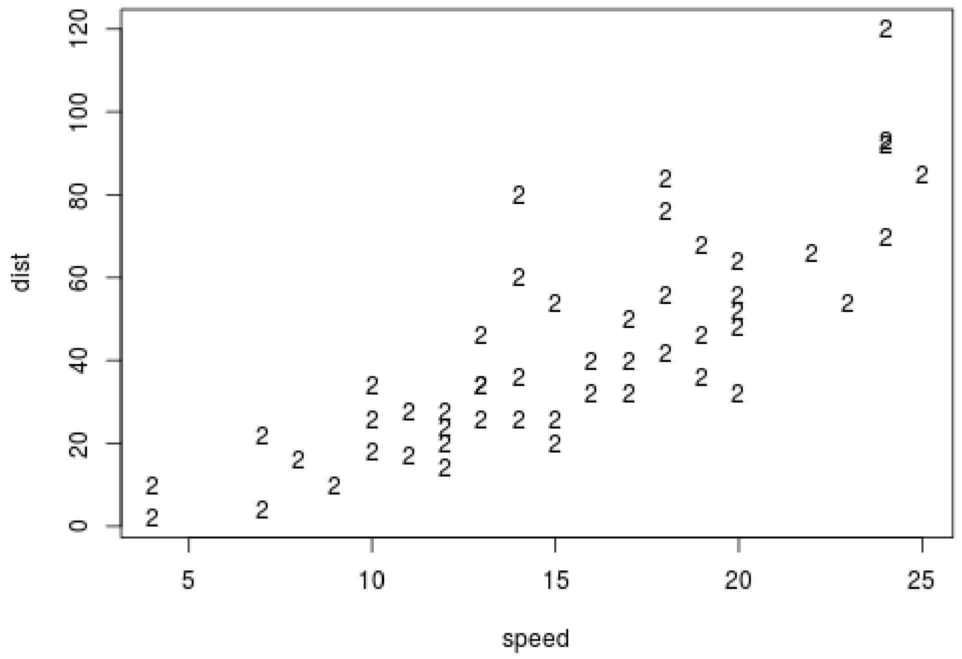
Symbols (1)

```
plot (1:20, rep(0,20), pch = 1:20, cex = 1.7, xlab = "", ylaB = "")
```



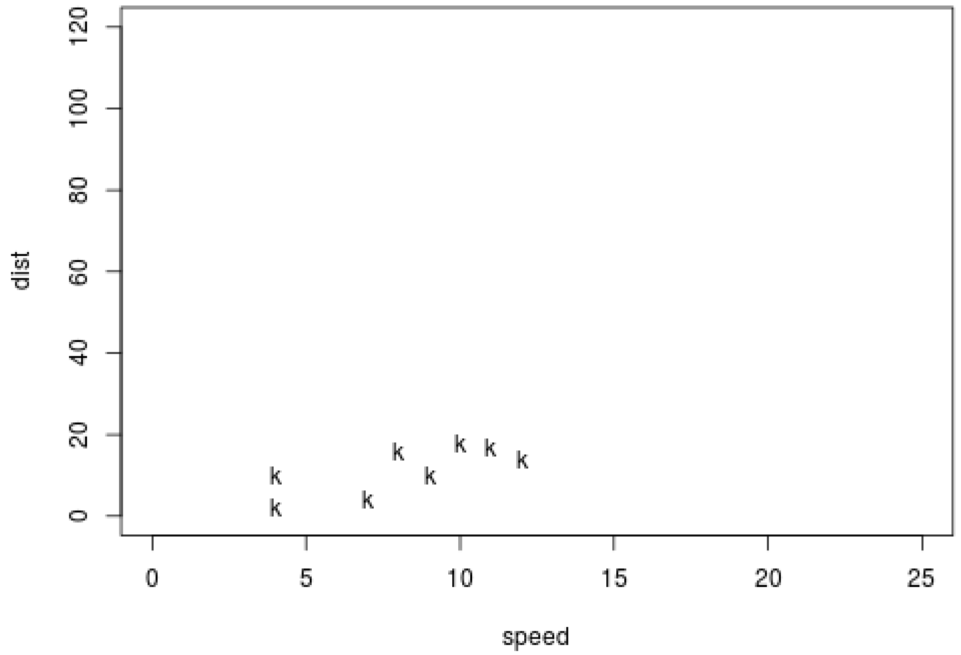
Symbols (2)

```
plot(cars, pch = "2")
```



Symbols (3)

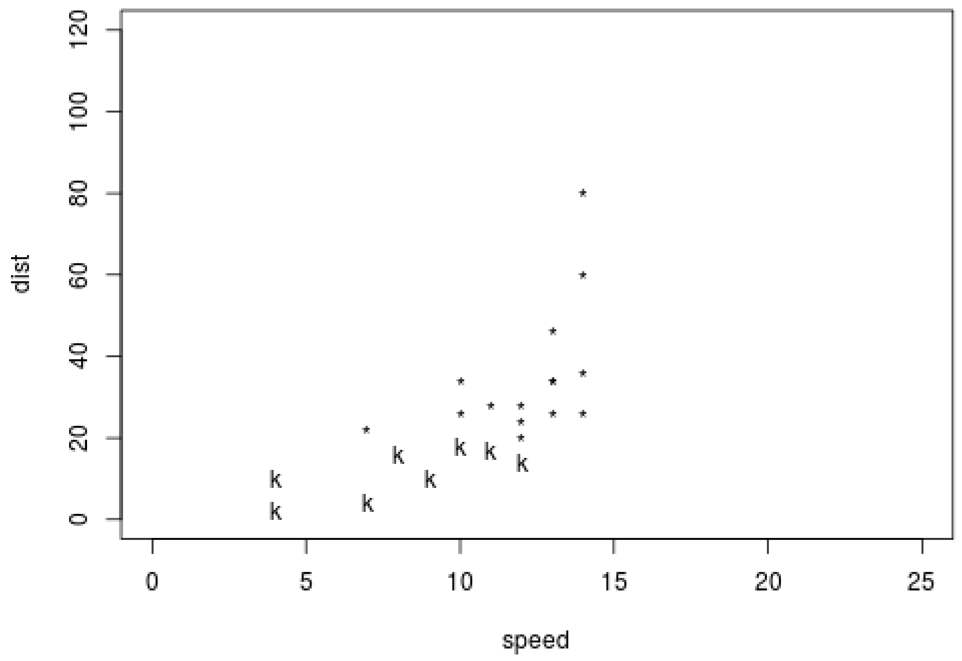
```
plot(cars[cars$speed < 15 & cars$dist < 20, ], pch="k", xlim=c(0,max(cars$speed)), ylim=c(0,max(cars$dist)))
```



Eurostat

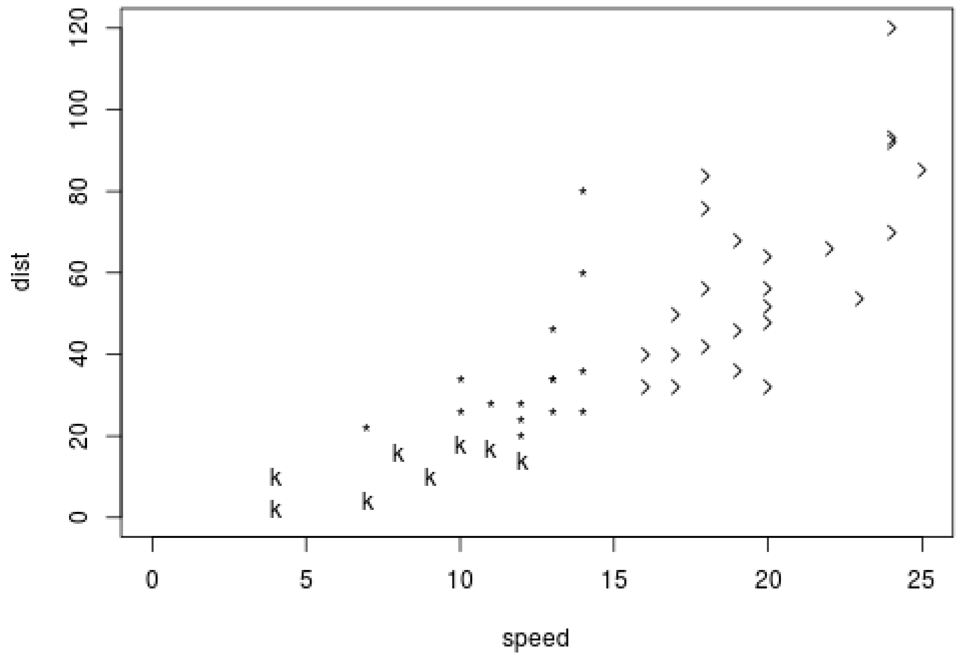
Symbols (4)

```
plot(cars[cars$speed < 15 & cars$dist < 20, ], pch="k", xlim=c(0,max(cars$speed)), ylim=c(0,max(cars$dist)))
points(cars[cars$speed < 15 & cars$dist >= 20, ], pch="*")
```



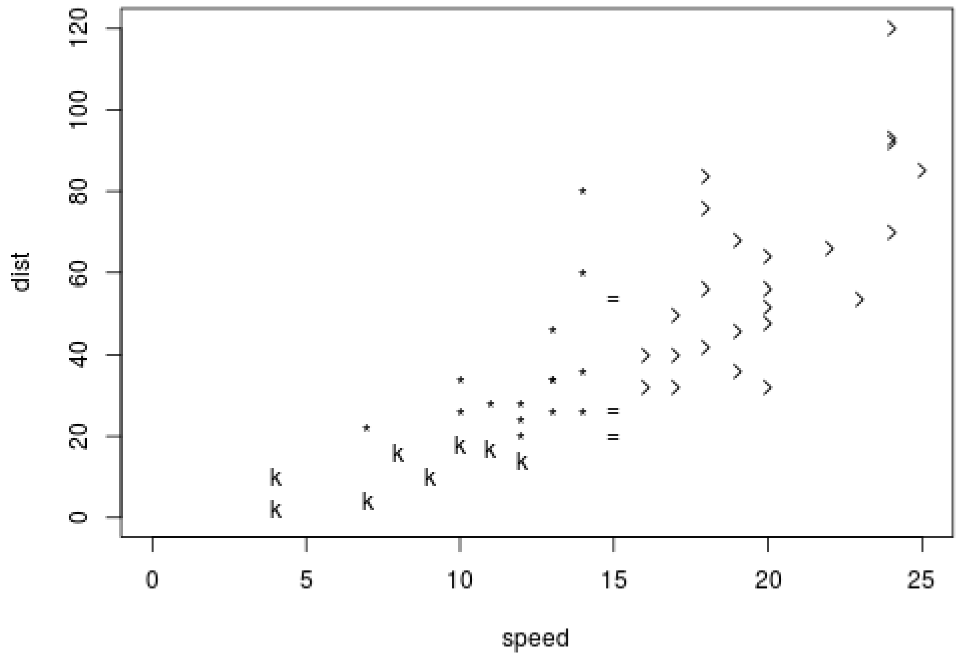
Symbols (5)

```
plot(cars[cars$speed < 15 & cars$dist < 20, ], pch="k", xlim=c(0,max(cars$speed)), ylim=c(0,max(cars$dist)))
points(cars[cars$speed < 15 & cars$dist >= 20, ], pch="*")
points(cars[cars$speed >15, ], pch=">")
```



Symbols (6)

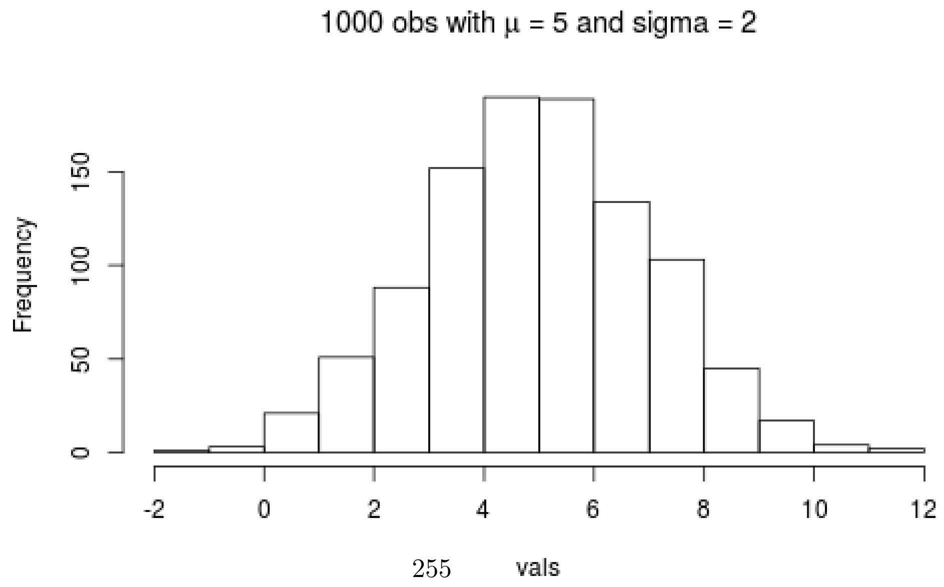
```
plot(cars[cars$speed < 15 & cars$dist < 20, ], pch="k", xlim=c(0,max(cars$speed)), ylim=c(0,max(cars$dist)))
points(cars[cars$speed < 15 & cars$dist >= 20, ], pch="*")
points(cars[cars$speed >15, ], pch=">")
points(cars[cars$speed == 15, ], pch="=")
```



Expressions (1)

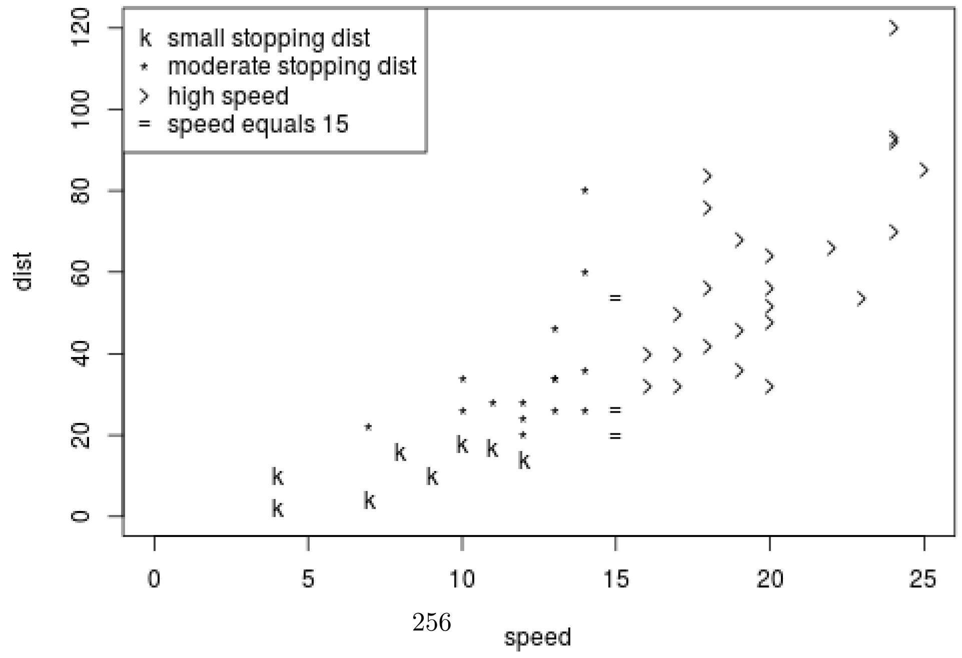
- It is possible to add (\(\LaTeX \)) code in titles
- this can be done using **expression()** or **bquote()**
- for detailed information have a look at **?plotmath**

```
vals <- rnorm(1000, mean=5, sd=2)
hist(vals, main=bquote(.(length(vals)) ~ "obs with" ~ mu ~ "= 5 and sigma = 2"))
```



Legends

```
plot(cars[cars$speed < 15 & cars$dist < 20, ], pch="k", xlim=c(0,max(cars$speed)), ylim=c(0,max(cars$dist)))
points(cars[cars$speed < 15 & cars$dist >= 20, ], pch="*")
points(cars[cars$speed >15, ], pch=">")
points(cars[cars$speed == 15, ], pch="=")
legend("topleft", pch=c("k","*",">","="), legend=c("small stopping dist","moderate stopping dist","high
speed","speed equals 15"))
```



Tasks / Exercises

Time for practical training! :)

Please continue to work on Exercises 1x) to 5x).

Summary (1)

- Very easy to create graphics
- High- and low-level graphics
- However plot margins has to be set for nice looking graphics (note: lattice and ggplot2 calculate margins automatically)
- par() !

The grammar of graphics in R: ggplot2

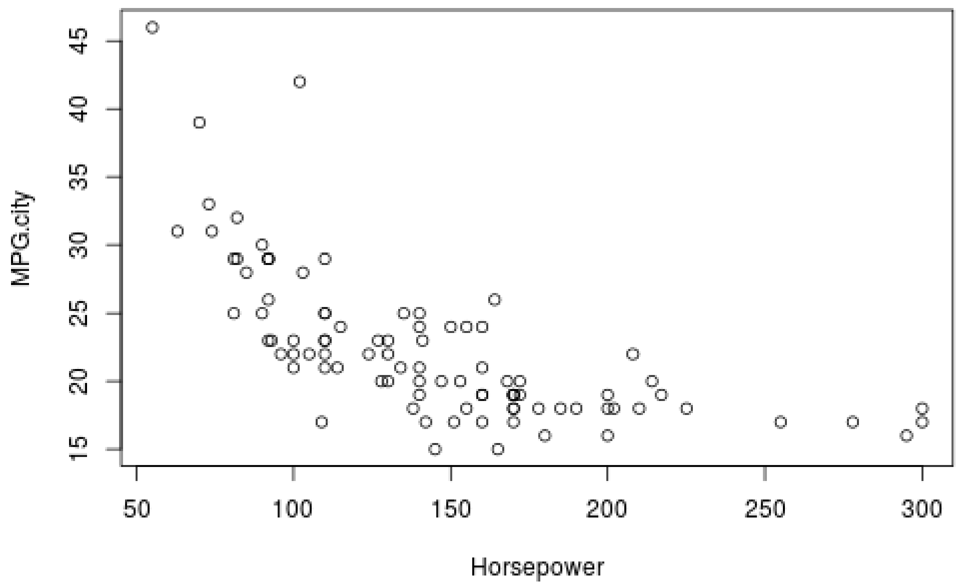
Alexander Kowarik, Bernhard Meindl

Why ggplot2

- consistent and systematic approach to generate graphics
- based on the book *Grammar of Graphics* by Wilkinson
- very flexible
- customizable. It allows to define themes (e.g. to match the corporate designs of your company)
- But: slow(er) and not as easy to learn

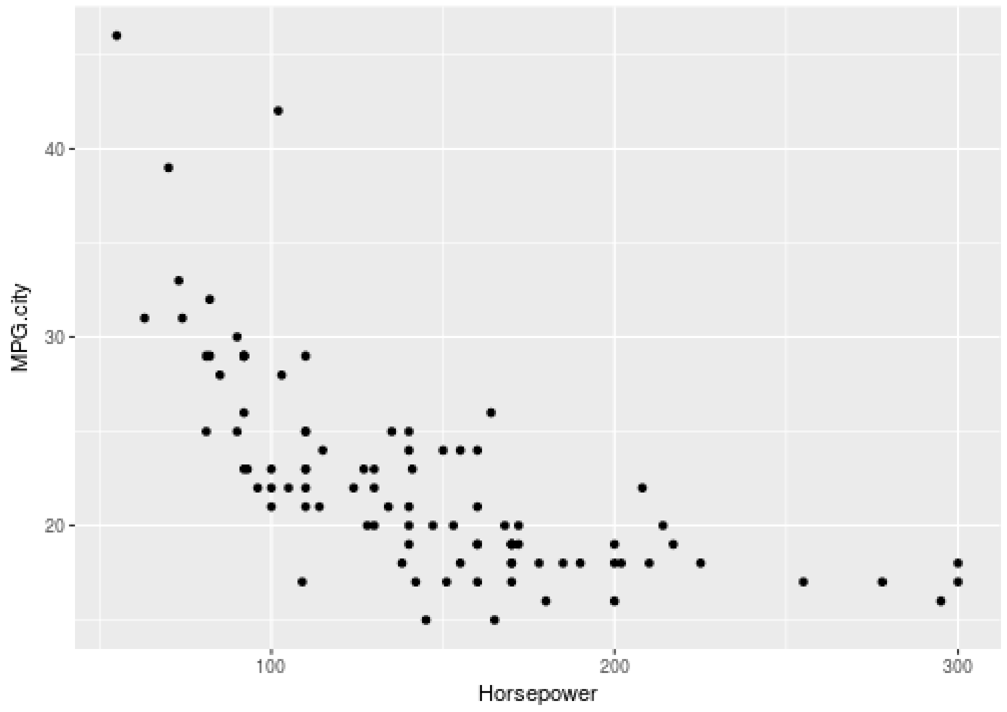
Example of a plot (1) - graphics

```
data("Cars93", package="MASS")
plot(MPG.city ~ Horsepower, data=Cars93)
```



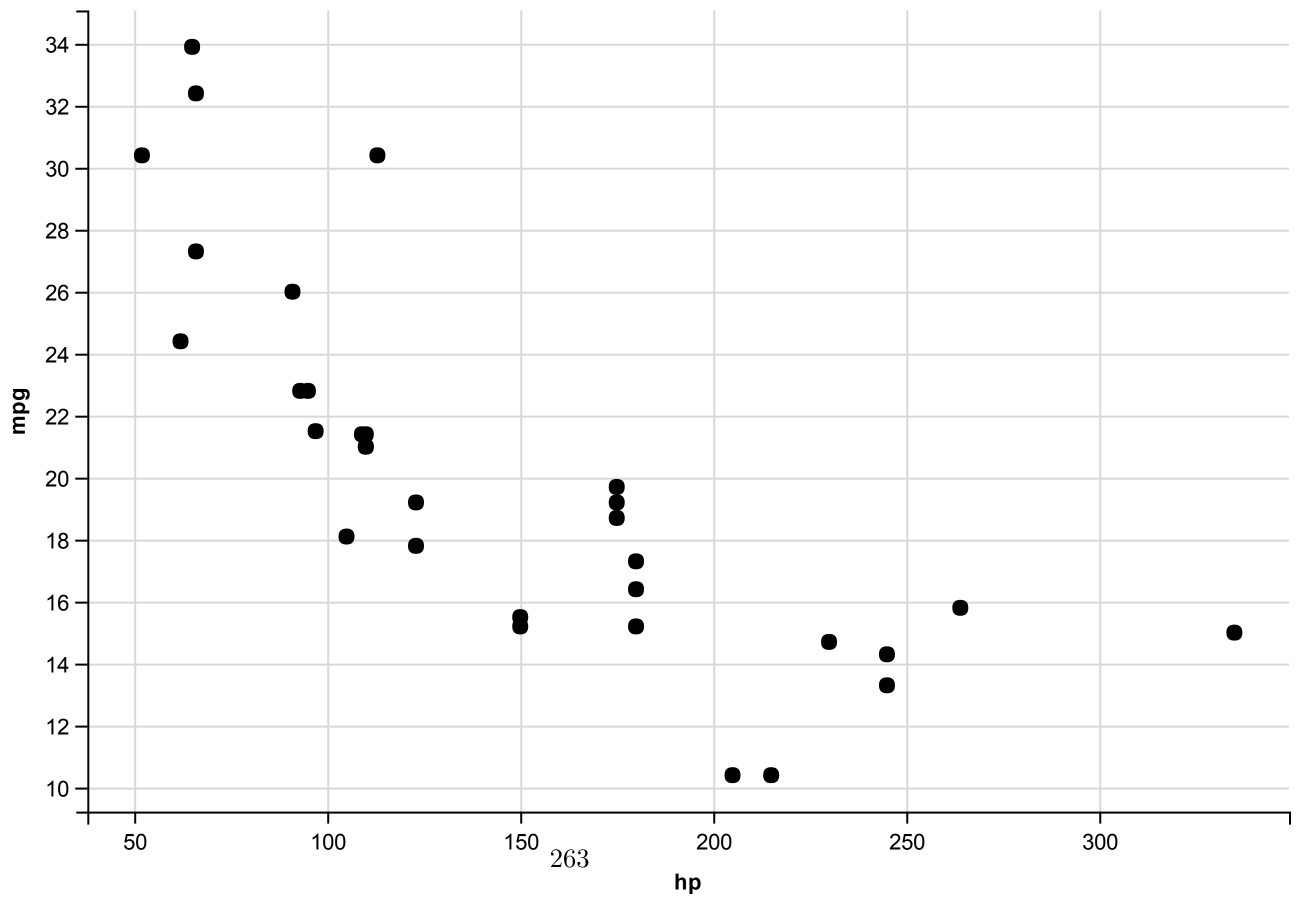
Example of a plot (2) - ggplot2

```
require(ggplot2)
ggplot(Cars93, aes(x=Horsepower, y=MPG.city)) + geom_point()
```



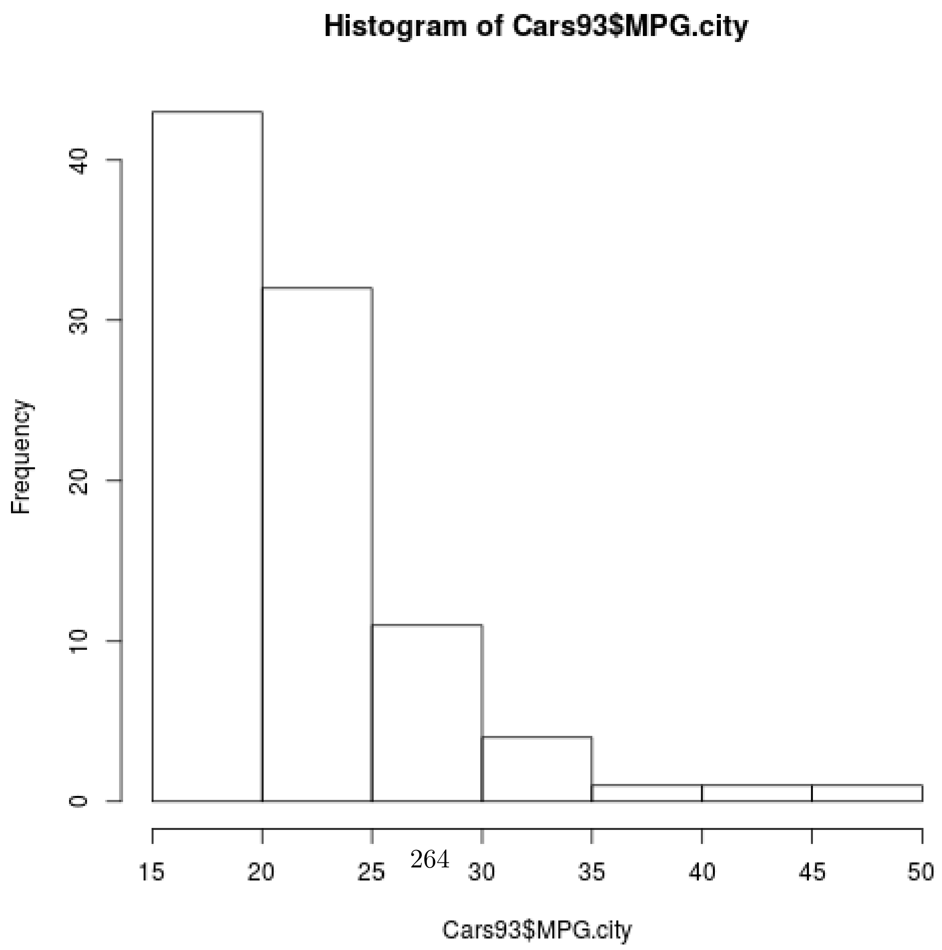
Example of a plot (3) - ggvis

```r
require(ggvis)
Cars93 %>% ggvis(x = ~Horsepower, y = ~MPG.city) %>% layer_points()
```



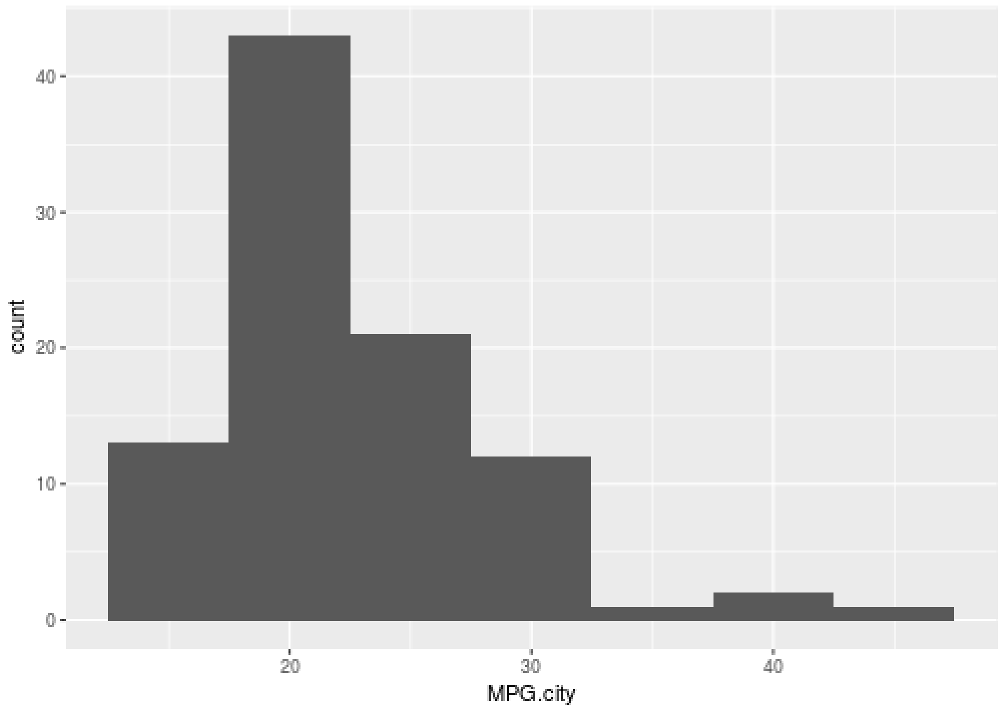
Output from graphics and ggplot2 (1)

```
hist(Cars93$MPG.city)
```



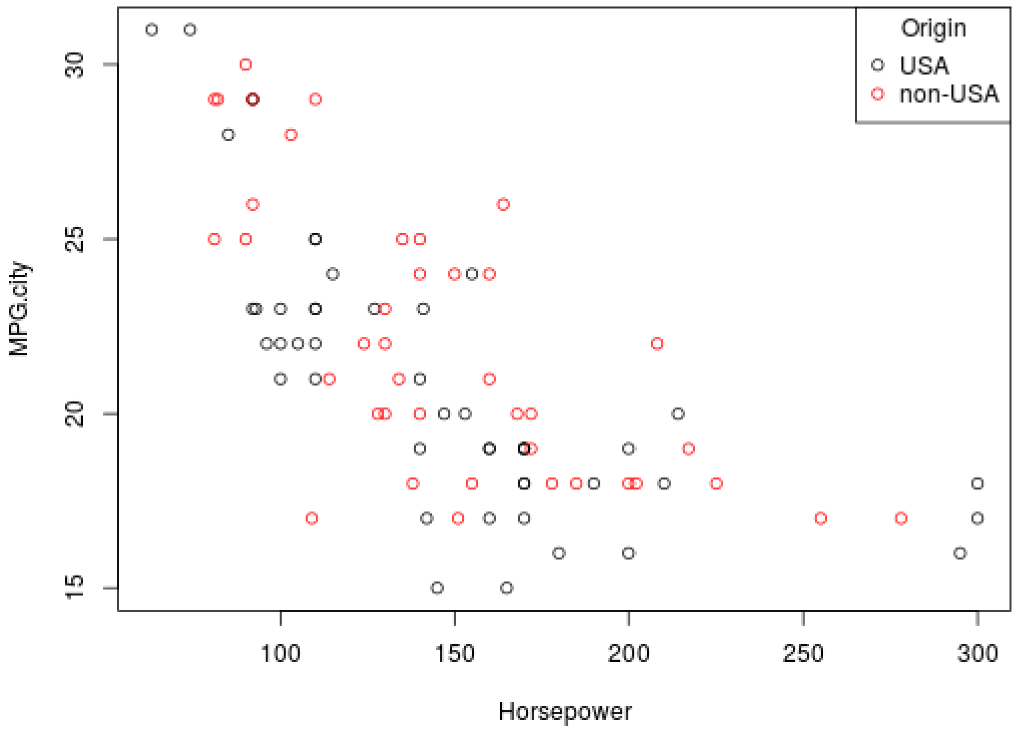
Output from graphics and ggplot2 (2)

```r
ggplot(Cars93, aes(x=MPG.city))+
  geom_histogram(binwidth=5)
```



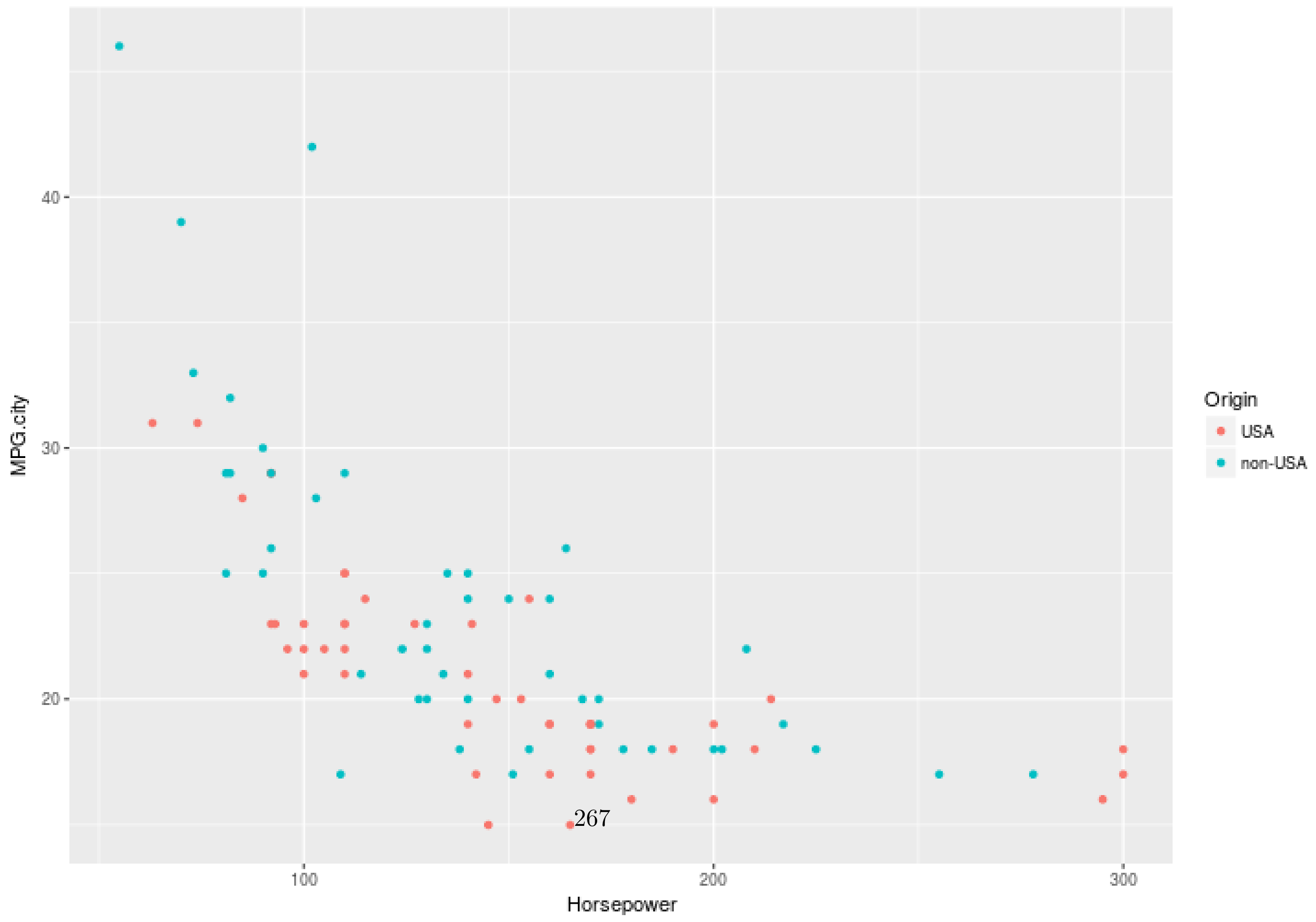
Output from graphics and ggplot2 (3)

```
par(mar=c(4,4,.1,.1))
plot(MPG.city ~ Horsepower, data=subset(Cars93, Origin == "USA"))
points(MPG.city ~ Horsepower, col="red", data=subset(Cars93, Origin == "non-USA"))
legend("topright", c("USA", "non-USA"), title="Origin", pch=c(1,1),  col=c("black", "red"))
```



Output from graphics and ggplot2 (4)

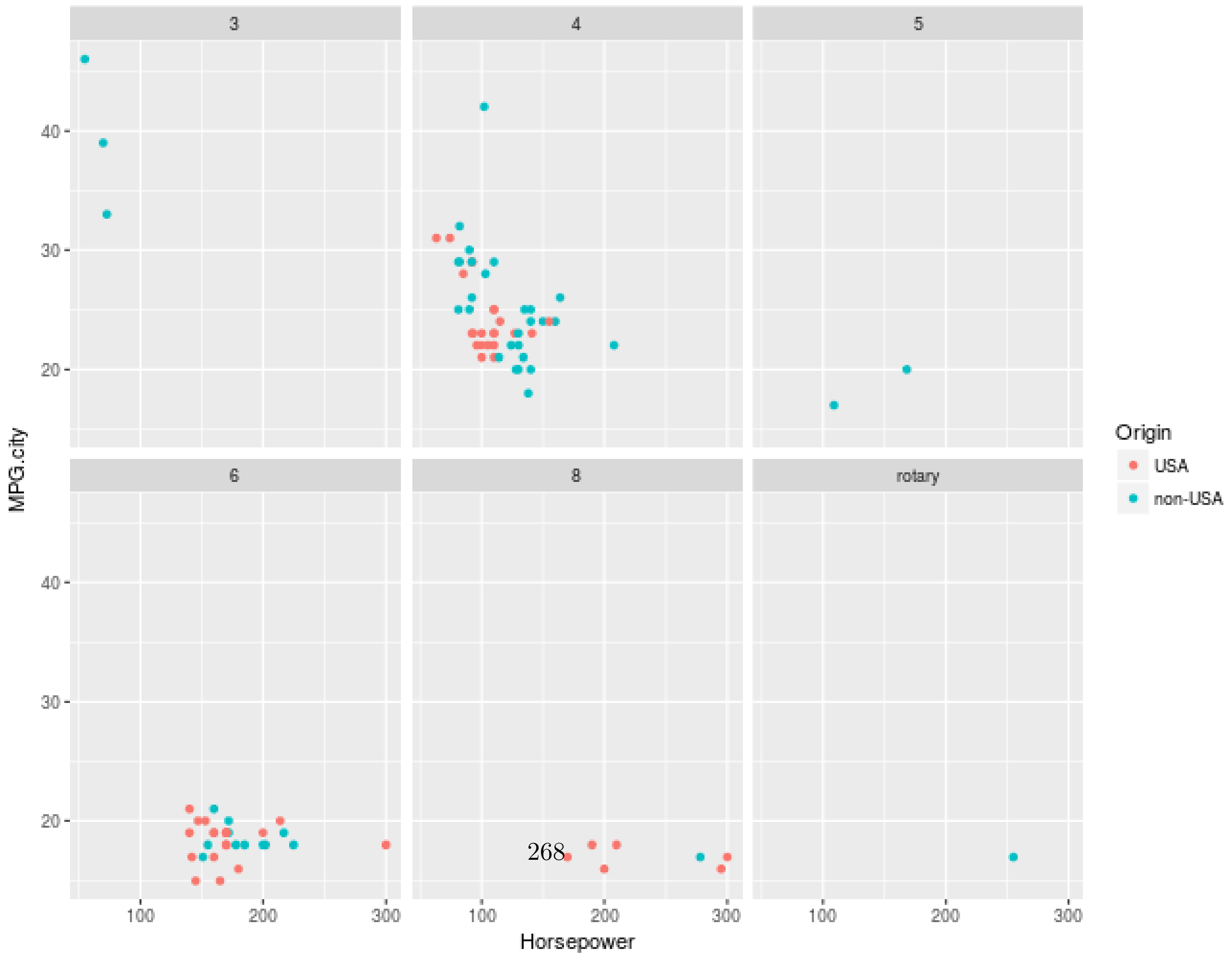
```
ggplot(Cars93, aes(x=Horsepower, y=MPG.city, color=Origin)) + geom_point()
```



Eurostat

Output from graphics and ggplot2 (5)

```
ggplot(Cars93, aes(x=Horsepower, y=MPG.city, color=Origin)) + geom_point() + facet_wrap(~Cylinders)
```



Eurostat

Grammar of graphics with ggplot2

- parts of a plot defined independently
- the anatomy of a plot:
  - data
  - **aesthetic mapping**: describe how **variables** in the data are mapped to visual properties (aesthetics) of geometric objects
  - **assignment**: values are assigned to visual properties
  - geometric objects (geom's, aesthetic will be mapped to geometric objects)
  - statistical transformation
  - scales
  - coordinate system
  - position adjustments
  - faceting

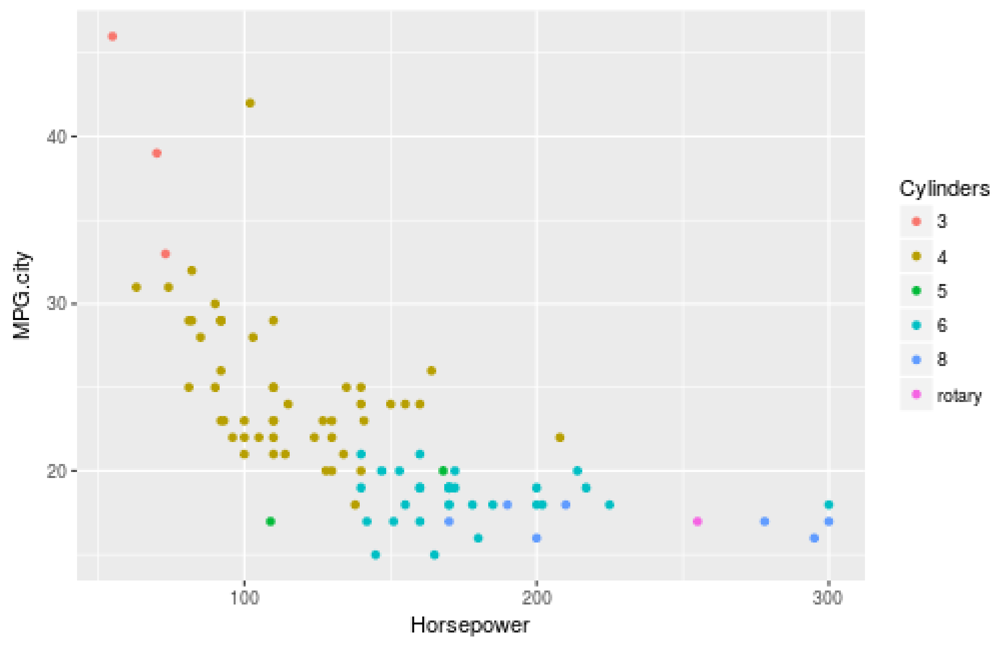
Aesthetics

- aesthetic means “something you can see”
  - color
  - fill (color)
  - shape (of points)
  - linetype
  - size
  - ...

Aesthetic mapping

- aesthetic mapping to geometric objects with function **aes()**

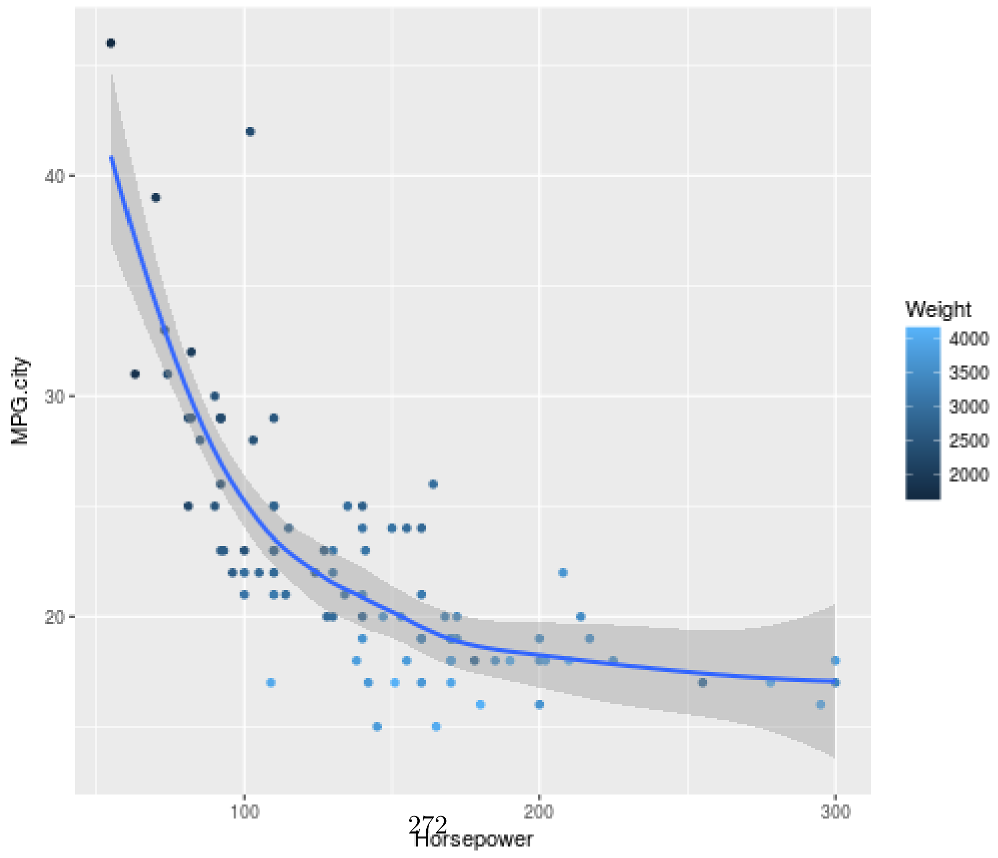
```
ggplot(Cars93, aes(x = Horsepower, y = MPG.city)) + geom_point(aes(color = Cylinders))
```



- each type of geom accepts only a subset of aesthetics (e.g. setting **shape** in **aes()** makes no sense in **geom_bar()**)
- add geom using **+**

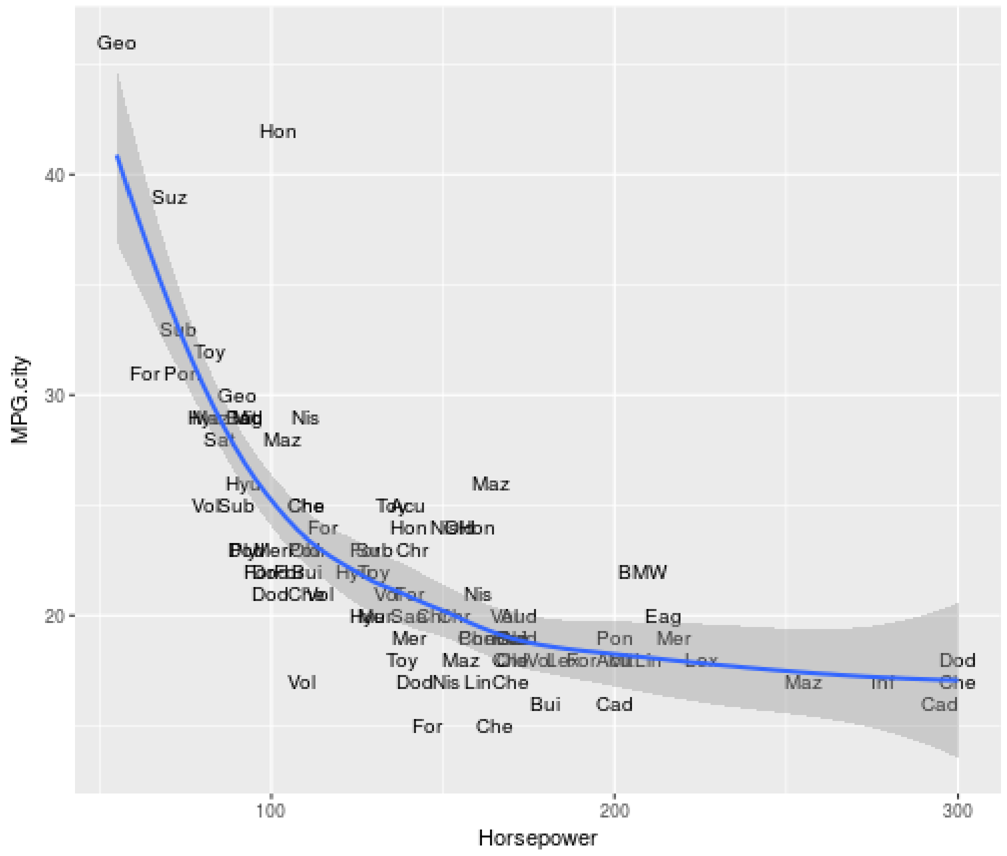
Multiple geoms (1)

```
g1 <- ggplot(Cars93, aes(x=Horsepower, y=MPG.city))
g2 <- g1 + geom_point(aes(color=Weight)) + geom_smooth()
g2
```



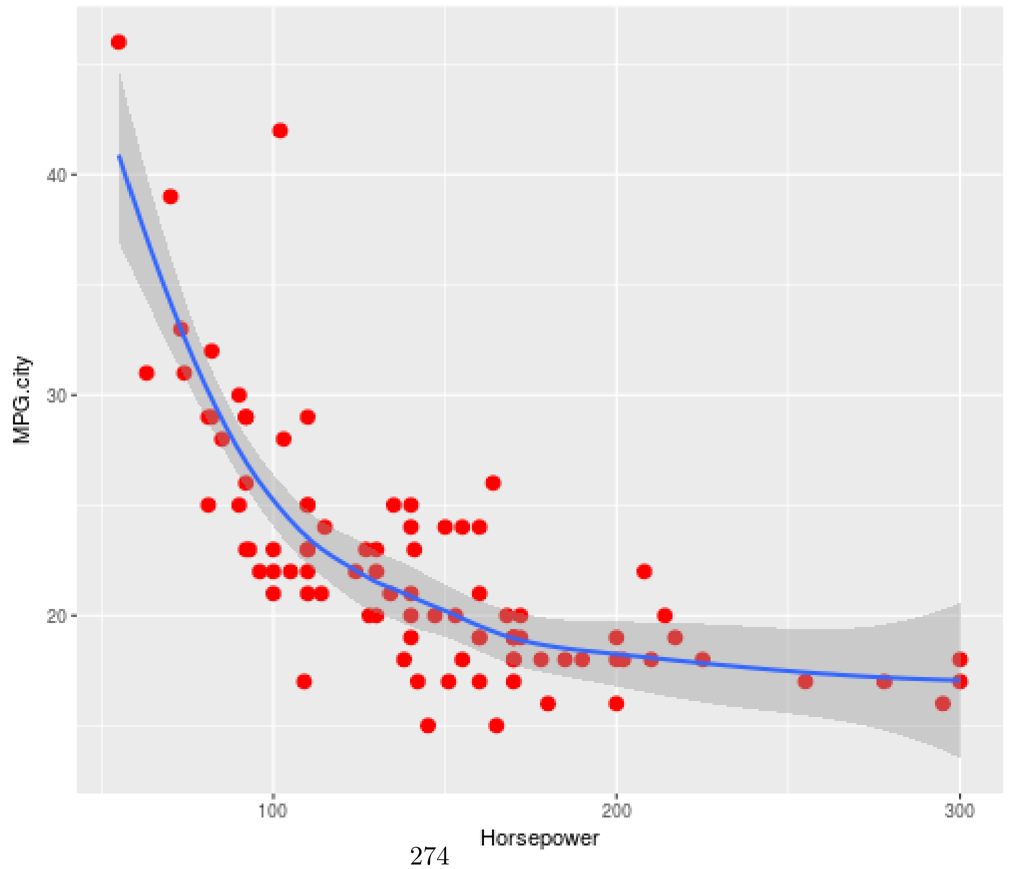
Multiple geoms (2)

```r
g1 + geom_text(aes(label=substr(Manufacturer,1,3)), size=3.5) + geom_smooth()
```



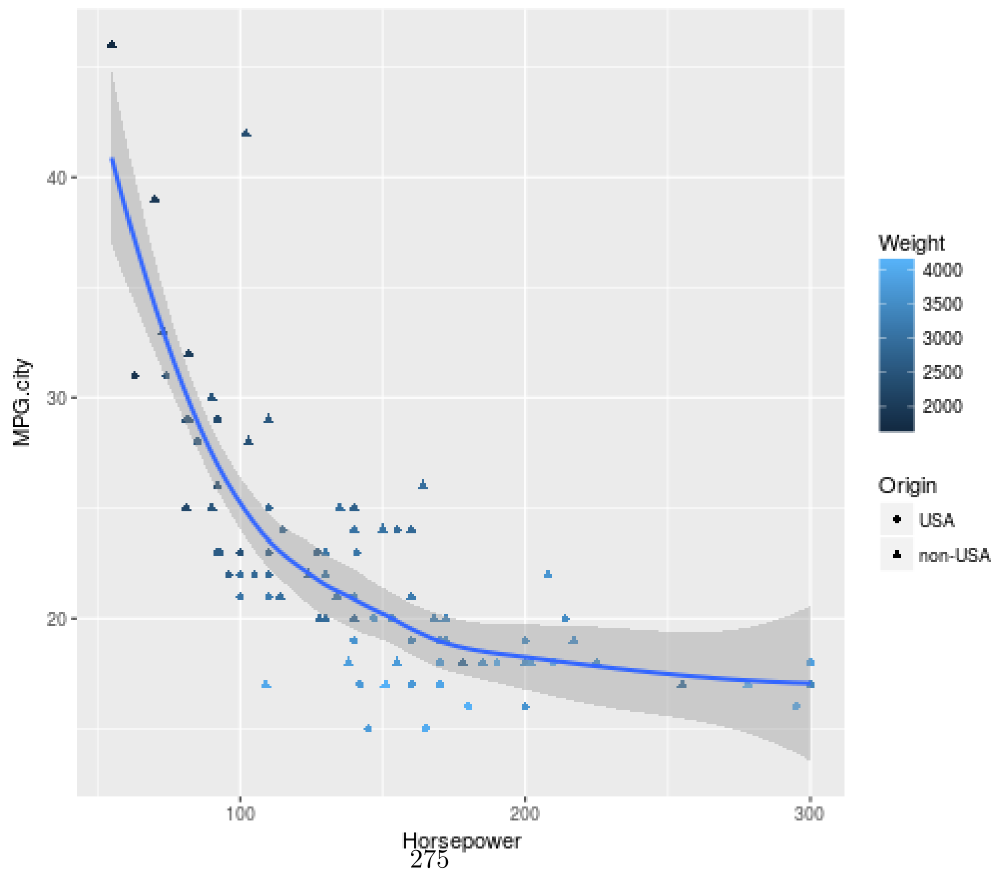
Multiple geoms (3)

```
## value outside aes() -- assignment
g1 + geom_point(color="red", size=3) + geom_smooth()
```



Multiple geoms (4)

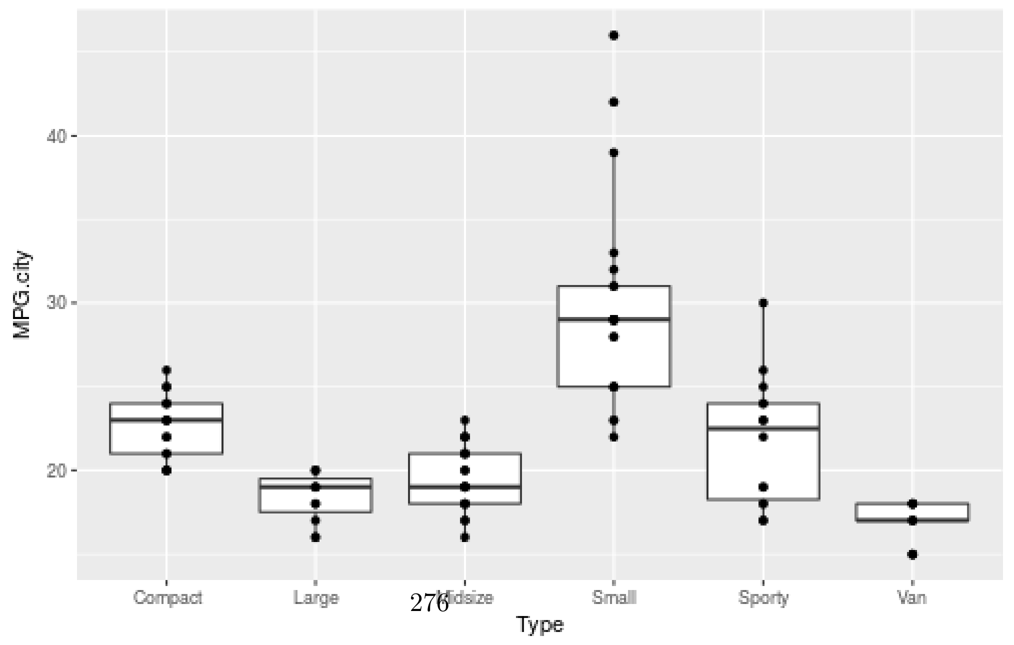
```
## variable inside aes() -- aesthetic mapping d
g1 + geom_point(aes(color=Weight, shape=Origin)) + geom_smooth()
```



Default parameters (1)

- each block of a ggplot2 can be defined with parameters
- to make life easy: standard default parameter values exist
- default values for geoms:

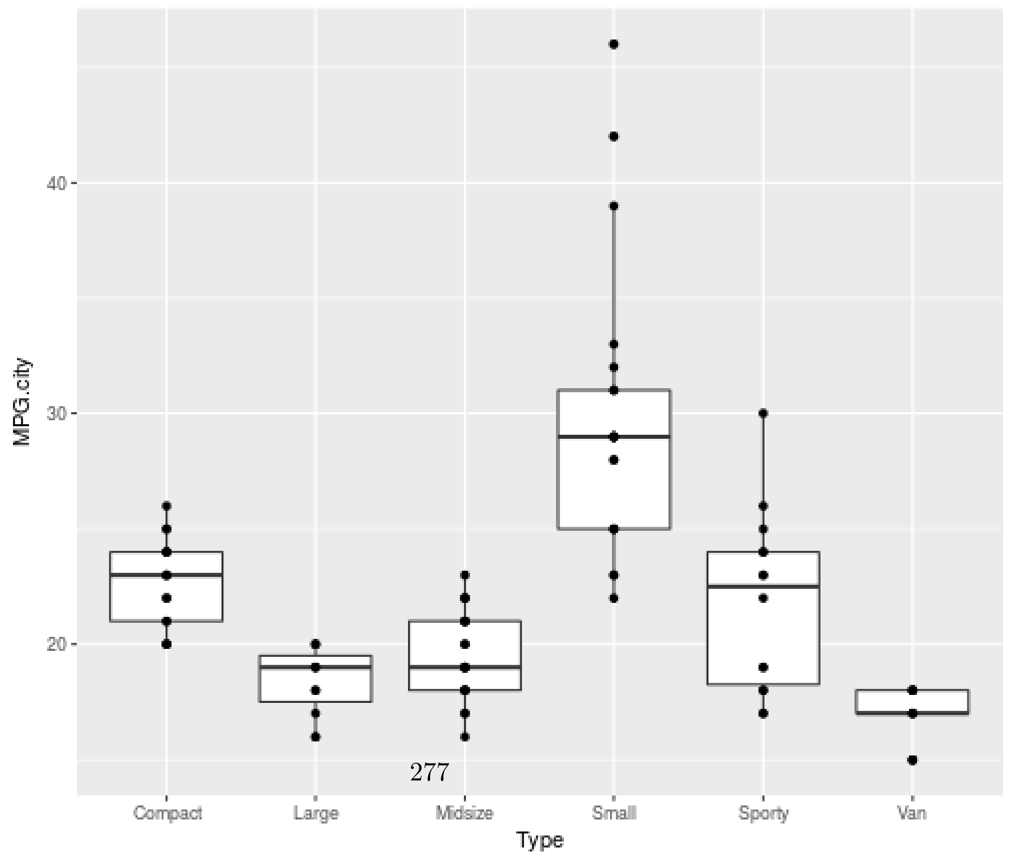
```
g <- ggplot(Cars93, aes(x = Type, y = MPG.city))
g + geom_boxplot() + geom_point()
```



Default parameters (2)

- default values for statistics:

```
g <- ggplot(Cars93, aes(x = Type, y = MPG.city))
g + stat_boxplot() + geom_point()
```

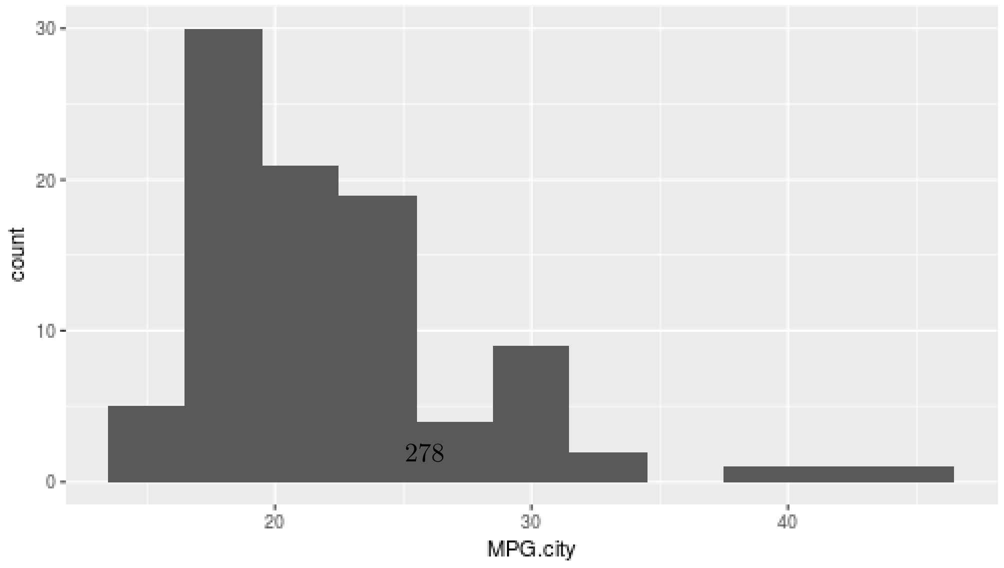


Stats

all (geoms) associated with statistical transformations - stats

- the easiest one: the identity
- for some geom's, the data are modified, e.g. geom_boxplot
- specific paramters for specific plots, e.g. *binwidth*

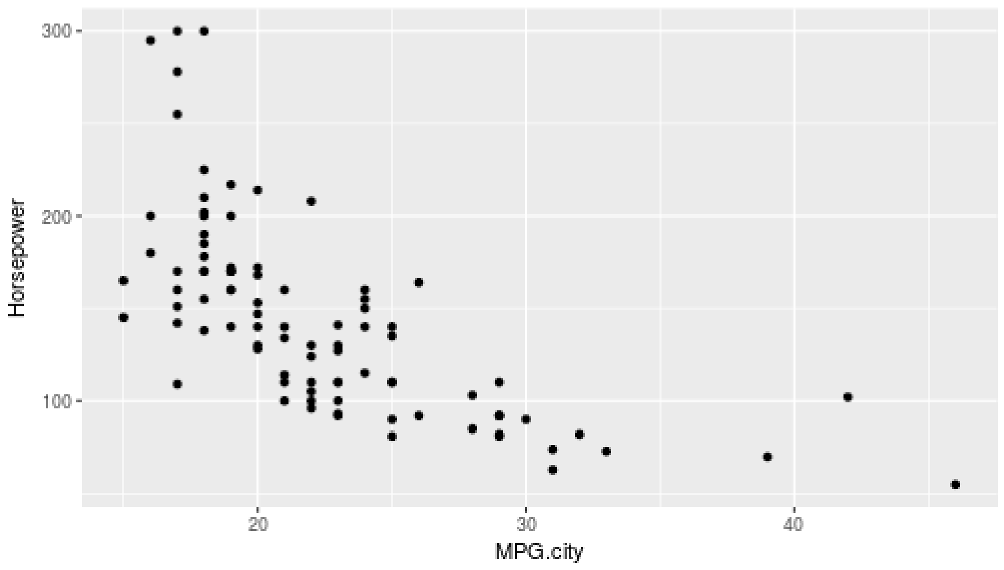
```
ggplot(Cars93, aes(x = MPG.city)) + geom_histogram(binwidth=3)
```



Example: Flexibilty (1)

- Scatterplot

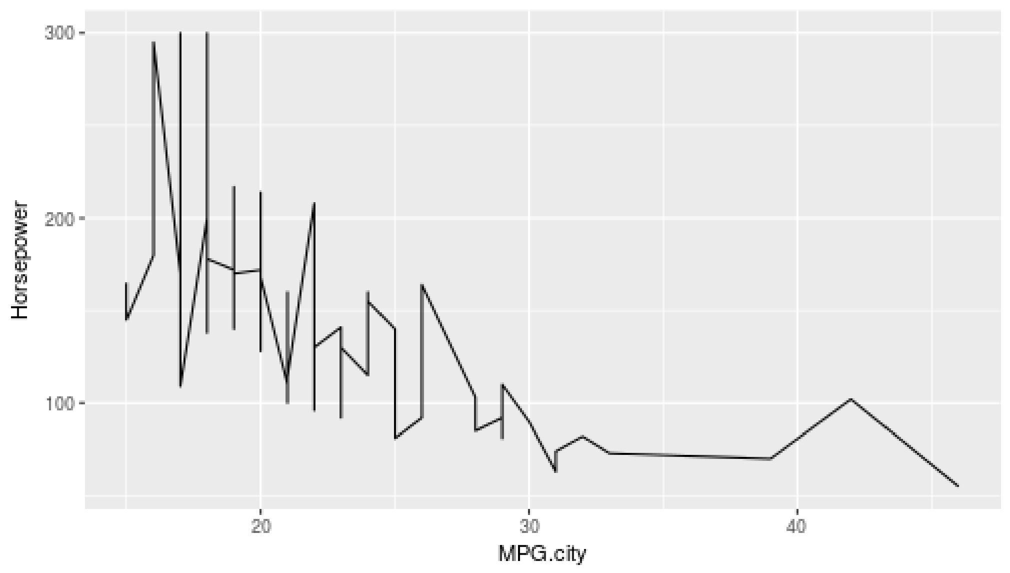
```
g <- ggplot(Cars93, aes(x = MPG.city, y = Horsepower))
g + geom_point()
```



Example: Flexibilty (2)

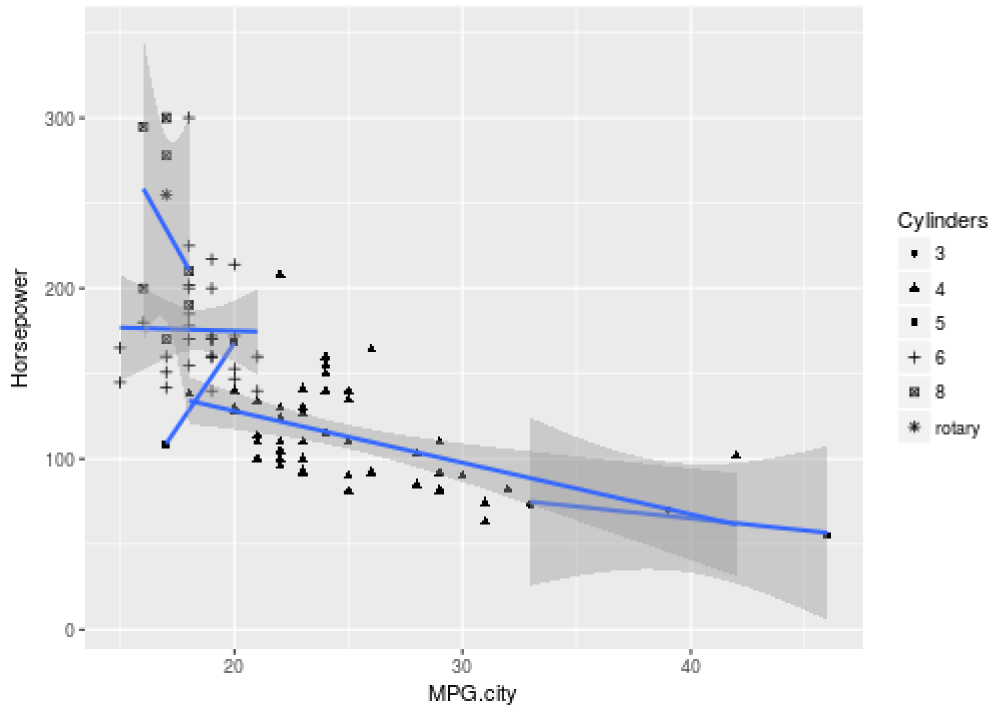
- Lines

```
g + geom_line()
```



Example: Grouping

```
ggplot(Cars93, aes(MPG.city, Horsepower, shape = Cylinders))+
  geom_point() + stat_smooth(method = "lm")
```



Eurostat

Scales: aesthetics for variables

scales can define:

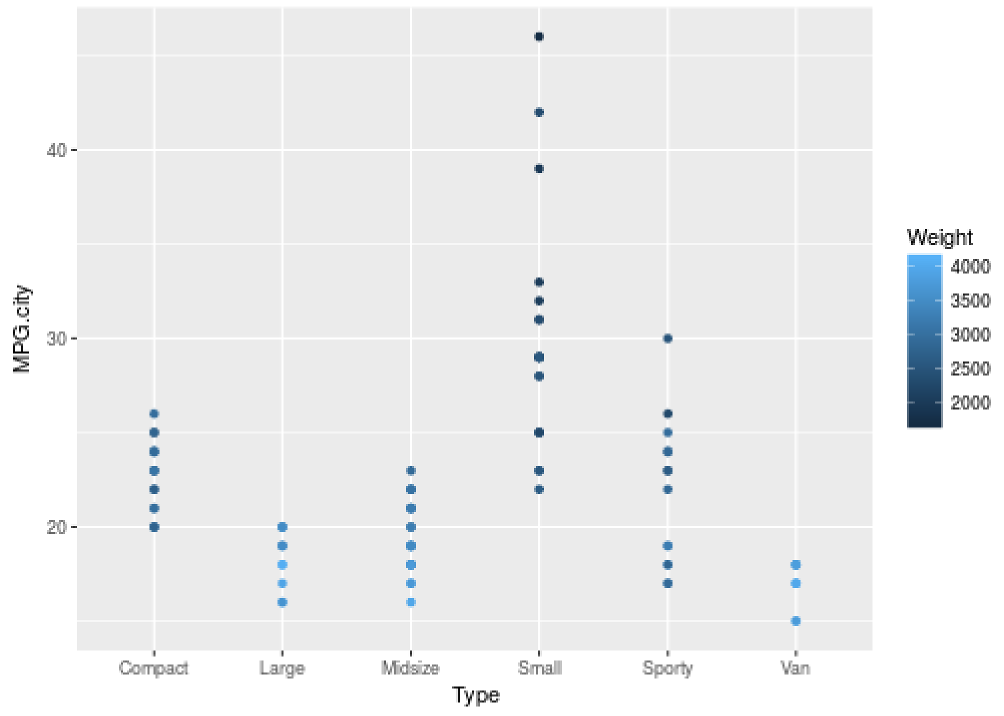
- color and fill (color)
- size
- shape
- linetype Scales are defined and modified by using the fucntion **scale**

Eurostat

STATISTIK AUSTRIA
Die Informationsmanager

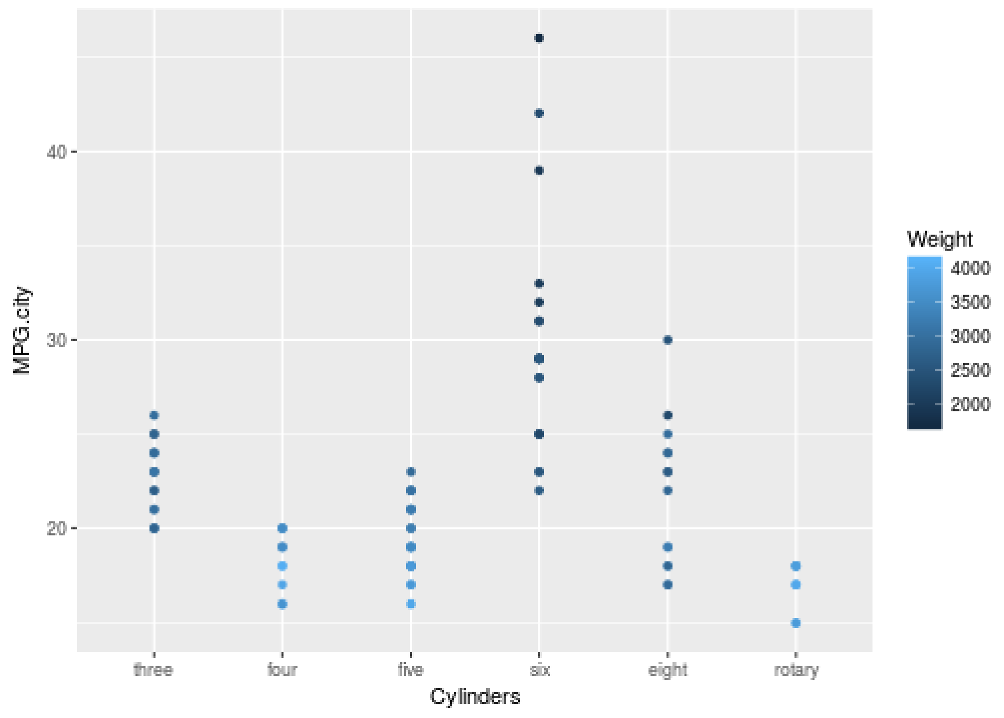
Example: Scales

```
g <- ggplot(Cars93, aes(x = Type, y = MPG.city))
g <- g + geom_point(aes(color = Weight))
g
```



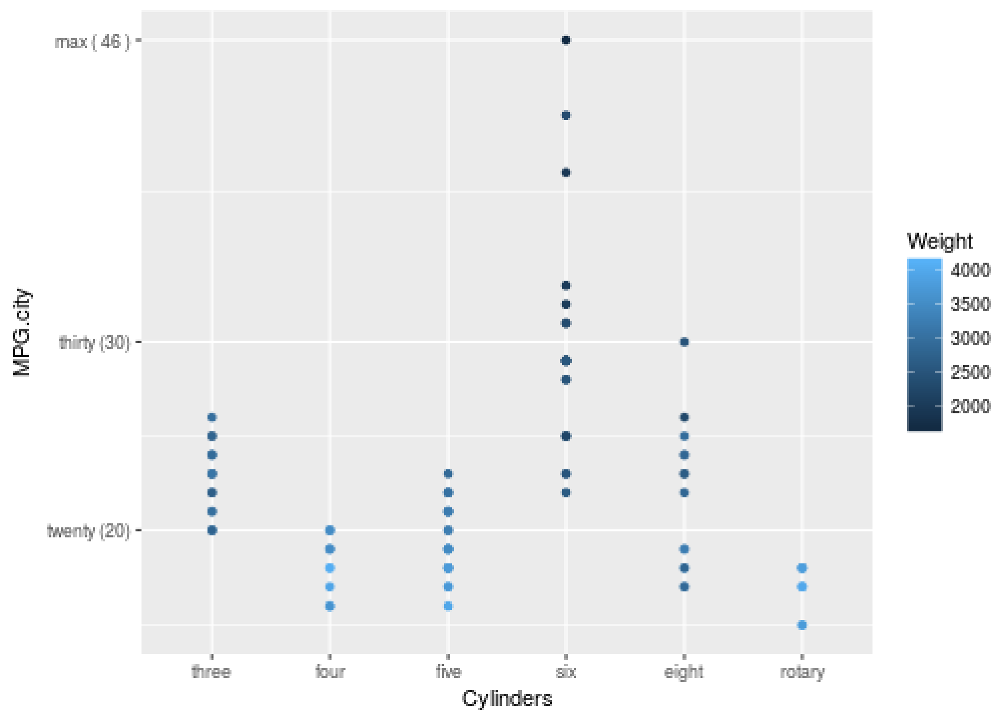
Example: Scales

```
g <- g + scale_x_discrete("Cylinders", labels = c("three","four","five","six","eight","rotary"))
g
```

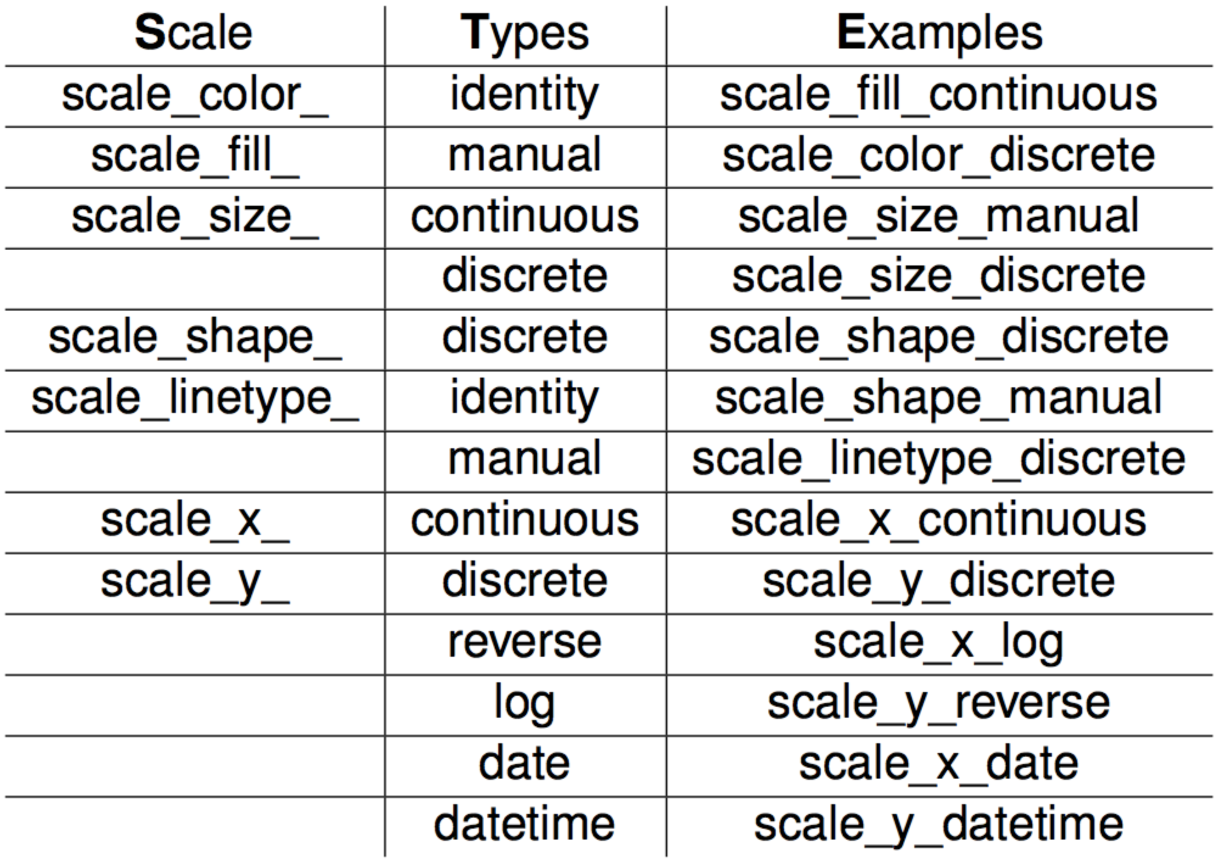


Example: Scales

```
m <- max(Cars93$MPG.city)
g <- g + scale_y_continuous(breaks=c(10,20,30,m), labels=c("ten (10)", "twenty (20)", "thirty (30)", paste("max
(",m,")")))
g
```



Overview: Scales



Scale	Types	Examples
scale_color_	identity	scale_fill_continuous
scale_fill_	manual	scale_color_discrete
scale_size_	continuous	scale_size_manual
	discrete	scale_size_discrete
scale_shape_	discrete	scale_shape_discrete
scale_linetype_	identity	scale_shape_manual
	manual	scale_linetype_discrete
scale_x_	continuous	scale_x_continuous
scale_y_	discrete	scale_y_discrete
	reverse	scale_x_log
	log	scale_y_reverse
	date	scale_x_date
	datetime	scale_y_datetime

Faceting

A *standardized* graphic for each group in the data.

- for one grouping variable: **facet_wrap()**
- for two grouping variables: **facet_grid()**

What happened on the Titanic? (1)

```
data(td)
str(td)
```

```
'data.frame':   59 obs. of  4 variables:
 $ pclass: int  1 1 1 1 1 1 1 1 1 1 ...
 $ age.g : int  1 1 2 2 3 3 4 4 5 5 ...
 $ sex   : Factor w/ 2 levels "female","male": 1 2 1 2 1 2 1 2 1 2 ...
 $ ps    : num  0 1 1 1 1 ...
```

```
head(td,12)
```

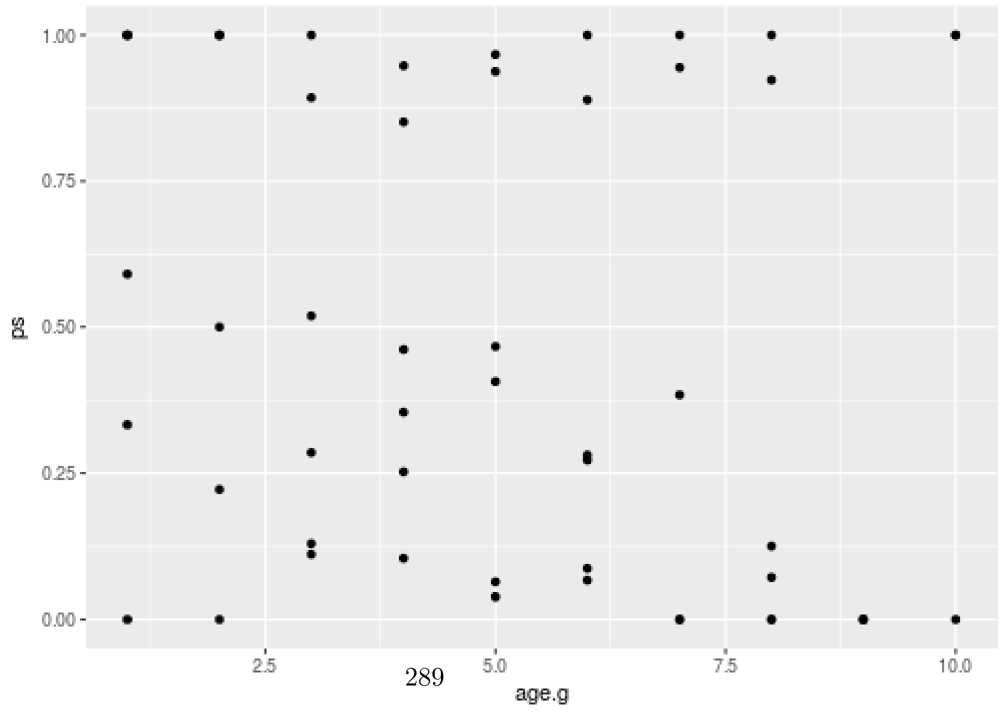
```
   pclass age.g    sex        ps
1       1     1 female 0.0000000
2       1     1   male 1.0000000
3       1     2 female 1.0000000
4       1     2   male 1.0000000
5       1     3 female 1.0000000
6       1     3   male 0.2857143
7       1     4 female 0.9473684
8       1     4   male 0.4615385
9       1     5 female 0.9666667
10      1     5   male 0.4074074
11      1     6 female 1.0000000
12      1     6   male 0.2812500
```

```
## note: ps = ratio of people survived
```

What happened on the Titanic? (2)

Scatterplot, two clusters are visible

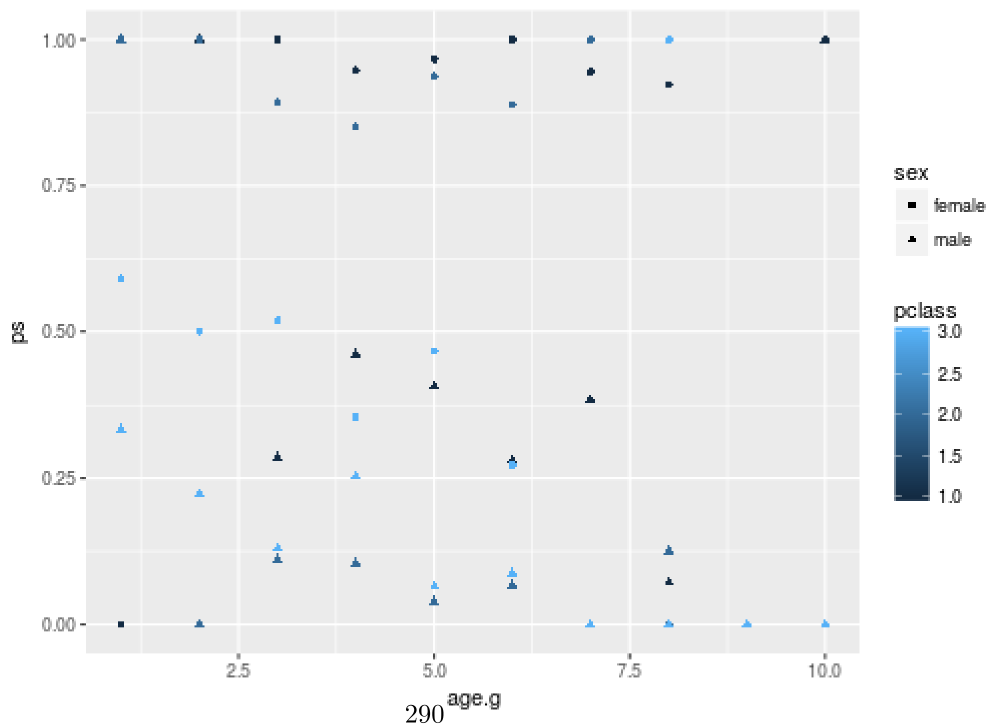
```
tdg <- ggplot(td, aes(x = age.g, y = ps))
tdg + geom_point()
```



What happened on the Titanic? (3)

what we already learned, aesthetic mapping…

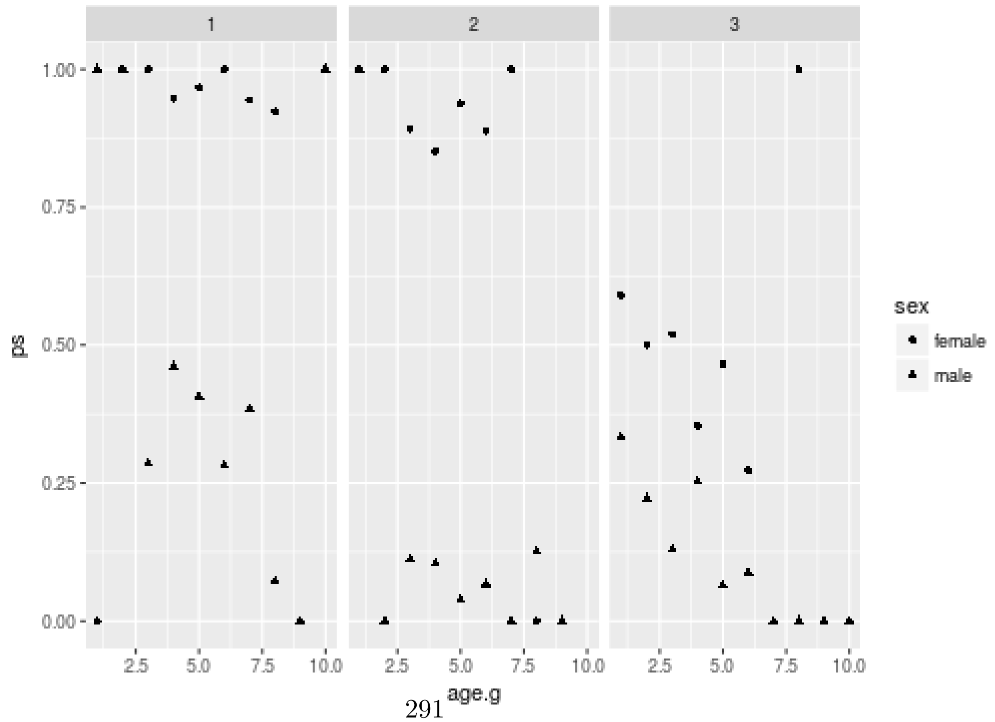
```
tdg + geom_point(aes(color = pclass, shape = sex))
```



What happened on the Titanic? (4)

faceting...

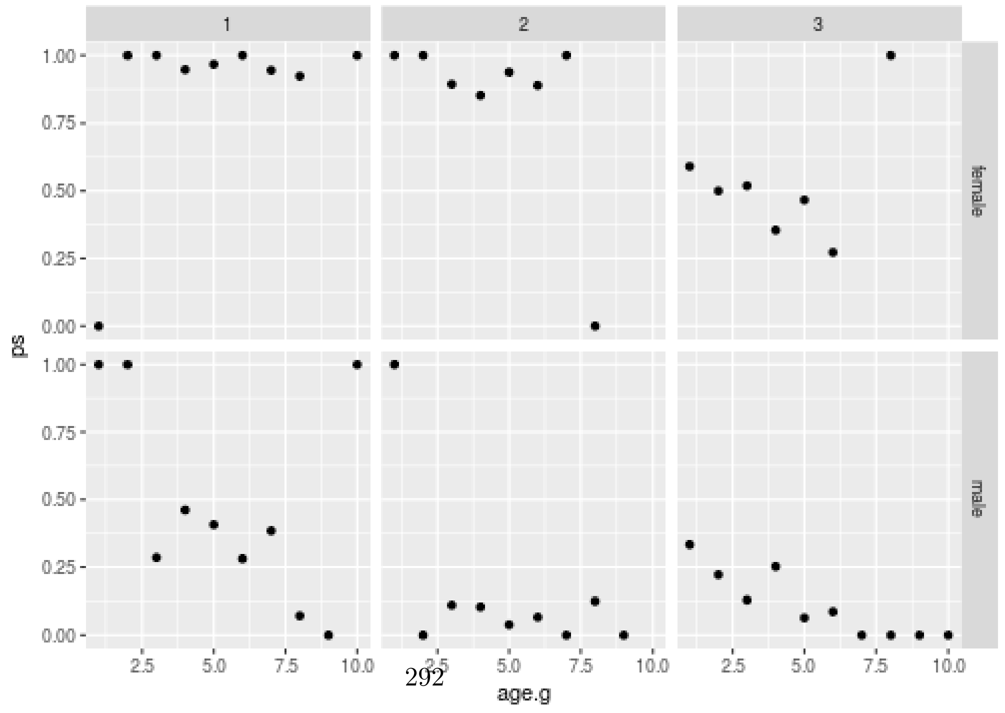
```
tdg + geom_point(aes(shape = sex)) + facet_wrap( ~ pclass)
```



What happened on the Titanic? (5)

faceting… with two grouping variables

```
tdg <- tdg + geom_point() + facet_grid(sex ~ pclass)
tdg
```

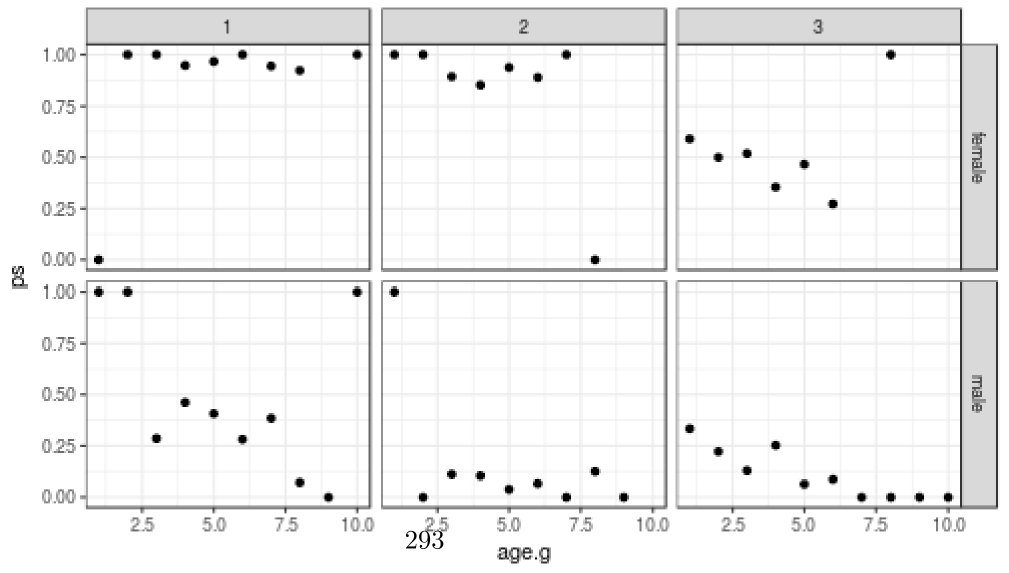


Example: Themes

Themes for cooperative design. Two themes which comes with ggplot2:

- **theme_gray()** - default
- **theme_bw()**

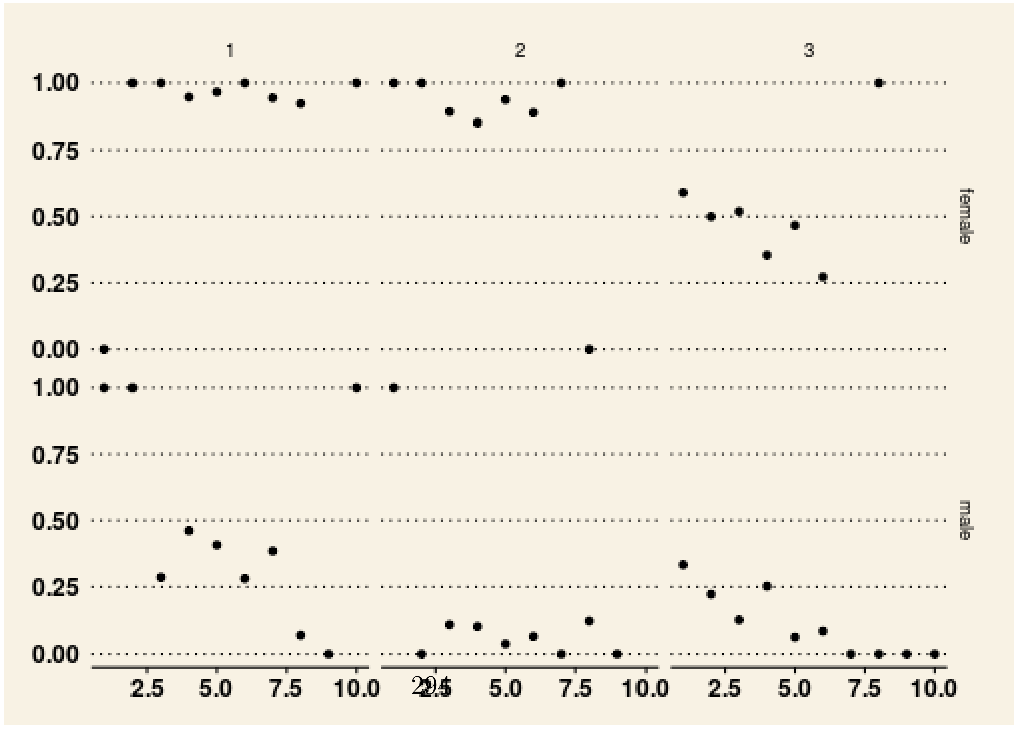
```
tdg + theme_bw()
```



Example: Themes

In the web you can find a number of user-generated themes

```r
library(ggthemes)
tdg + theme_wsj()
```

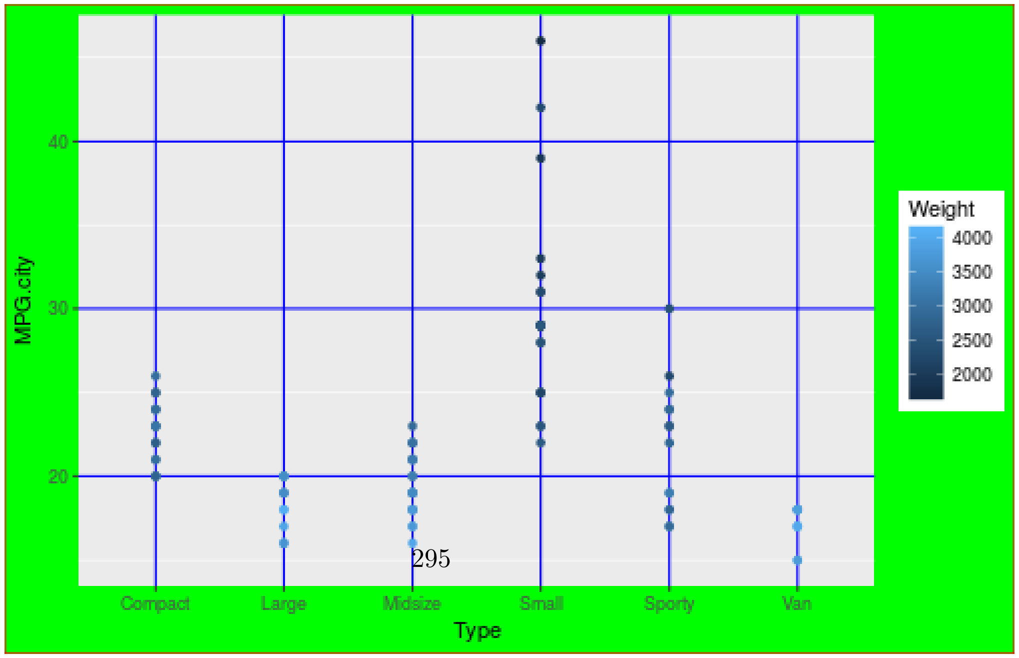


Eurostat

Example: Themes

type **theme_gray()** to see a list with options. Using function **theme()** you can modify them.

```
gg <- ggplot(Cars93, aes(x = Type, y = MPG.city))
gg <- gg + geom_point(aes(color = Weight))
gg + theme(plot.background = element_rect(fill = 'green', colour = 'red')) +  theme(panel.grid.major =
element_line(colour = "blue"))
```



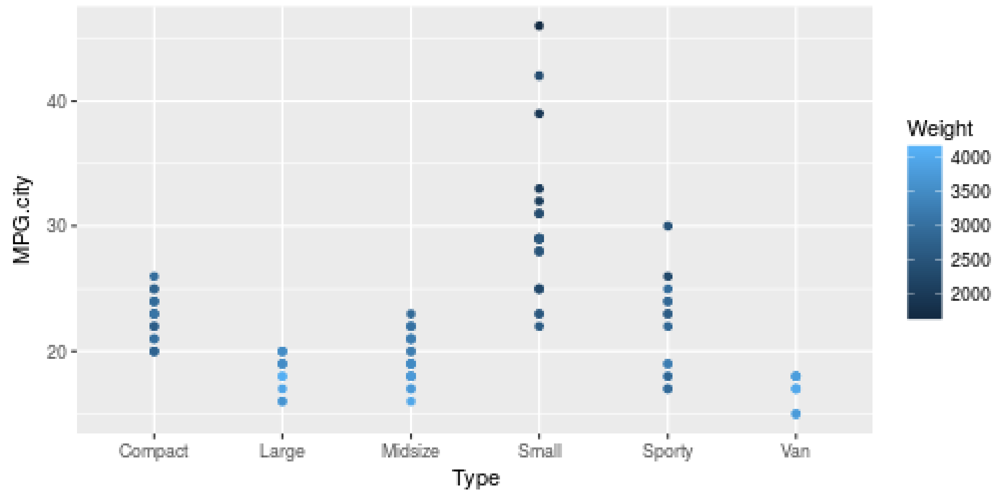
Example: Themes

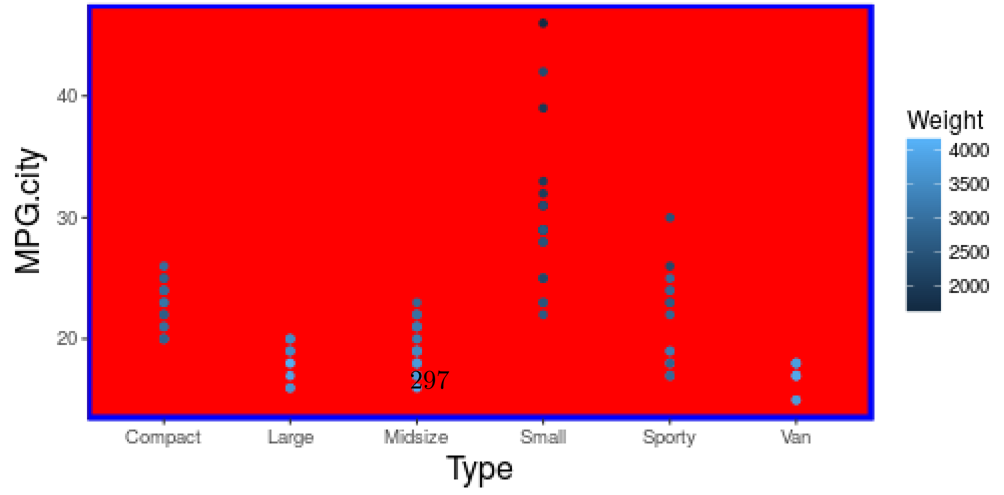
Create your own theme!

```
theme_new <- function(base_size = 12, base_family = "Helvetica"){
  theme_bw(base_size = base_size, base_family = base_family) %+replace%
    theme(
      axis.title = element_text(size = 16),
      legend.key=element_rect(colour=NA, fill =NA),
      panel.grid = element_blank(),
      panel.border = element_rect(fill = NA, colour = "blue", size=2),
      panel.background = element_rect(fill = "red", colour = "black"),
      strip.background = element_rect(fill = NA)
      )
}
```

Example: Themes

```
gg; gg + theme_new()
```





Tasks / Exercises

Time for practical training! :)

Please continue to work on Exercises 1x) and 2x).

Dynamic and automated reports with R

Alexander Kowarik, Bernhard Meindl

Overview / Objectives

- Presentation of the basic concepts of *full* reproducibility
- Options and tools that we discuss:
  - Formatting Output with **rmarkdown**
  - Preparation of final reports with **knitr**
- Example application with in *RStudio*
- A list of some alternatives

Reproducibility (1)

- An important goal: full **reproducibility**
  - Everything from the past should be able to recomputed at any time thereafter
- Reproducibility is often a necessity
  - in (medical) studies
  - for submissions in journals
- Required: organization and management of *text*, *code* and *graphics*
- Reproducibility should not only be within one researcher but should be made possible for all.
  - R as free software is an ideal candidate

Reproducibility (2) a general view

- Hotorn und Leisch (2011): case studies in reproducible research - 53 Papers, 17 providing data, 8 code, 6 come with data and code
- Reinhart und Rogoff (2011): *Growth in a Time of Debt* based on Excel-Sheets – errorprone. But it was used for world-wide policy decisions
- practice is often to link a dozen of Excel files without documenting changes of values
- copy & paste mistakes for reports and articles
- huge workload if numbers, graphics or tables changes (reports, web-publications,...)
- huge workload for periodical reports

... use dynamical reporting tools to raise quality and reduction of work-load:

Reproducibility (3)

- In **R**: Functionality of the package **knitr**
- **knitr** provides functionality for creating reproducible reports
  - Links **code** and **text** elements
  - The code is executed, the results embedded in the text
- Different output formats are possible
  - PDF output
  - HTML output (eg. in this presentation)
- Structuring: use of markdown within **R**

Why Markdown? (1)

- **rmarkdown** == **R** + **markdown**
- **markdown**: a markup language with many features
  - headings of different sizes
  - text formatting (bold, italic, strikethrough)
  - lists (ordered / unordered)
  - links, HTML, JavaScript
  - \(\LaTeX \) equations
  - tables
- **Aim**: to generate documents from plain text

Why Markdown? (2)

- Easy to learn and use
- Focus on content instead of code possible
- **Flexibility:** usual output is HTML
- With extra tools (*pandoc*) are other possible output formats (pdf, word)
- Automatically included in **RStudio**
  - **Packages** and **knitr rmarkdown**
  - Including help, automatic pre-view ...
- In addition:
  - HTML code can be included directly (and also Javascript)
  - Cooperate designs and style through CSS stylesheets

Markdown Basics (1)

- Headlines: 3 levels are defined

```
# Header 1
## Header 2
### Header 3
```

Header 1

Header 2

Header 3

- Bold text

```
this is a **bold** text, as well as __this__
```

this is a **bold** text, as well as **this**

Markdown Basics (2)

- Italic text

```
this is a *italic* text, as well as _this_
```

this is a *italic* text, as well as *this*

- Strikethrough text

```
this is a ~~marked as deleted~~  text
```

this is a ~~marked as deleted~~ text

Markdown Basics (3)

- Block Quotes

```
> ## A quote
> *Where faith fails, helps statistics* (Werner Ehrenforth)
```

results:

> ## A quote
>
> *Where faith fails, helps the statistics.* (Werner Ehrenforth)

Markdown Basics (4)

- Bulleted (unordered)

```
- Element 1
 * Element 1a
 + Element 1aa
* Element 2
 * Element 2a
 + Element 2aa
```

- Element 1
  - Element 1a
  - Element 1aa
- Element 2
  - Element 2a
  - Element 2aa

Markdown Basics (5)

- Bullet lists (ordered)

```
1. Element 1
 * Element 1a
 + Element 1aa
2. Element 2
 * Element 2a
 + Element 2aa
```

1. Element 1
    - Element 1a
    - Element 1aa
2. Element 2
    - Element 2a
    - Element 2aa

Markdown Basics (6)

- Hyperlinks

```
[link to r-project.org](http://r-project.org)
```

link to r-project.org

- Direct inserting HTML code

```
<img src = "http://www.r-project.org/Rlogo.jpg">
```

Markdown Basics (7)

- \(\LaTeX \) equations

```
$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i$
```

\(\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i \)

Markdown Basics (8)

- Tables can be created with the following Synatx

```
V1 | V2 | V3
---- | --- | ---
v1a | v2a | v3a
v1b | v2b | v3b
```

- Renders to

V1	V2	V3
v1a	v2a	v3a
v1b	v2b	v3b

- Styling the tables using stylesheet possible
- Easier with kable() from the knitr package (?kable)

R-Chunks (1)

- Inserting **R**- code
- General syntax: (shortcut in R-Studio: *Ctrl* - *Alt* - *i*)

~~~
```{r}
x <- mean (1:10); mean (x)
mean (x)
```
~~~

- is rendered as

```
x <- mean (1:10); mean (x)
```

```
[1] 5.5
```

- In code chunks many options can be set
  - echo: will be displayed and the code itself
  - eval: to code chunk to be evaluated
  - cache: computerinensive calculations cached?

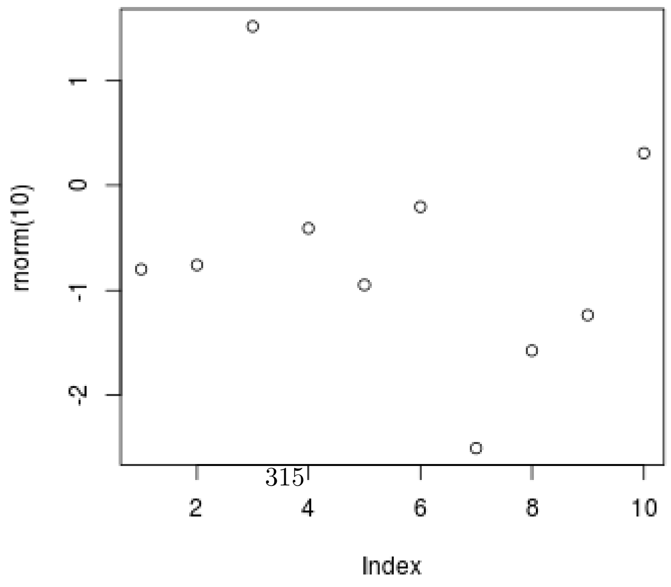
R chunks (2)

- Also graphics can be created directly in chunks.

~~~
```{r, echo = TRUE, fig = TRUE, fig.height = 5, fig.width = 5, fig.align ="center"}
plot (rnorm (10))
```
~~~

- renders to:

```
plot (rnorm (10))
```



Eurostat

R-Chunks (3)

- Chunks can be embedded directly into text.
- Example:

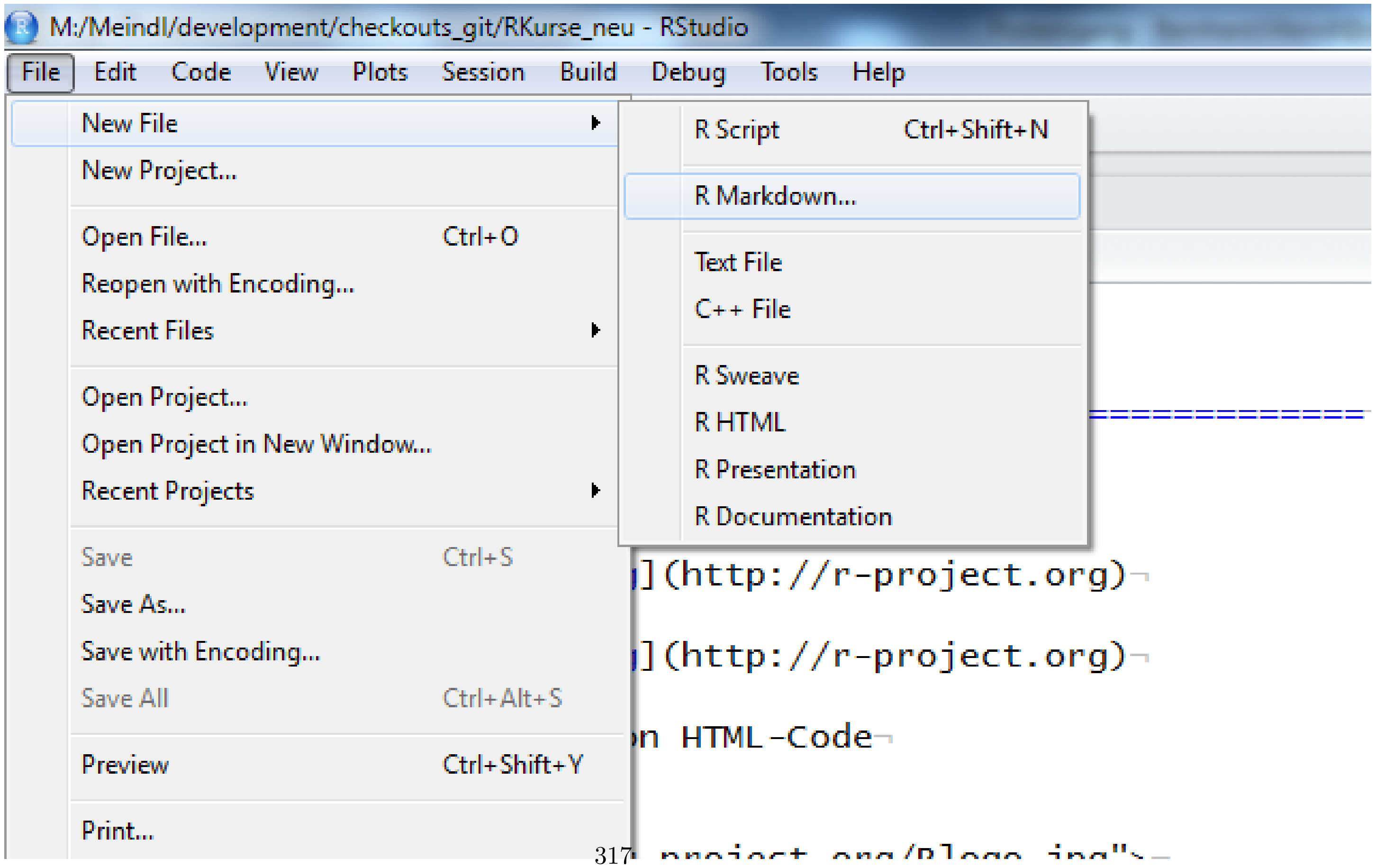
  A standard normally distributed random number: `` `r rnorm(1)` ``

- renders to

  A standard normally distributed random number: 0.3000526

Markdown in Rstudio (1)

- Create a new Markdown document


M:/Meindl/development/checkouts_git/RKurse_neu - RStudio

File Edit Code View Plots Session Build Debug Tools Help

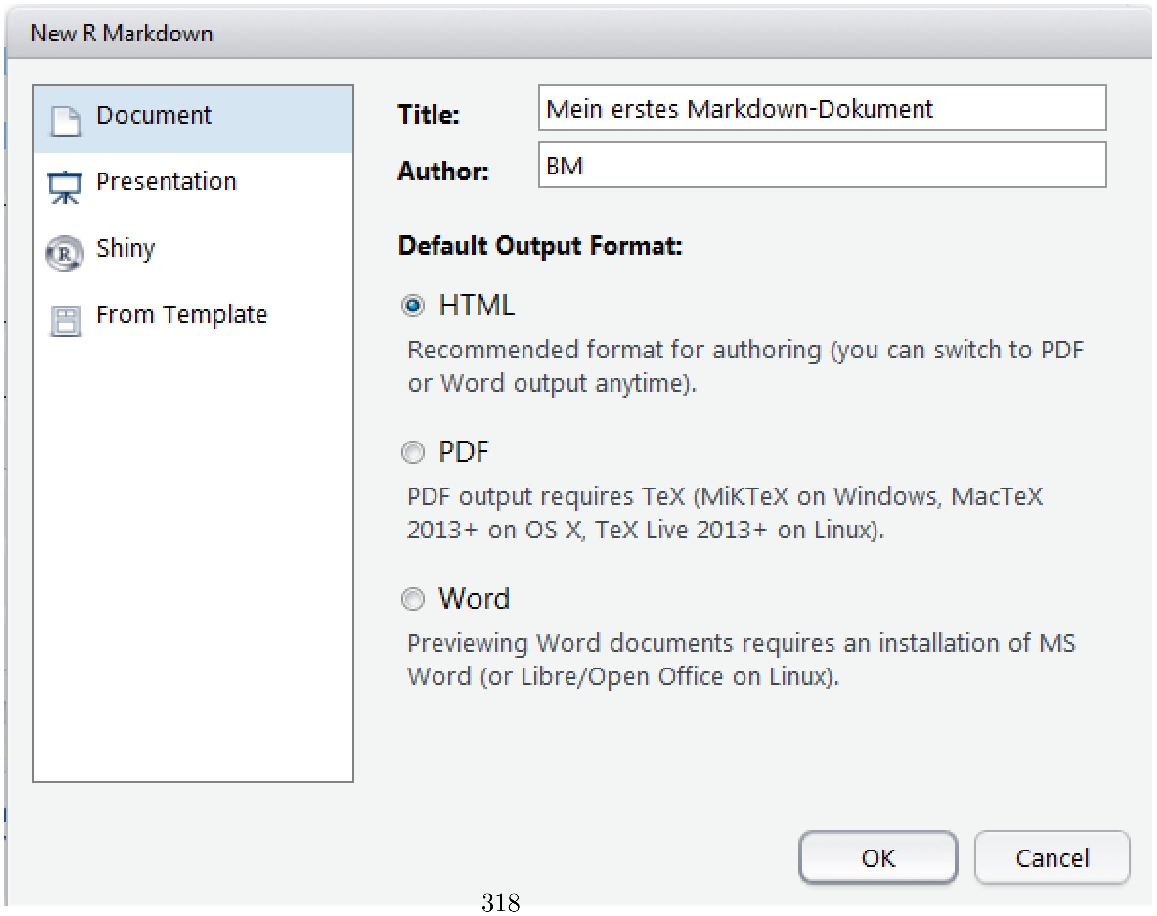
New File ▸	
New Project...	
Open File...	Ctrl+O
Reopen with Encoding...	
Recent Files ▸	
Open Project...	
Open Project in New Window...	
Recent Projects ▸	
Save	Ctrl+S
Save As...	
Save with Encoding...	
Save All	Ctrl+Alt+S
Preview	Ctrl+Shift+Y
Print...	

R Script	Ctrl+Shift+N
R Markdown...	
Text File	
C++ File	
R Sweave	
R HTML	
R Presentation	
R Documentation	

```
](http://r-project.org)¬
](http://r-project.org)¬
n HTML-Code¬
```

Markdown in Rstudio (2)

- Set output format and other options



Title: Mein erstes Markdown-Dokument

Author: BM

Default Output Format:

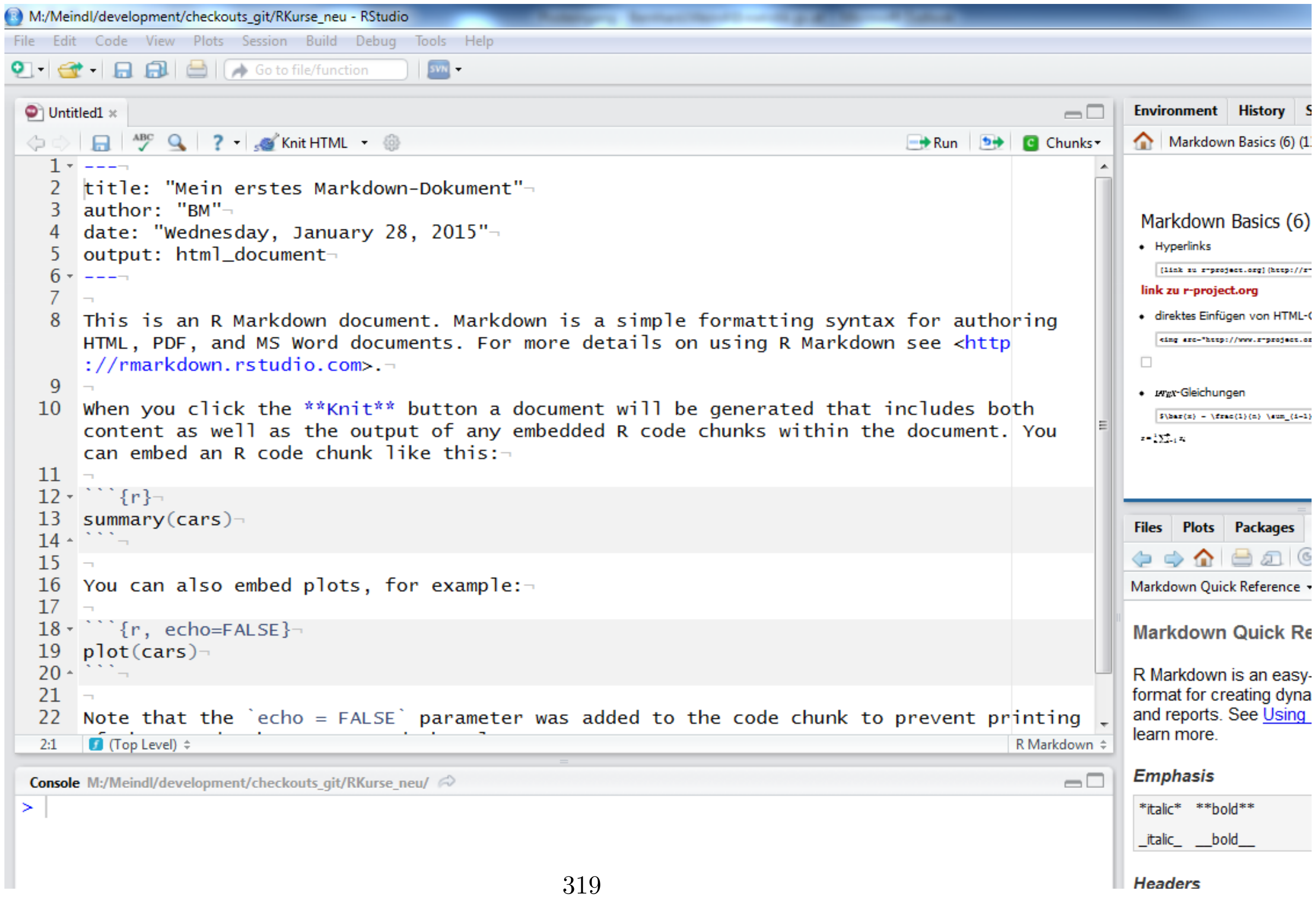
[✓ HTML]
Recommended format for authoring (you can switch to PDF or Word output anytime).

PDF
PDF output requires TeX (MiKTeX on Windows, MacTeX 2013+ on OS X, TeX Live 2013+ on Linux).

Word
Previewing Word documents requires an installation of MS Word (or Libre/Open Office on Linux).

Markdown in Rstudio (3)

- For a quick start is already defined and filled content



M:/Meindl/development/checkouts_git/RKurse_neu - RStudio
File Edit Code View Plots Session Build Debug Tools Help
Go to file/function
Untitled1
Knit HTML
Run
Chunks
Environment History
Markdown Basics (6)
Files Plots Packages
Markdown Quick Reference
Console M:/Meindl/development/checkouts_git/RKurse_neu/
2:1 (Top Level) R Markdown

~~~
---
title: "Mein erstes Markdown-Dokument"
author: "BM"
date: "Wednesday, January 28, 2015"
output: html_document
---

This is an R Markdown document. Markdown is a simple formatting syntax for authoring
HTML, PDF, and MS Word documents. For more details on using R Markdown see <http
://rmarkdown.rstudio.com>.

When you click the **Knit** button a document will be generated that includes both
content as well as the output of any embedded R code chunks within the document. You
can embed an R code chunk like this:

```{r}
summary(cars)
~~~

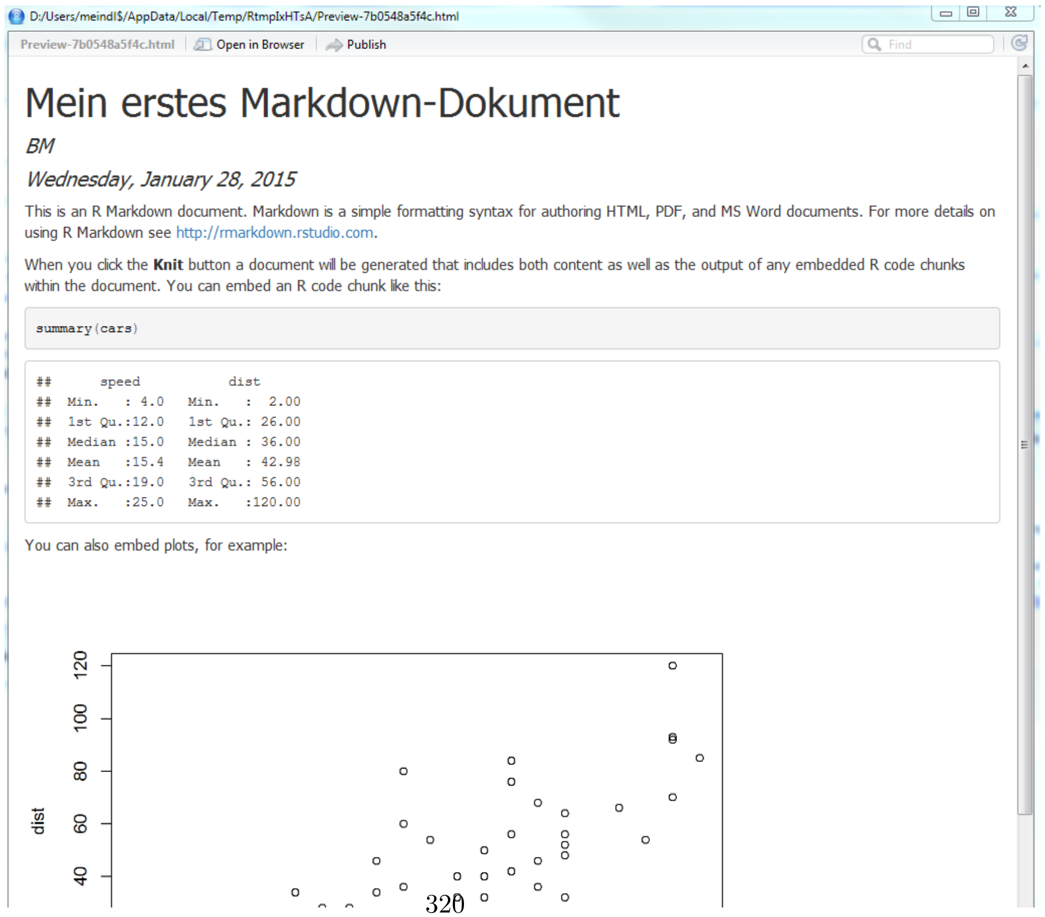
You can also embed plots, for example:

```{r, echo=FALSE}
plot(cars)
```

Note that the `echo = FALSE` parameter was added to the code chunk to prevent printing
```

Markdown Basics (6)

- Hyperlinks

`[link zu r-project.org](http://r-`

link zu r-project.org

- direktes Einfügen von HTML-

`<img src="http://www.r-project.o`

- LaTeX-Gleichungen

`$\bar{x} = \frac{1}{n} \sum_{i=1}`

Markdown Quick Re

R Markdown is an easy-
format for creating dyna
and reports. See Using
learn more.

Emphasis

italic	**bold**
italic	__bold__

Headers
```

# Markdown in Rstudio (4)

- Consider the HTML output in the preview window



D:/Users/meindl$/AppData/Local/Temp/RtmpIxHTsA/Preview-7b0548a5f4c.html

Mein erstes Markdown-Dokument

*BM*

*Wednesday, January 28, 2015*

This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details on using R Markdown see http://rmarkdown.rstudio.com.

When you click the **Knit** button a document will be generated that includes both content as well as the output of any embedded R code chunks within the document. You can embed an R code chunk like this:

```
summary(cars)
```

```
##      speed           dist       
##  Min.   : 4.0   Min.   :  2.00  
##  1st Qu.:12.0   1st Qu.: 26.00  
##  Median :15.0   Median : 36.00  
##  Mean   :15.4   Mean   : 42.98  
##  3rd Qu.:19.0   3rd Qu.: 56.00  
##  Max.   :25.0   Max.   :120.00  
```

You can also embed plots, for example:

Tips and Tricks (1)

- RStudio is helpful, but all can be done without RStudio as well
- rmarkdown files usually have the extension **rmd**
- workflow to generate Html: **rmd** -> **md** -> **html**
- Necessary packages are **knitr** and **rmarkdown**
- The button *knitHTML* in Rstudio is equivalent to

```
library(knitr); knit2html(input="index.rmd", output="index.html")
```

- Markdown code can be extracted from **input.rmd**

```
knit(input="index.rmd", output="index.md")
```

- Markdown code can be (including options) translate to Html

```
markdownToHTML(file="index.md", output="index.html", stylesheet="my.css")
```

Tips and Tricks (2)

- Extract code from rmarkdown with **purl()**
  - Very useful when debugging errors
  - Useful for documentation purposes of the code
- Application:

```
purl(input="input.rmd", output="input.r")
```

- Note: Short demo session with **demo.rmd**

```
knit(input = "demo.rmd") # create "demo.md"
markdownToHTML(file="demo.md", output="demo.html") # create "demo.html"
purl(input="demo.rmd") # creates "demo.R"
```

Tips and Tricks (3)

- There are some alternatives available
  - **Sweave**: it also allows to connect \(LaTeX \) and R code
  - **brew**: can be viewed as a templating framework to R-code
  - the package **odfWeave** works with LibreOffice files.
- But: **knitr** is (further) developed active and *the-way-to-go*

Tasks / Exercises

Time for practical training! :)

Please continue to work on Exercises 1x).

Reporting - Summary (1)

- rmarkdown:
  - Easy to use by very simple syntax
  - Focus on content possible
- Markdown is very well integrated in RStudio
- Documentation / reporting of R with rmarkdown helps to
  - to remain reproducible
  - avoid copy / paste error
- A helpful rmarkdown / knitr reference is available here

Data manipulation package dplyr

Alexander Kowarik, Bernhard Meindl

Overview / Objectives

Gain knowledge of basic data manipulation techniques -

- The package **dplyr** offers functions for:
  - Filtering of observations
  - Variable Selection
  - Recoding
  - Grouping
  - Aggregation (in groups)
- **Note:** other useful packages such as **reshape2**, **data.table**, **stringr** or **lubridate** are not covered now.

Data manipulation (general)

- Data management necessary but (time) consuming
- In **R**: many paths (packages) lead to Rome
- In *R base* all operations for data manipulation are supported
- **But** additional packages make sometimes life easier, and the calculation time can be considerable faster

Data manipulation (more)

- Some of the steps in data management can be *abstracted*
- These tasks are e.g:
  - Selection of rows or columns
  - Ordering
  - Recodenig
  - Grouping
  - Aggregation
- using package dplyr, all these challenges can be met!

dplyr - Reasons for an additional package

- only **few important keywords** to remember
- Consistency
- Works with different input
  - *data.frame, data.tables, sqlite*
- Simple (but new) syntax
- Less code, less error?
- From now on, the following applies:
  - a **column** corresponds to a **variable**
  - a **line** corresponds to a **observation**

Warm-up (1)

- First, the package must be loaded

```
require(dplyr, quiet = TRUE)
```

- Some vignettes (short instructions) available

```
# help(pa = "dplyr")
```

- In the following we use the Cars93 data

```
data(Cars93, package="MASS") # ?Cars93
```

Warm-up (2)

- A brief inspection of the data

```
print(head(Cars93, 3))
```

```
  Manufacturer   Model    Type Min.Price Price Max.Price MPG.city
1        Acura Integra   Small      12.9  15.9      18.8       25
2        Acura  Legend Midsize      29.2  33.9      38.7       18
3         Audi      90 Compact      25.9  29.1      32.3       20
  MPG.highway            AirBags DriveTrain Cylinders EngineSize
1          31               None      Front         4        1.8
2          25 Driver & Passenger      Front         6        3.2
3          26        Driver only      Front         6        2.8
  Horsepower  RPM Rev.per.mile Man.trans.avail Fuel.tank.capacity
1        140 6300         2890             Yes               13.2
2        200 5500         2335             Yes               18.0
3        172 5500         2280             Yes               16.9
  Passengers Length Wheelbase Width Turn.circle Rear.seat.room
1          5    177       102    68          37           26.5
2          5    195       115    71          38           30.0
3          5    180       102    67          37           28.0
  Luggage.room Weight  Origin           Make
1           11   2705 non-USA Acura Integra
2           15   3560 non-USA  Acura Legend
3           14   3375 non-USA        Audi 90
```

Warm-up (3)

- Brief description of the first variables

```
summary(Cars93[,1:14])
```

```
    Manufacturer      Model         Type      Min.Price         Price      
 Chevrolet: 8    100     : 1   Compact:16   Min.   : 6.70   Min.   : 7.40  
 Ford     : 8    190E    : 1   Large  :11   1st Qu.:10.80   1st Qu.:12.20  
 Dodge    : 6    240     : 1   Midsize:22   Median :14.70   Median :17.70  
 Mazda    : 5    300E    : 1   Small  :21   Mean   :17.13   Mean   :19.51  
 Pontiac  : 5    323     : 1   Sporty :14   3rd Qu.:20.30   3rd Qu.:23.30  
 Buick    : 4    535i    : 1   Van    : 9   Max.   :45.40   Max.   :61.90  
 (Other)  :57    (Other):87                                                
   Max.Price       MPG.city      MPG.highway                  AirBags  
 Min.   : 7.9   Min.   :15.00   Min.   :20.00   Driver & Passenger:16  
 1st Qu.:14.7   1st Qu.:18.00   1st Qu.:26.00   Driver only       :43  
 Median :19.6   Median :21.00   Median :28.00   None              :34  
 Mean   :21.9   Mean   :22.37   Mean   :29.09                          
 3rd Qu.:25.3   3rd Qu.:25.00   3rd Qu.:31.00                          
 Max.   :80.0   Max.   :46.00   Max.   :50.00                          
                                                                       
 DriveTrain  Cylinders    EngineSize      Horsepower         RPM      
 4WD  :10   3      : 3   Min.   :1.000   Min.   : 55.0   Min.   :3800  
 Front:67   4      :49   1st Qu.:1.800   1st Qu.:103.0   1st Qu.:4800  
 Rear :16   5      : 2   Median :2.400   Median :140.0   Median :5200  
            6      :31   Mean   :2.668   Mean   :143.8   Mean   :5281  
            8      : 7   3rd Qu.:3.300   3rd Qu.:170.0   3rd Qu.:5750  
            rotary: 1   Max.   :5.700   Max.   :300.0   Max.   :6500  
```

Local Data Frame (1)

- With tbl_df(), a *local* data frame to be created
- Why do we need this?
  - Improved, efficient output (print -method)
  - No accidental print of huge data sets
- Remember Cars93 is a *data.frame*

```
class(Cars93)
```

```
[1] "data.frame"
```

- We convert to a *local* data frame for dplyr…

```
Cars93 <- tbl_df(Cars93)
class(Cars93)
```

```
[1] "tbl_df"     "tbl"        "data.frame"
```

Local Data Frame (2)

class - methods for this class are implemented

```
print(Cars93)
```

```
# A tibble: 93 × 27
   Manufacturer      Model    Type Min.Price Price Max.Price MPG.city
*        <fctr>     <fctr>  <fctr>     <dbl> <dbl>     <dbl>    <int>
1         Acura    Integra   Small      12.9  15.9      18.8       25
2         Acura     Legend Midsize      29.2  33.9      38.7       18
3          Audi         90 Compact      25.9  29.1      32.3       20
4          Audi        100 Midsize      30.8  37.7      44.6       19
5           BMW       535i Midsize      23.7  30.0      36.2       22
6         Buick    Century Midsize      14.2  15.7      17.3       22
7         Buick    LeSabre   Large      19.9  20.8      21.7       19
8         Buick Roadmaster   Large      22.6  23.7      24.9       16
9         Buick    Riviera Midsize      26.3  26.3      26.3       19
10     Cadillac    DeVille   Large      33.0  34.7      36.3       16
# ... with 83 more rows, and 20 more variables: MPG.highway <int>,
#   AirBags <fctr>, DriveTrain <fctr>, Cylinders <fctr>, EngineSize <dbl>,
#   Horsepower <int>, RPM <int>, Rev.per.mile <int>,
#   Man.trans.avail <fctr>, Fuel.tank.capacity <dbl>, Passengers <int>,
#   Length <int>, Wheelbase <int>, Width <int>, Turn.circle <int>,
#   Rear.seat.room <dbl>, Luggage.room <int>, Weight <int>, Origin <fctr>,
#   Make <fctr>
```

- output of *print() now looks different

Extracting rows (1)

- Using function **slice()** you can select rows according to their line number

```
slice(Cars93, 1:2) # first two observations
```

```
# A tibble: 2 × 27
  Manufacturer   Model    Type Min.Price Price Max.Price MPG.city
        <fctr>  <fctr>  <fctr>     <dbl> <dbl>     <dbl>    <int>
1        Acura Integra   Small      12.9  15.9      18.8       25
2        Acura  Legend Midsize      29.2  33.9      38.7       18
# ... with 20 more variables: MPG.highway <int>, AirBags <fctr>,
#   DriveTrain <fctr>, Cylinders <fctr>, EngineSize <dbl>,
#   Horsepower <int>, RPM <int>, Rev.per.mile <int>,
#   Man.trans.avail <fctr>, Fuel.tank.capacity <dbl>, Passengers <int>,
#   Length <int>, Wheelbase <int>, Width <int>, Turn.circle <int>,
#   Rear.seat.room <dbl>, Luggage.room <int>, Weight <int>, Origin <fctr>,
#   Make <fctr>
```

Extracting rows (2)

- Function n() returns the number of observations (rows)

```
slice(Cars93, n()) # shows the last observation
```

```
# A tibble: 1 × 27
  Manufacturer  Model    Type Min.Price Price Max.Price MPG.city
        <fctr> <fctr>  <fctr>     <dbl> <dbl>     <dbl>    <int>
1        Volvo    850 Midsize      24.8  26.7      28.5       20
# ... with 20 more variables: MPG.highway <int>, AirBags <fctr>,
#   DriveTrain <fctr>, Cylinders <fctr>, EngineSize <dbl>,
#   Horsepower <int>, RPM <int>, Rev.per.mile <int>,
#   Man.trans.avail <fctr>, Fuel.tank.capacity <dbl>, Passengers <int>,
#   Length <int>, Wheelbase <int>, Width <int>, Turn.circle <int>,
#   Rear.seat.room <dbl>, Luggage.room <int>, Weight <int>, Origin <fctr>,
#   Make <fctr>
```

Extracting rows (3)

- You can also select multiple rows at once
- Note: c() creates a vector from the input numbers
- We select the 1,4,10,15 and last line of the data

```
slice(Cars93, c(1,4,10,15,n()))
```

```
# A tibble: 5 × 27
  Manufacturer   Model    Type Min.Price Price Max.Price MPG.city
        <fctr>  <fctr>  <fctr>     <dbl> <dbl>     <dbl>    <int>
1        Acura Integra   Small      12.9  15.9      18.8       25
2         Audi     100 Midsize      30.8  37.7      44.6       19
3     Cadillac DeVille   Large      33.0  34.7      36.3       16
4    Chevrolet  Lumina Midsize      13.4  15.9      18.4       21
5        Volvo     850 Midsize      24.8  26.7      28.5       20
# ... with 20 more variables: MPG.highway <int>, AirBags <fctr>,
#   DriveTrain <fctr>, Cylinders <fctr>, EngineSize <dbl>,
#   Horsepower <int>, RPM <int>, Rev.per.mile <int>,
#   Man.trans.avail <fctr>, Fuel.tank.capacity <dbl>, Passengers <int>,
#   Length <int>, Wheelbase <int>, Width <int>, Turn.circle <int>,
#   Rear.seat.room <dbl>, Luggage.room <int>, Weight <int>, Origin <fctr>,
#   Make <fctr>
```

Filtering using a condition

- The function **filter()** can be selected rows that satisfy a condition:
- Example: all observations where variable *Manufacturer* == is *Audi* and at the same time the value of variable *Min.Price* > *25* is.

```
filter(Cars93, Manufacturer=="Audi" & Min.Price > 25)
```

```
# A tibble: 2 × 27
  Manufacturer  Model    Type Min.Price Price Max.Price MPG.city
        <fctr> <fctr>  <fctr>     <dbl> <dbl>     <dbl>    <int>
1         Audi     90 Compact      25.9  29.1      32.3       20
2         Audi    100 Midsize      30.8  37.7      44.6       19
# ... with 20 more variables: MPG.highway <int>, AirBags <fctr>,
#   DriveTrain <fctr>, Cylinders <fctr>, EngineSize <dbl>,
#   Horsepower <int>, RPM <int>, Rev.per.mile <int>,
#   Man.trans.avail <fctr>, Fuel.tank.capacity <dbl>, Passengers <int>,
#   Length <int>, Wheelbase <int>, Width <int>, Turn.circle <int>,
#   Rear.seat.room <dbl>, Luggage.room <int>, Weight <int>, Origin <fctr>,
#   Make <fctr>
```

- Note: the condition can be arbitrarily complex (& , |)

Ordering (1)

- With arrange() you can sort the data by one or more variables
- By default is sorted in ascending order, with desc() descending

```
Cars93 <- arrange(Cars93, Price); head(Cars93, 15)
```

```
# A tibble: 15 × 27
   Manufacturer   Model   Type Min.Price Price Max.Price MPG.city
         <fctr>  <fctr> <fctr>     <dbl> <dbl>     <dbl>    <int>
1          Ford Festiva  Small       6.9   7.4       7.9       31
2       Hyundai   Excel  Small       6.8   8.0       9.2       29
3         Mazda     323  Small       7.4   8.3       9.1       29
4           Geo   Metro  Small       6.7   8.4      10.0       46
5        Subaru   Justy  Small       7.3   8.4       9.5       33
6        Suzuki   Swift  Small       7.3   8.6      10.0       39
7       Pontiac  LeMans  Small       8.2   9.0       9.9       31
8    Volkswagen     Fox  Small       8.7   9.1       9.5       25
9         Dodge    Colt  Small       7.9   9.2      10.6       29
10       Toyota  Tercel  Small       7.8   9.8      11.8       32
11      Hyundai Elantra  Small       9.0  10.0      11.0       22
12      Hyundai  Scoupe Sporty       9.1  10.0      11.0       26
13         Ford  Escort  Small       8.4  10.1      11.9       23
14   Mitsubishi  Mirage  Small       7.7  10.3      12.9       29
15       Subaru  Loyale  Small      10.5  10.9      11.3       25
# ... with 20 more variables: MPG.highway <int>, AirBags <fctr>,
#   DriveTrain <fctr>, Cylinders <fctr>, EngineSize <dbl>,
#   Horsepower <int>, RPM <int>, Rev.per.mile <int>,
#   Man.trans.avail <fctr>, Fuel.tank.capacity <dbl>, Passengers <int>,
#   Length <int>, Wheelbase <int>, Width <int>, Turn.circle <int>,
#   Rear.seat.room <dbl>, Luggage.room <int>, Weight <int>, Origin <fctr>,
#   Make <fctr>
```

Ordering (2)

- You can also sort by multiple variables

```
head(arrange(Cars93, desc(MPG.city), Max.Price), 15)
```

```
# A tibble: 15 × 27
   Manufacturer   Model   Type Min.Price Price Max.Price MPG.city
         <fctr>  <fctr> <fctr>     <dbl> <dbl>     <dbl>    <int>
1           Geo   Metro  Small       6.7   8.4      10.0       46
2         Honda   Civic  Small       8.4  12.1      15.8       42
3        Suzuki   Swift  Small       7.3   8.6      10.0       39
4        Subaru   Justy  Small       7.3   8.4       9.5       33
5        Toyota  Tercel  Small       7.8   9.8      11.8       32
6          Ford Festiva  Small       6.9   7.4       7.9       31
7       Pontiac  LeMans  Small       8.2   9.0       9.9       31
8           Geo   Storm Sporty      11.5  12.5      13.5       30
9         Mazda     323  Small       7.4   8.3       9.1       29
10      Hyundai   Excel  Small       6.8   8.0       9.2       29
11        Dodge    Colt  Small       7.9   9.2      10.6       29
12   Mitsubishi  Mirage  Small       7.7  10.3      12.9       29
13       Nissan  Sentra  Small       8.7  11.8      14.9       29
14        Eagle  Summit  Small       7.9  12.2      16.5       29
15        Mazda Protege  Small      10.9  11.6      12.3       28
# ... with 20 more variables: MPG.highway <int>, AirBags <fctr>,
#   DriveTrain <fctr>, Cylinders <fctr>, EngineSize <dbl>,
#   Horsepower <int>, RPM <int>, Rev.per.mile <int>,
#   Man.trans.avail <fctr>, Fuel.tank.capacity <dbl>, Passengers <int>,
#   Length <int>, Wheelbase <int>, Width <int>, Turn.circle <int>,
#   Rear.seat.room <dbl>, Luggage.room <int>, Weight <int>, Origin <fctr>,
#   Make <fctr>
```

Selection of variables (1)

- Function **select()** allows you to select variables from the data

```
head(select(Cars93, Manufacturer, Price), 3)
```

```
# A tibble: 3 × 2
  Manufacturer Price
        <fctr> <dbl>
1         Ford   7.4
2      Hyundai   8.0
3        Mazda   8.3
```

- Sequence of variables operator **:** selectable

```
head(select(Cars93, Manufacturer:Price), 3)
```

```
# A tibble: 3 × 5
  Manufacturer   Model   Type Min.Price Price
        <fctr>  <fctr> <fctr>     <dbl> <dbl>
1         Ford Festiva  Small       6.9   7.4
2      Hyundai   Excel  Small       6.8   8.0
3        Mazda     323  Small       7.4   8.3
```

Selection of variables (2)

- Negative indexing possible, while all variables letter prefix minus (-) away

```
head(select(Cars93, -Min.Price, -Max.Price), 3)
```

```
# A tibble: 3 × 25
  Manufacturer   Model   Type Price MPG.city MPG.highway AirBags
        <fctr>  <fctr> <fctr> <dbl>    <int>       <int>  <fctr>
1         Ford Festiva  Small   7.4       31          33    None
2      Hyundai   Excel  Small   8.0       29          33    None
3        Mazda     323  Small   8.3       29          37    None
# ... with 18 more variables: DriveTrain <fctr>, Cylinders <fctr>,
#   EngineSize <dbl>, Horsepower <int>, RPM <int>, Rev.per.mile <int>,
#   Man.trans.avail <fctr>, Fuel.tank.capacity <dbl>, Passengers <int>,
#   Length <int>, Wheelbase <int>, Width <int>, Turn.circle <int>,
#   Rear.seat.room <dbl>, Luggage.room <int>, Weight <int>, Origin <fctr>,
#   Make <fctr>
```

Selection of variables (3)

- Special functions within select() :
  - starts_with(), ends_with()
  - contains()
  - matches()
  - num_range()

Selection of variables (4)

```
head(select(Cars93, starts_with("Man")), 2)
```

```
# A tibble: 2 × 2
  Manufacturer Man.trans.avail
        <fctr>          <fctr>
1         Ford             Yes
2      Hyundai             Yes
```

```
head(select(Cars93, contains("Price")), 2)
```

```
# A tibble: 2 × 3
  Min.Price Price Max.Price
      <dbl> <dbl>     <dbl>
1       6.9   7.4       7.9
2       6.8   8.0       9.2
```

```
head(select(Cars93, -contains("Price")), 2)
```

```
# A tibble: 2 × 24
  Manufacturer   Model   Type MPG.city MPG.highway AirBags DriveTrain
        <fctr>  <fctr> <fctr>    <int>       <int>  <fctr>     <fctr>
1         Ford Festiva  Small       31          33    None      Front
2      Hyundai   Excel  Small       29          33    None      Front
# ... with 17 more variables: Cylinders <fctr>, EngineSize <dbl>,
#   Horsepower <int>, RPM <int>, Rev.per.mile <int>,
#   Man.trans.avail <fctr>, Fuel.tank.capacity <dbl>, Passengers <int>,
#   Length <int>, Wheelbase <int>, Width <int>, Turn.circle <int>,
#   Rear.seat.room <dbl>, Luggage.room <int>, Weight <int>, Origin <fctr>,
#   Make <fctr>
```

Renaming variables (1)

- Both **select()** and **rename()** can be used to rename
- Simple *new* = *old* syntax
  - **select()** returns only the specified variables

```
head(select(Cars93, myPrize = Price, Min.Price))
```

```
# A tibble: 6 × 2
  myPrize Min.Price
    <dbl>     <dbl>
1     7.4       6.9
2     8.0       6.8
3     8.3       7.4
4     8.4       6.7
5     8.4       7.3
6     8.6       7.3
```

Renaming variables (2)

- **rename()** returns all variables

```
head(rename(Cars93, Manu2 = Manufacturer))
```

```
# A tibble: 6 × 27
    Manu2   Model   Type Min.Price Price Max.Price MPG.city MPG.highway
   <fctr>  <fctr> <fctr>     <dbl> <dbl>     <dbl>    <int>       <int>
1    Ford Festiva  Small       6.9   7.4       7.9       31          33
2 Hyundai   Excel  Small       6.8   8.0       9.2       29          33
3   Mazda     323  Small       7.4   8.3       9.1       29          37
4     Geo   Metro  Small       6.7   8.4      10.0       46          50
5  Subaru   Justy  Small       7.3   8.4       9.5       33          37
6  Suzuki   Swift  Small       7.3   8.6      10.0       39          43
# ... with 19 more variables: AirBags <fctr>, DriveTrain <fctr>,
#   Cylinders <fctr>, EngineSize <dbl>, Horsepower <int>, RPM <int>,
#   Rev.per.mile <int>, Man.trans.avail <fctr>, Fuel.tank.capacity <dbl>,
#   Passengers <int>, Length <int>, Wheelbase <int>, Width <int>,
#   Turn.circle <int>, Rear.seat.room <dbl>, Luggage.room <int>,
#   Weight <int>, Origin <fctr>, Make <fctr>
```

Tasks / Exercises

Time for practical training! :)

Please continue to work on Exercises 1x).

Uniqueness (1)

- **distinct()** can be used to keep only unique rows.

```
Cars93_1 <- select(Cars93, Manufacturer, EngineSize)
dim(Cars93_1)
```

```
[1] 93  2
```

```
Cars93_1 <- distinct(Cars93_1); dim(Cars93_1)
```

```
[1] 79  2
```

- By default, all variables are used to assess whether a row *multiple* occurs in the data set

Uniqueness (2)

- We can specify (calculated) variables that should be used as **keys** when calculating distinct data sets.

```
dim(Cars93)
```

```
[1] 93 27
```

```
dim(distinct(Cars93, Manufacturer))
```

```
[1] 32  1
```

```
dim(distinct(Cars93, Manufacturer, EngineSize)) # EngineSize as is
```

```
[1] 79  2
```

```
dim(distinct(Cars93, Manufacturer, rr=round(EngineSize))) # EngineSize is rounded
```

```
[1] 57  2
```

Creating variables (1)

- **mutate()**: adds new variables are added and retains the old

```
m <- mutate(Cars93, is_manu = Manufacturer == "Ford")
m[1:3, c(1,28)]
```

```
# A tibble: 3 × 2
  Manufacturer is_manu
        <fctr>   <lgl>
1         Ford    TRUE
2      Hyundai   FALSE
3        Mazda   FALSE
```

- **transmute()**: retains only the listed variables

```
head(transmute(Cars93, is_manu = Manufacturer == "Ford", Manufacturer), 3)
```

```
# A tibble: 3 × 2
  is_manu Manufacturer
    <lgl>       <fctr>
1    TRUE         Ford
2   FALSE      Hyundai
3   FALSE        Mazda
```

Creating variables (2)

- Newly created variables can be used again in the same statement

```
head(transmute(Cars93,
  Manufacturer,
  is_manu = Manufacturer == "Ford",
  num = ifelse(is_manu, -1,1)), 15)
```

```
# A tibble: 15 × 3
   Manufacturer is_manu   num
         <fctr>   <lgl> <dbl>
1          Ford    TRUE    -1
2       Hyundai   FALSE     1
3         Mazda   FALSE     1
4           Geo   FALSE     1
5        Subaru   FALSE     1
6        Suzuki   FALSE     1
7       Pontiac   FALSE     1
8    Volkswagen   FALSE     1
9         Dodge   FALSE     1
10       Toyota   FALSE     1
11      Hyundai   FALSE     1
12      Hyundai   FALSE     1
13         Ford    TRUE    -1
14   Mitsubishi   FALSE     1
15       Subaru   FALSE     1
```

Grouping and Aggregation (1)

- Often wants to perform calculations in *groups*
- Is often performed not very elegant
- However, simple syntax in the package **dplyr**
- It can not use the **group_by()** and **sumarise()** be used
  - **group_by()** creates a clustered data set
  - **summarize()** is used to calculate a statistics that provide exactly one number.

Grouping and Aggregation (2)

- The statistics for **summarize()** of *base-R*:
  - **sum()**, **mean()**, **median()**, **sd()**,…
- **dplyr** provides additional useful aggregation statistics
  - **n():** … number of observations per group
  - **first_value(x)**, **last_value(x)**, **nth_value(x)**: first, last, nth value of a variable *x*

Grouping and Aggregation (3)

- Grouping by variable *Manufacturer* and calculation of:
  - group size
  - the minimum of the variables *Prize*
  - the maximum of the variable *Prize*

```
by_type <- group_by(Cars93, Type)
summarize(by_type, count = n(), min_es = min(EngineSize), max_es = max(EngineSize))
```

```
# A tibble: 6 × 4
     Type count min_es max_es
   <fctr> <int>  <dbl>  <dbl>
1 Compact    16    2.0    3.0
2   Large    11    3.3    5.7
3 Midsize    22    2.0    4.6
4   Small    21    1.0    2.2
5  Sporty    14    1.3    5.7
6     Van     9    2.4    4.3
```

Grouping and Aggregation (4)

- via **group_by()** functions are applied on defined groups
- note: **arrange()** and **select()** are independent of grouping
- **Example**: report the first two observations per group

```
by_type <- group_by(Cars93, Type)
slice(by_type, 1:2)
```

```
Source: local data frame [12 x 27]
Groups: Type [6]

   Manufacturer      Model    Type Min.Price Price Max.Price MPG.city
         <fctr>     <fctr>  <fctr>     <dbl> <dbl>     <dbl>    <int>
1       Pontiac    Sunbird Compact       9.4  11.1      12.8       23
2          Ford      Tempo Compact      10.4  11.3      12.2       22
3      Chrylser   Concorde   Large      18.4  18.4      18.4       20
4     Chevrolet    Caprice   Large      18.0  18.8      19.6       17
5       Hyundai     Sonata Midsize      12.4  13.9      15.3       20
6       Mercury     Cougar Midsize      14.9  14.9      14.9       19
7          Ford    Festiva   Small       6.9   7.4       7.9       31
8       Hyundai      Excel   Small       6.8   8.0       9.2       29
9       Hyundai     Scoupe  Sporty       9.1  10.0      11.0       26
10          Geo      Storm  Sporty      11.5  12.5      13.5       30
11    Chevrolet Lumina_APV     Van      14.7  16.3      18.0       18
12    Chevrolet      Astro     Van      14.7  16.6      18.6       15
# ... with 20 more variables: MPG.highway <int>, AirBags <fctr>,
#   DriveTrain <fctr>, Cylinders <fctr>, EngineSize <dbl>,
#   Horsepower <int>, RPM <int>, Rev.per.mile <int>,
#   Man.trans.avail <fctr>, Fuel.tank.capacity <dbl>, Passengers <int>,
#   Length <int>, Wheelbase <int>, Width <int>, Turn.circle <int>,
#   Rear.seat.room <dbl>, Luggage.room <int>, Weight <int>, Origin <fctr>,
#   Make <fctr>
```

Pipes (1)

- we have shown by example that **dplyr** provides a simple syntax
- With the operator **%>%**, the syntax becomes easily readable
- Makes it possible to provide commands like in a *pipe* together
- Output of the previous is the first input to the command following

```
Cars93 %>% group_by(Type) %>% slice(1:2)
```

```
Source: local data frame [12 x 27]
Groups: Type [6]

   Manufacturer      Model    Type Min.Price Price Max.Price MPG.city
         <fctr>     <fctr>  <fctr>     <dbl> <dbl>     <dbl>    <int>
1       Pontiac    Sunbird Compact       9.4  11.1      12.8       23
2          Ford      Tempo Compact      10.4  11.3      12.2       22
3      Chrylser   Concorde   Large      18.4  18.4      18.4       20
4     Chevrolet    Caprice   Large      18.0  18.8      19.6       17
5       Hyundai     Sonata Midsize      12.4  13.9      15.3       20
6       Mercury     Cougar Midsize      14.9  14.9      14.9       19
7          Ford    Festiva   Small       6.9   7.4       7.9       31
8       Hyundai      Excel   Small       6.8   8.0       9.2       29
9       Hyundai     Scoupe  Sporty       9.1  10.0      11.0       26
10          Geo      Storm  Sporty      11.5  12.5      13.5       30
11    Chevrolet Lumina_APV     Van      14.7  16.3      18.0       18
12    Chevrolet      Astro     Van      14.7  16.6      18.6       15
# ... with 20 more variables: MPG.highway <int>, AirBags <fctr>,
#   DriveTrain <fctr>, Cylinders <fctr>, EngineSize <dbl>,
#   Horsepower <int>, RPM <int>, Rev.per.mile <int>,
#   Man.trans.avail <fctr>, Fuel.tank.capacity <dbl>, Passengers <int>,
#   Length <int>, Wheelbase <int>, Width <int>, Turn.circle <int>,
#   Rear.seat.room <dbl>, Luggage.room <int>, Weight <int>, Origin <fctr>,
#   Make <fctr>
```

Pipes (2)

- Command strings can be any length
- Are performed from left to right (in the direction of *arrow*!)
- **Example:**
  - Compute new variable *EngineSize* as the square of *EngineSize*
  - Compute for each group the minimum of the new variable
  - Sort the results in descending order accordingly to it

```
Cars93 %>% mutate(ES2 = EngineSize^2) %>% group_by(Type) %>%
 summarize(min.ES2 = min(ES2)) %>% arrange(desc(min.ES2))
```

```
# A tibble: 6 × 2
     Type min.ES2
   <fctr>   <dbl>
1   Large   10.89
2     Van    5.76
3 Compact    4.00
4 Midsize    4.00
5  Sporty    1.69
6   Small    1.00
```

Window Functions (1)

- We had: summarize() works for functions that return a single value
- What if we want to make more complex aggregations? -> *window functions*
- Different Types of *window functions*
  - Ranking / ordering: row_number(), min_rank(), percent_rank(), …
  - offsets: lag(), lead()
  - cumulative Functions: cumsum(), cummin(), cummax(), cummean(), …

Window Functions (2)

- Simple example: calculate cumulative sum and average value within each group

```
Cars93 %>% group_by(Type) %>% arrange(Type) %>% select(Manufacturer:Price) %>% mutate(cmean = cummean(Price),
csum = cumsum(Price))
```

```
Source: local data frame [93 x 7]
Groups: Type [6]

   Manufacturer    Model     Type Min.Price Price    cmean  csum
         <fctr>   <fctr>   <fctr>     <dbl> <dbl>    <dbl> <dbl>
1       Pontiac  Sunbird Compact       9.4  11.1 11.10000  11.1
2          Ford    Tempo Compact      10.4  11.3 11.20000  22.4
3     Chevrolet  Corsica Compact      11.4  11.4 11.26667  33.8
4         Dodge   Spirit Compact      11.9  13.3 11.77500  47.1
5     Chevrolet Cavalier Compact       8.5  13.4 12.10000  60.5
6    Oldsmobile  Achieva Compact      13.0  13.5 12.33333  74.0
7        Nissan   Altima Compact      13.0  15.7 12.81429  89.7
8      Chrysler  LeBaron Compact      14.5  15.8 13.18750 105.5
9         Mazda      626 Compact      14.3  16.5 13.55556 122.0
10        Honda   Accord Compact      13.8  17.5 13.95000 139.5
# ... with 83 more rows
```

Additional Notes

- **dplyr** provides opportunities to apply the same verbs in different input
  - in *remote databases* (see **vignette**)
  - objects of the class *data.table*
- abstraction: same functions, regardless of whether an input *data.frame*, *data.table* or a *data base.*
- functions in *base-R* are mostly *vector based*
- functions in **dplyr** are *data.frame*-oriented

Tasks / Exercises

Time for practical training! :)

Please continue to work on Exercises 2x).

dplyr - Summary (1)

- Package **dplyr** offers few verbs for common tasks
  - Selection of rows or columns: **filter()** or **select()**
  - Order: **arange()**
  - Uniqueness: **distinct()**
  - Recode/re-encoding: **mutate()**, **or transmute()**
  - Rename variables: **select()**, or **rename()**
  - Group: **group_by()**
  - Aggregate: **summarize()**
- A new *pipe* operator: **%>%**
- Auxiliary functions, including: **starts_with()**, **n()**

Basic Statistics in R

Alexander Kowarik, Bernhard Meindl

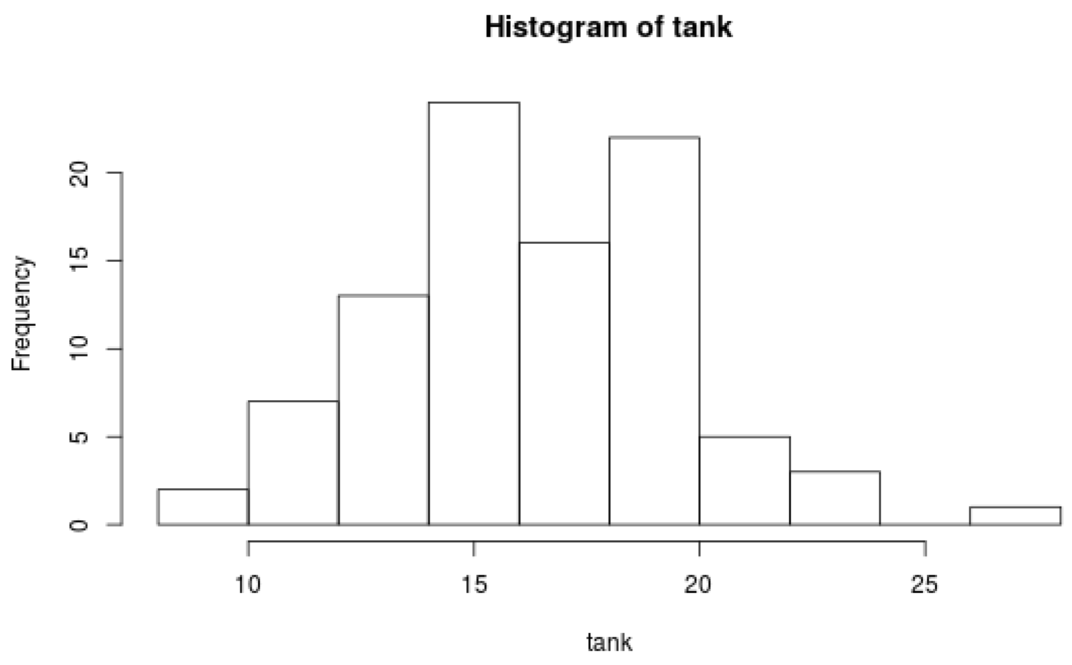
Overview / Objectives

- Univariate plots
  - Histogram
  - QQ-Plot
- Testing normality
- Comparing data groups
  - graphically
  - testing group means
- Regression analysis
  - OLS
  - robust regression
  - Nonlinear relationships

Univariate plots (1)

- Load the **Cars93** data and look at **Fuel.tank.capacity**:

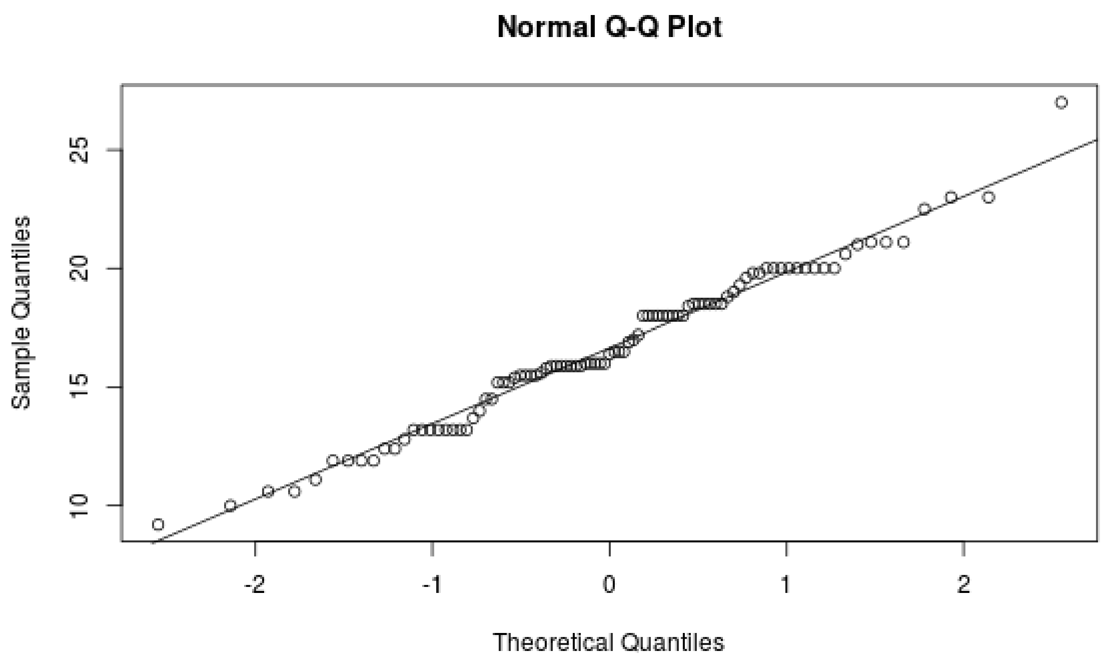
```r
data(Cars93,package="MASS")
tank <- Cars93$Fuel.tank.capacity
hist(tank)
```



Univariate plots (2)

- Normally distributed? Look at QQ-plot:

```r
qqnorm(tank)
qqline(tank)
```



Eurostat

Testing normality

- Kolmogorov-Smirnov test:

```
ks.test(tank,"pnorm")
```

```

	One-sample Kolmogorov-Smirnov test

data:  tank
D = 1, p-value < 2.2e-16
alternative hypothesis: two-sided
```

```
# Warning: ties should not be present for the Kolmogorov-Smirnov test
```

- Shapiro-Wilk normality test:

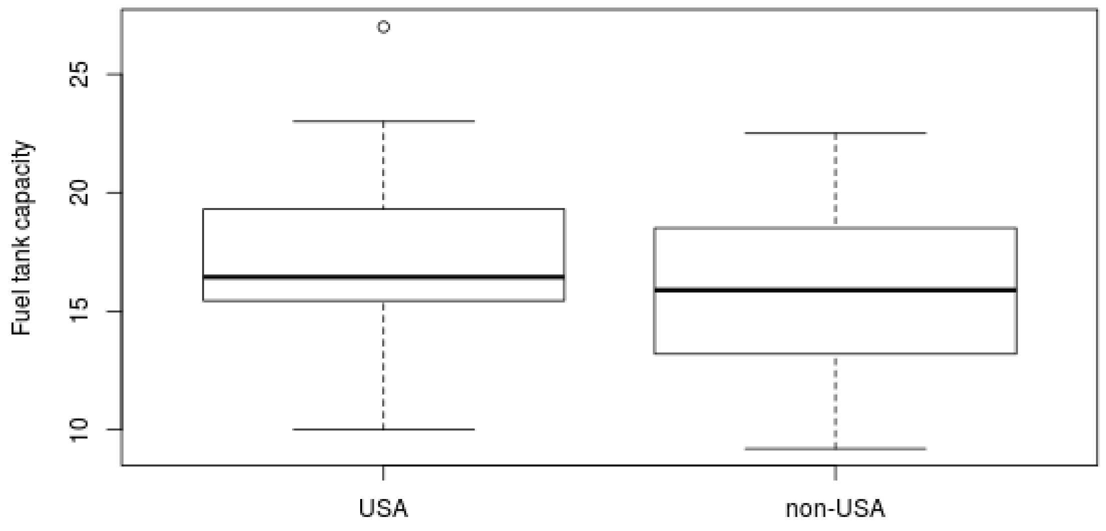
```
shapiro.test(tank)
```

```

	Shapiro-Wilk normality test

data:  tank
W = 0.98341, p-value = 0.287
```

Comparing two data groups (1)

- Use **Origin** (USA, non-USA) for grouping **tank**.

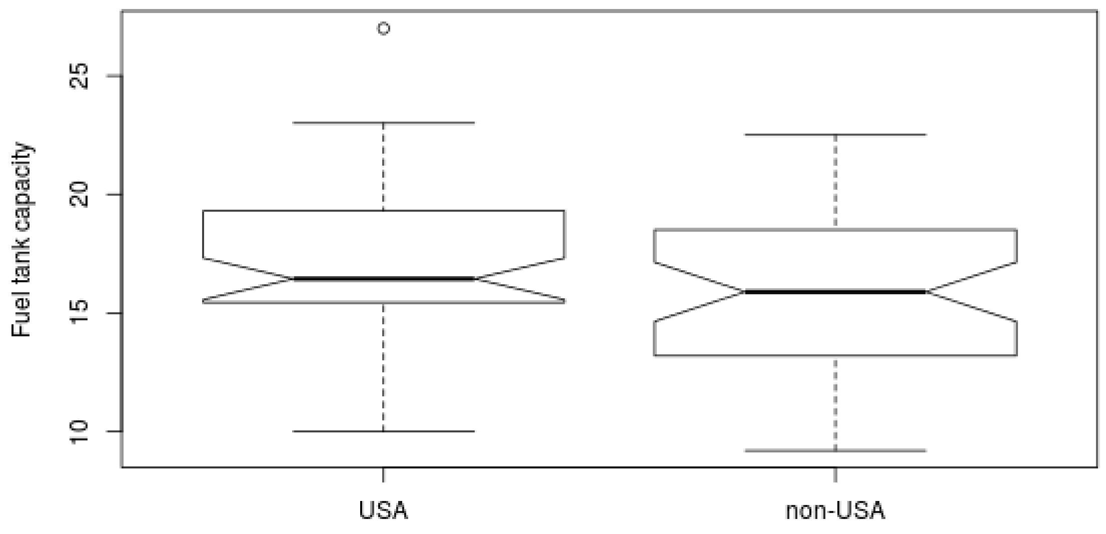
```
Origin <- Cars93$Origin
boxplot(tank~Origin,ylab="Fuel tank capacity")
```



Comparing two data groups (2)

- Confidence intervals for the medians are shown by notches in the boxplot.

```
boxplot(tank~Origin,ylab="Fuel tank capacity",notch=TRUE)
```



Overlapping notches indicate a non-significant difference.

Testing two group means

- Welch two-sample t-test:

```
t.test(tank~Origin)
```

```

    Welch Two Sample t-test

data:  tank by Origin
t = 1.1769, df = 89.344, p-value = 0.2424
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -0.5512865  2.1532309
sample estimates:
    mean in group USA mean in group non-USA
             17.05208              16.25111
```

- Wilcoxon test (non-parametric):

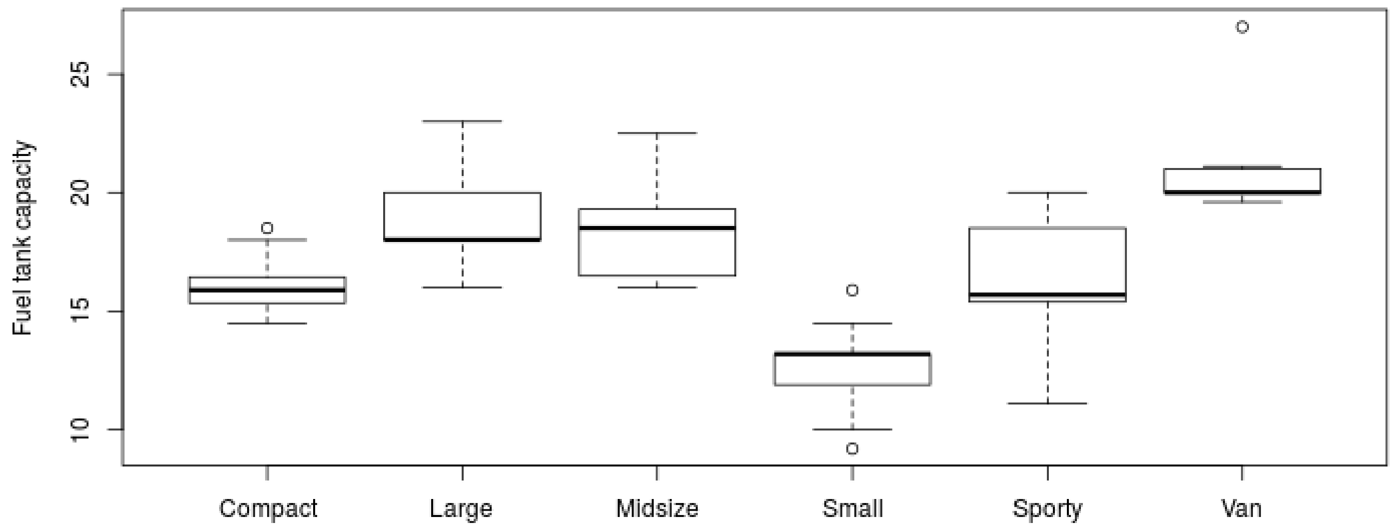
```
wilcox.test(tank~Origin)
```

```

    Wilcoxon rank sum test with continuity correction

data:  tank by Origin
W = 1187, p-value = 0.4121
alternative hypothesis: true location shift is not equal to 0
```

Comparing several data groups (1)

- Use **Type** for grouping **tank**.

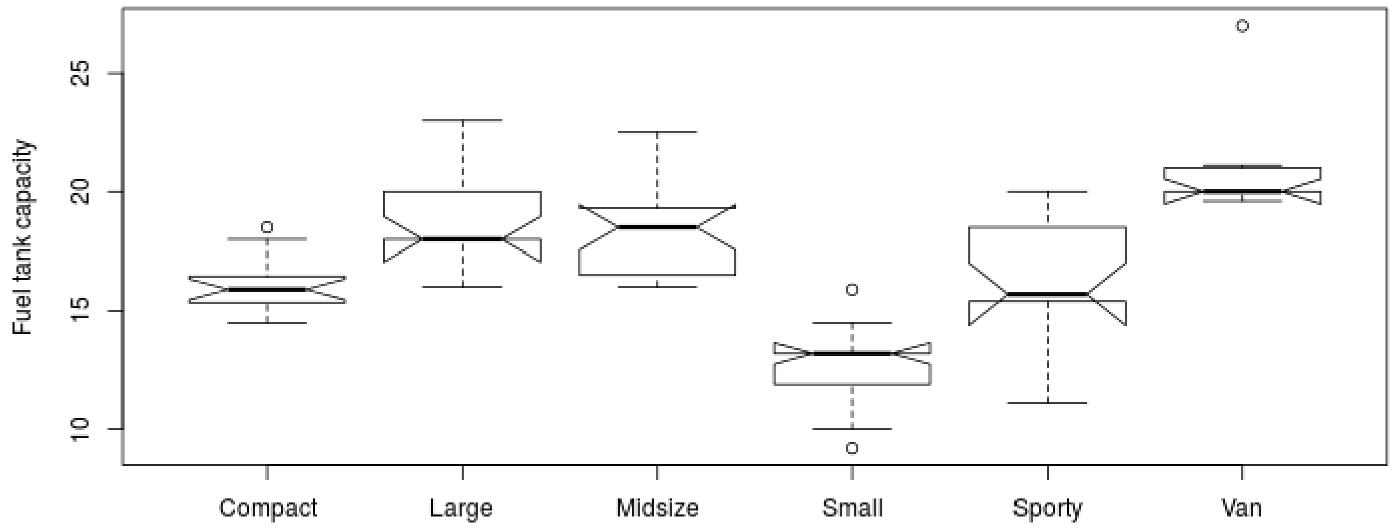
```
Type <- Cars93$Type
boxplot(tank~Type,ylab="Fuel tank capacity")
```



Comparing several data groups (2)

- Notched boxplots look a bit scaring.

```
boxplot(tank~Type,ylab="Fuel tank capacity",notch=TRUE)
```



Testing several group means

- ANOVA - analysis of variance:

```
summary(aov(tank~Type))
```

```
            Df Sum Sq Mean Sq F value Pr(>F)    
Type         5  656.3  131.25   34.28 <2e-16 ***
Residuals   87  333.1    3.83                   
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Kruskal-Wallis test (non-parametric):

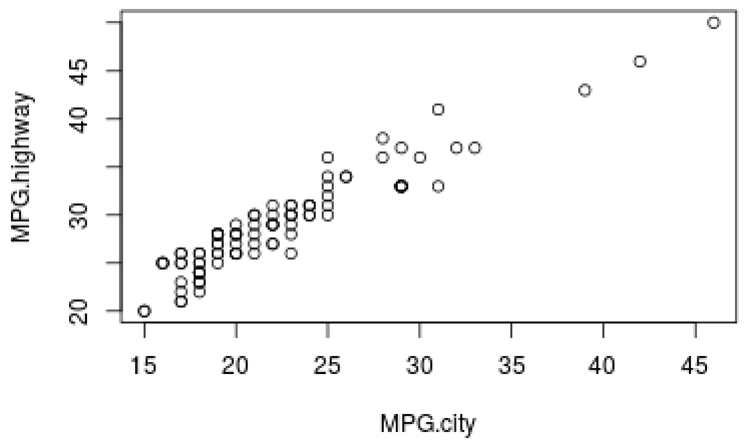
```
kruskal.test(tank~Type)
```

```

    Kruskal-Wallis rank sum test

data:  tank by Type
Kruskal-Wallis chi-squared = 64.364, df = 5, p-value = 1.518e-12
```

Bivariate analysis

- Plot data:

```
plot(Cars93[,c("MPG.city","MPG.highway")])
```



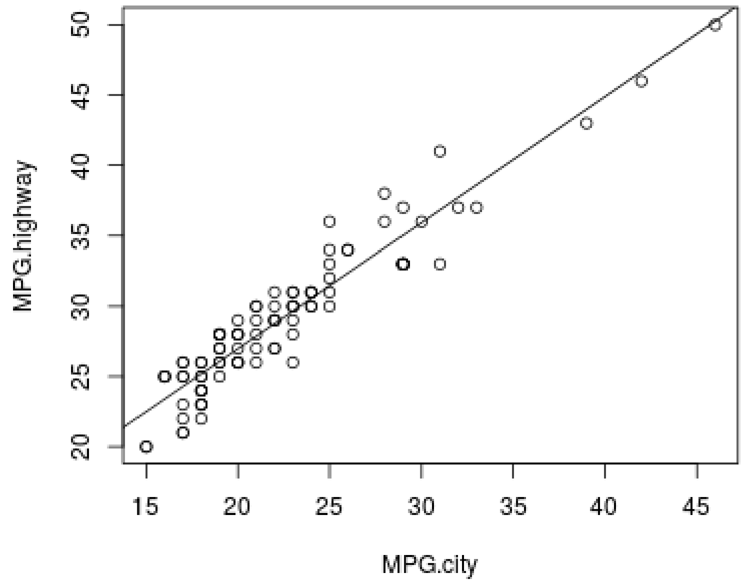
```
cor(Cars93[,"MPG.city"],Cars93[,"MPG.highway"])
```

```
[1] 0.9439358
```

Regression analysis (1)

- Plot data, add LS-regression line:

```
plot(Cars93[,c("MPG.city","MPG.highway")])
res <- lm(MPG.highway~MPG.city,data=Cars93)
abline(res)
```



Regression analysis (2)

- LS-regression inference statistics:

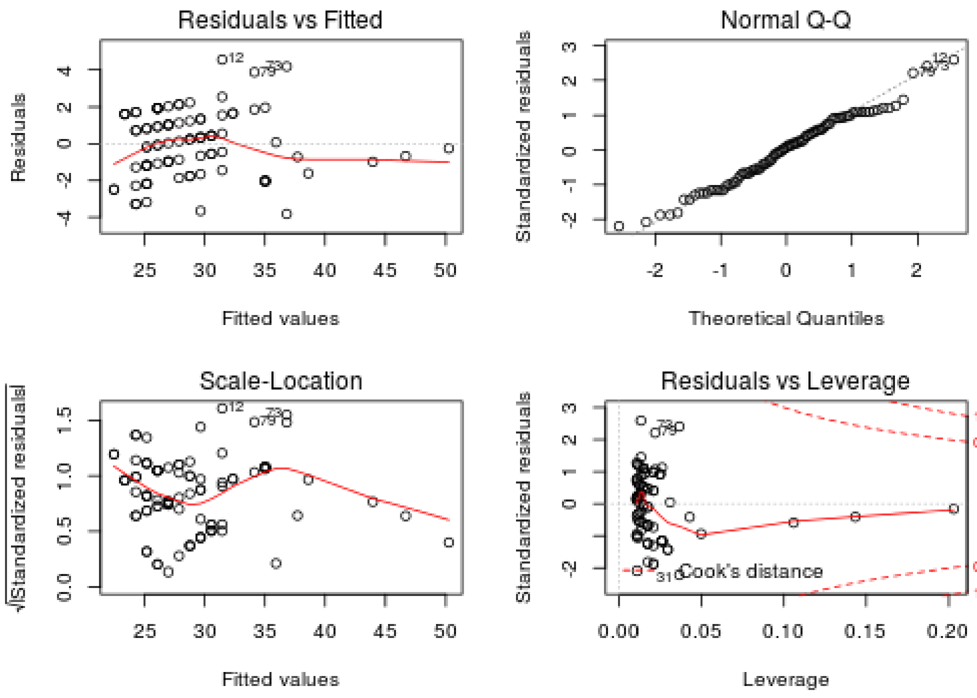
```
summary(res)
```

```

Call:
lm(formula = MPG.highway ~ MPG.city, data = Cars93)

Residuals:
    Min      1Q  Median      3Q     Max 
-3.8185 -1.1764  0.1369  1.3458  4.5547 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept)  9.05658    0.75691   11.96   <2e-16 ***
MPG.city     0.89555    0.03283   27.28   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.77 on 91 degrees of freedom
Multiple R-squared:  0.891, Adjusted R-squared:  0.8898 
F-statistic:   744 on 1 and 91 DF,  p-value: < 2.2e-16
```

Regression analysis (3)

- LS-regression diagnostic plots:

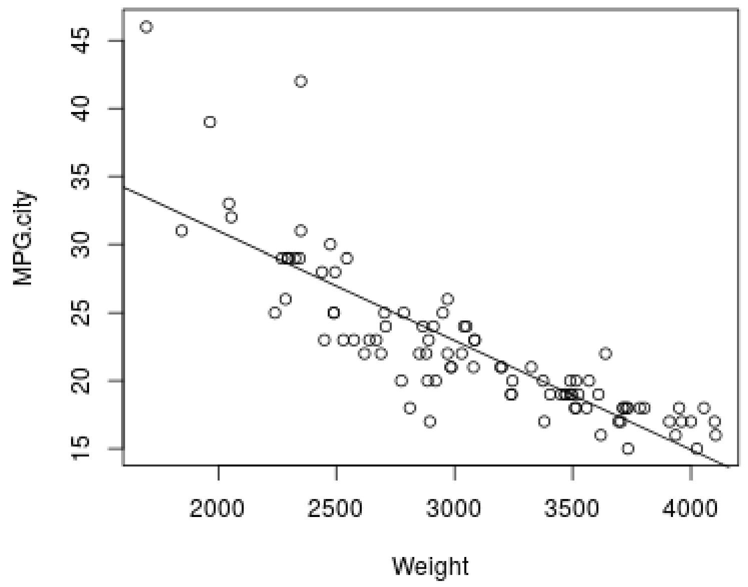
```r
par(mfrow=c(2,2),mar=c(4,4,3,2))
plot(res)
```



Robust regression (1)

- Is LS-regression appropriate?

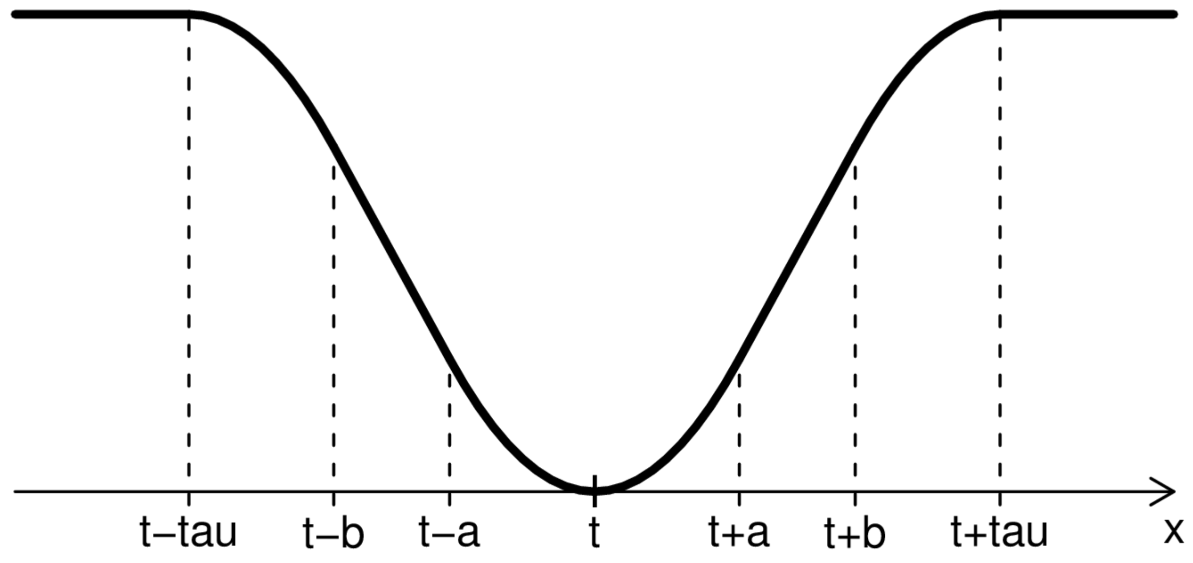
```
plot(MPG.city~Weight,data=Cars93)
res <- lm(MPG.city~Weight,data=Cars93); abline(res)
```



- Three outliers with small **Weight** and high **MPG.city** may spoil the regression line.

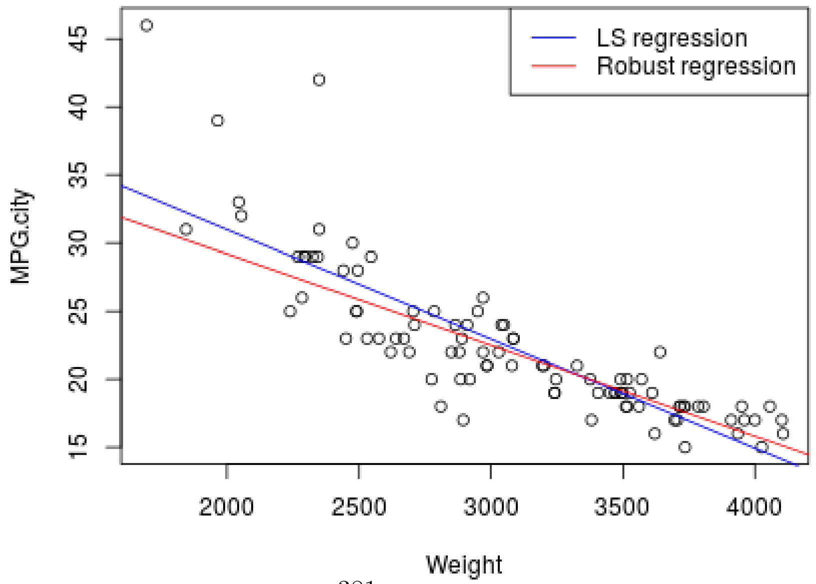
Robust regression (2)

- LS (least-squares) minimizes *sum of squared residuals.*
- Robust regression: minimize sum of a function of the residuals.



Robust regression (3)

```
plot(MPG.city~Weight,data=Cars93)
abline(lm(MPG.city~Weight,data=Cars93),col="blue")

library(robustbase) # tools from robust statistics
abline(lmrob(MPG.city~Weight,data=Cars93),col="red")

legend("topright",legend=c("LS regression","Robust regression"),col=c("blue","red"),lty=c(1,1))
```



Multiple linear regression (1)

- Response variable **Price** versus most of the other variables

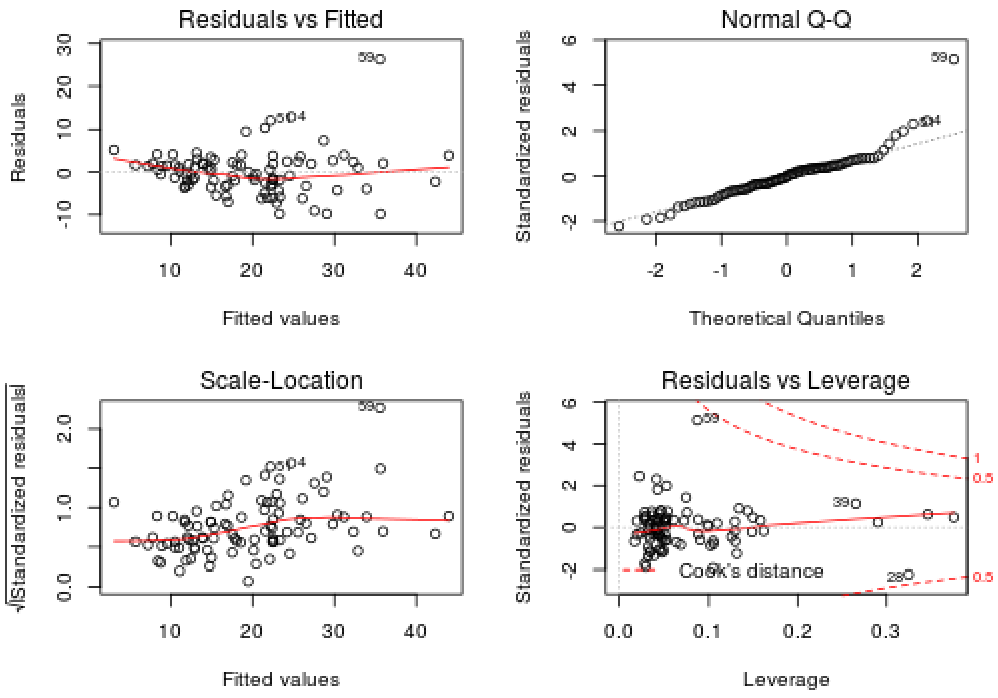
```
res <- lm(Price~MPG.city+EngineSize+Horsepower+Weight+Wheelbase+
         Width,data=Cars93)
summary(res)
```

```

Call:
lm(formula = Price ~ MPG.city + EngineSize + Horsepower + Weight + 
    Wheelbase + Width, data = Cars93)

Residuals:
    Min      1Q  Median      3Q     Max 
-9.8193 -3.0679  0.0285  2.0600 26.3307 

Coefficients:
             Estimate Std. Error t value Pr(>|t|)    
(Intercept) 49.701693  23.449123   2.120  0.03693 *  
MPG.city    -0.158120   0.191412  -0.826  0.41105    
EngineSize   1.637861   1.236217   1.325  0.18871    
Horsepower   0.140015   0.019156   7.309 1.29e-10 ***
Weight       0.001253   0.003694   0.339  0.73538    
Wheelbase    0.535795   0.199345   2.688  0.00863 ** 
Width       -1.595736   0.361519  -4.414 2.93e-05 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 5.357 on 86 degrees of freedom
Multiple R-squared:  0.7125,	Adjusted R-squared:  0.6924 
F-statistic: 35.51 on 6 and 86 DF,  p-value: < 2.2e-16
```

Multiple linear regression (2)

- LS-regression diagnostic plots:

```
par(mfrow=c(2,2),mar=c(4,4,3,2))
plot(res)
```



Multiple linear robust regression (1)

- Response variable Price versus most of the other variables

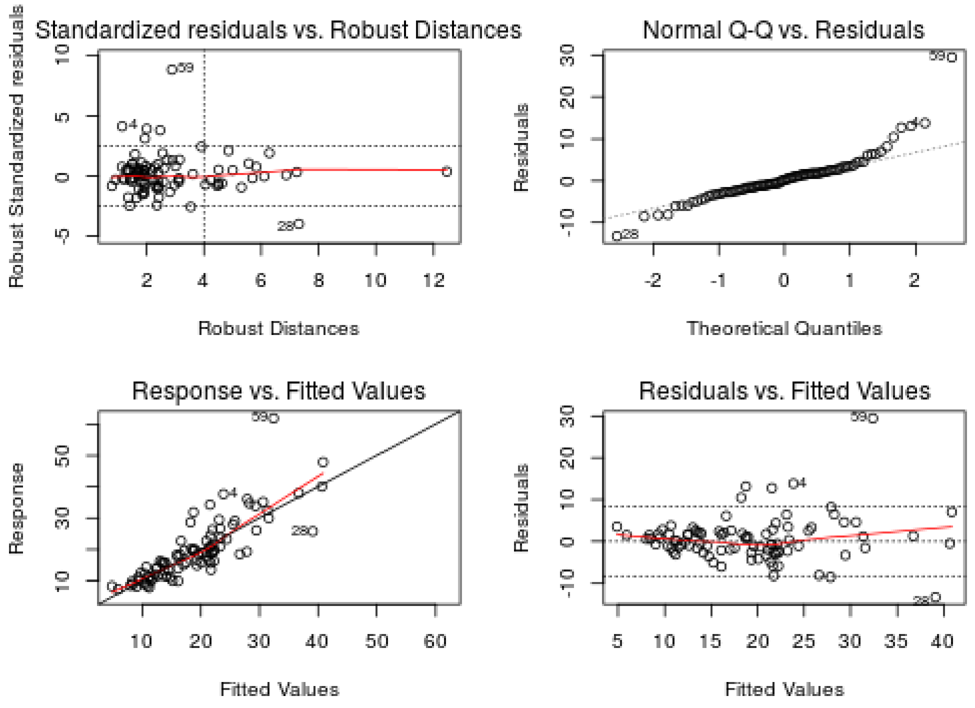
```
res1 <- lmrob(Price~MPG.city+EngineSize+Horsepower+Weight+Wheelbase+
          Width,data=Cars93)
names(summary(res1))
```

```
 [1] "call"          "terms"         "residuals"     "scale"
 [5] "rweights"      "converged"     "iter"          "control"
 [9] "df"            "coefficients"  "r.squared"     "adj.r.squared"
[13] "cov"           "aliased"       "sigma"
```

Multiple linear robust regression (2)

- Robust regression diagnostic plots:

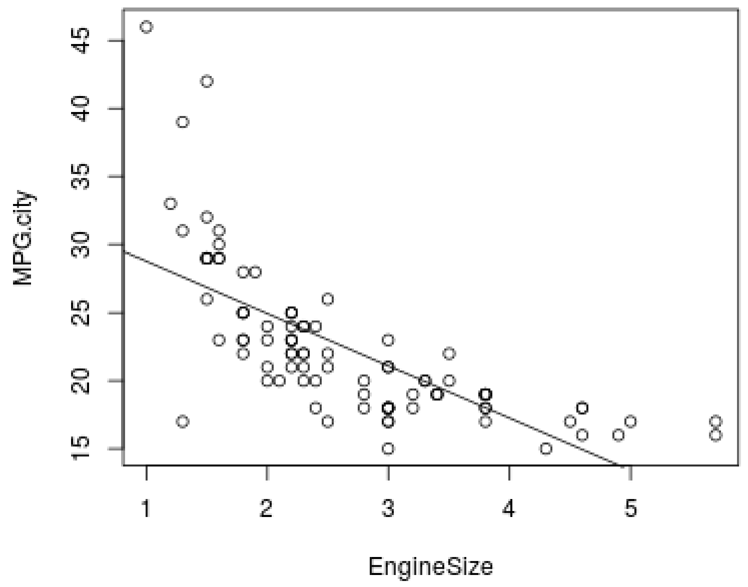
```
par(mfrow=c(2,2),mar=c(4,4,3,2))
plot(res1,which=c(1,2,3,4))
```



Nonlinear relationships (1)

- Linear or nonlinear trend?

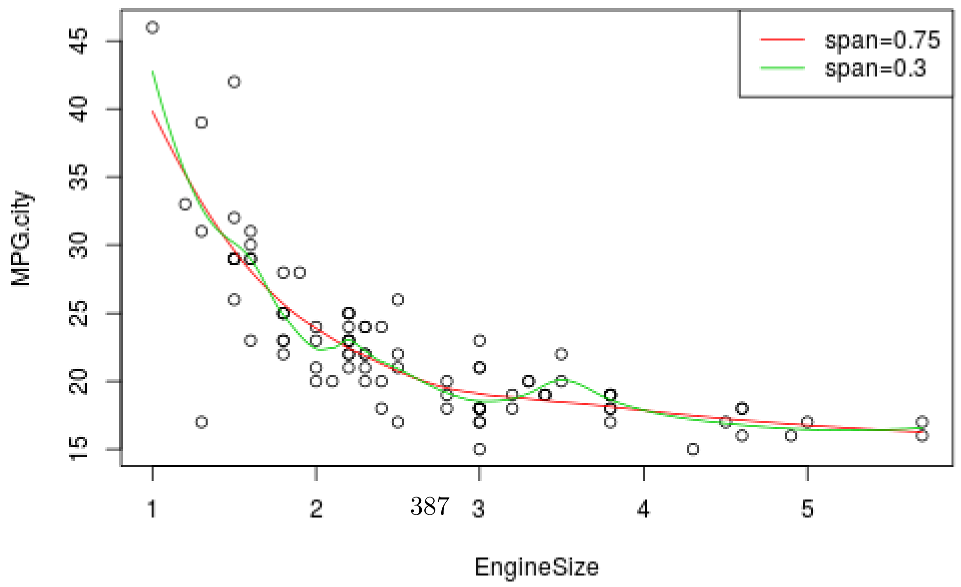
```
plot(MPG.city~EngineSize,data=Cars93)
abline(lm(MPG.city~EngineSize,data=Cars93))
```



Nonlinear relationships (2)

- Use loess() to fit the nonlinear trend:

```
plot(MPG.city~EngineSize,data=Cars93)
xrange <- seq(0,6,by=0.01)
res1 <- loess(MPG.city~EngineSize,data=Cars93,span=0.75) # default span
res2 <- loess(MPG.city~EngineSize,data=Cars93,span=0.3) # smaller span
lines(xrange,predict(res1,xrange),col=2)
lines(xrange,predict(res2,xrange),col=3)
legend("topright",legend=c("span=0.75","span=0.3"),col=c(2,3),lty=c(1,1))
```



Eurostat

Tasks / Exercises

Time for practical training! :)

Please continue to work on Exercises 1x) to 4x).

Summary (1)

- Viewing distributions using **hist()** or **qqplot()**
- Testing for normality using **ks.test()** or **shapiro.test()**
- Graphically displaying groups using **boxplot()**
- Testing group means using **t.test()**, **wilcox.test()**, **aov()** or **kruskal.test()**
- Regression analysis using **lm()** or robust regression using **lmrob()** from package **robustbase**

Some Remarks on Classes and Object-Orientation

Alexander Kowarik, Bernhard Meindl

Aim

- Replicate some basic concepts about classes in **R**
- Howto make use of object-orientation for user-friendly implementation of functions
- Brief example about on S4 (just an impression)

Class-Systems in R

- R has 2 different class systems:
    - S3 (Simple, is covered here)
    - S4 (*clean* but more complex)
- Assigning each object to a single class (attribute *class*)
- Classes allow object-oriented programming and **function overloading**
- Users can very easily define custom-classes

```
class(y) <- "newclass" ## more later
```

- Generic functions: different output for objects of different classes

Method dispatch (1)

- Summary of an object of class integer

```
x <- rep(0:1, c(5,10)); x
```

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

```
class(x)
```

```
[1] "integer"
```

```
summary(x)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
 0.0000  0.0000  1.0000  0.6667  1.0000  1.0000 
```

Method dispatch (2)

- Summary of an object of class factor

```
y <- as.factor(x); class(y)
```

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

```
summary(y)
```

```
 0  1 
 5 10 
```

Overloading and method-dispatch

S3 classes give the possibility to do function overloading

- Implementation: define a generic function

```
foo <- function (x, ...) UseMethod ("foo")
```

- Is object *myx* from class “bar”, then R looks after calling foo(myx) the following functions (in this order):
    1. foo.bar()
    2. foo.default()
- this is called method dispatching.

Example: print() for objects of class ’foo’

- **Definition:**

```
print.foo <- function(x, ...) {
  cat(paste0("This is an object of class 'foo' (Length=", length(x), "):\n"))
  print(unclass(x))
}
```

- **Test:**

```
x <- 1:10; x
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

```
class(x) <- "foo"; print(x)
```

```
This is an object of class 'foo' (Length=10):
 [1]  1  2  3  4  5  6  7  8  9 10
```

Methods, some remarks

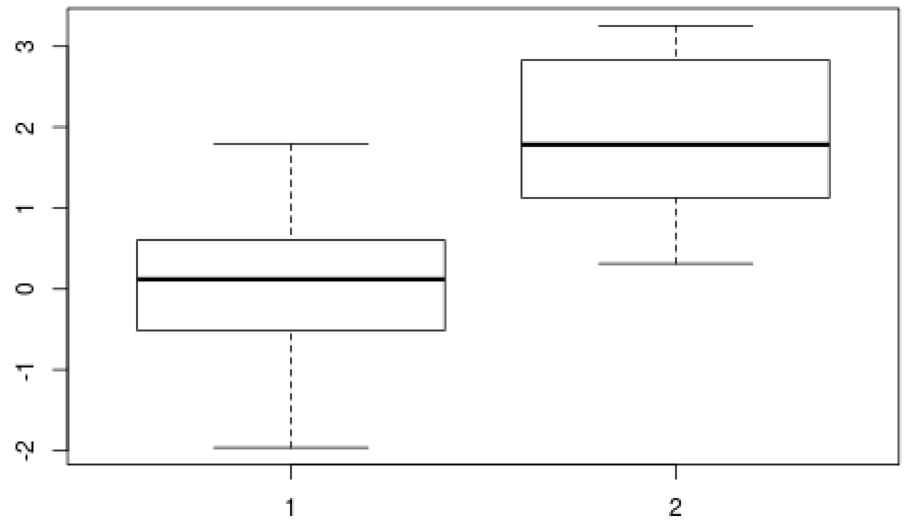
- A method shall have all the arguments of the generic.
- The order of the arguments must be the same as in the generic function.
- same defaults as the generic
- see for example the generic function **?print**
  - a print method for a certain class must have these arguments:

```
args(print)
```

```
function (x, ...)
NULL
```

Example - user-friendly implementation

```r
set.seed(123)
x <- rnorm(20)
y1 <- rnorm(15, 2)
boxplot(x, y1)
```



Example - user-friendly implementation

- We want to test \(H_0: \mu_1 = \mu_2 \) against \(H_1: \mu_1 \neq \mu_2 \).
- We use the two-sample t-test.
- Step-by-step, we create better solutions of the implementation
- Note: there is an own class on tests (htest) that we do not touch in the beginning.

Example - not user-friendly:

How 95 percent of R users will code (not very nice):

```
ttest1 <- function(x, y, mu=0) {
  nx <- length(x);  ny <- length(y)
  df <- (nx+ny-2) ## degrees of freedom
  ## pooled variance
  s2 <- ((nx-1)*var(x)+(ny-1)*var(y))* (nx+ny)/(nx*ny*df)
  tstat <- ((mean(y)-mean(x))-mu)/sqrt(s2) ## test statistics
  pval <- 2*pt(-abs(tstat), df) ## p-value
  list(tstat=tstat , pval=pval) ## return result
}
```

Now look at the output

```
ttest1(x, y1)
```

```
$tstat
[1] 5.400884

$pval
[1] 5.650172e-06
```

Can we do better?

Example - Better print-output

```r
ttest2 <- function(x, y, mu=0){
  z <- ttest1(x, y, mu=mu)
  cat("The t statistic is:", z$tstat , "\n")
  cat("The p value is: ", z$pval , "\n")
}
```

Now look at the output

```r
ttest2(x, y1)
```

```
The t statistic is: 5.400884
The p value is:  5.650172e-06
```

- **ttest1** is the *workhorse' …
- print-output is nicer

But can we do it better?

Example - Return results:

```r
ttest3 <- function(x, y, mu=0){
  z <- ttest1(x, y, mu=mu)
  cat("The t statistic is:",z$tstat)
  cat(" ( p-value:",z$pval ,")\n")
  return(z)
}
```

Now look at the output

```r
ttest2(x, y1)
```

```
The t statistic is: 5.400884
The p value is:  5.650172e-06
```

- Results are returned, but ...

```r
res <- double(4)
for (k in 1:4) res[k] <- ttest3(x, rnorm(20))$pval
```

```
The t statistic is: -0.1914919 ( p-value: 0.8491598 )
The t statistic is: -0.6118234 ( p-value: 0.5442981 )
The t statistic is: 0.4490157 ( p-value: 0.6559708 )
The t statistic is: -0.9877459 ( p-value: 0.329525 )
```

We can do better!

Example - Use a custom class

```r
ttest4 <- function(x, y, mu=0) {
  z <- ttest1(x, y, mu=mu)
  z$name <- "Two-sample t-test"
  class(z) <- "ttest"
  z
}
print.ttest = function(x, ...) {
  cat(x$name , "\n")
  cat(" The t statistic is:", x$tstat , "\n")
  cat(" The p value is: ", x$pval , "\n")
}
```

- nice print-output and results can be accessed

```r
res <- ttest4(x, y1); print(res)
```

```
Two-sample t-test
 The t statistic is: 5.400884
 The p value is:  5.650172e-06
```

```r
res$pval
```

```
[1] 5.650172e-06
```

- function overloading –> user-friendly

Example - enhance with methods for plot and summary

```
plot.ttest <- function(x, y, ...) {
  plot(1, xlab="create a nice plot")
}
summary.ttest <- function(object , ...) {
  cat("make a useful summary")
}
erg4 <- ttest4(x, y1)
```

again method dispatch: **summary()** looks first (internally) on

```
class(erg4)
```

```
[1] "ttest"
```

```
methods(class="ttest")
```

```
[1] plot    print   summary
see '?methods' for accessing help and source code
```

and if a method is found, it will be applied

```
summary(erg4)
```

```
make a useful summary
```

Example - and a formula...

Formula

```
ttest.formula <- function(formula, data=list(), mu=0){
  mf <- model.frame(formula, data=data)
  x <- split(mf[,1], mf[,2])
  if(length(x)==2)
    return(ttest(x[[1]],x[[2]],mu=mu))
  else
    stop("Grouping variable must have two levels!")
}
```

Generic function

```
ttest <- function(x, y, mu=0, ...)
  UseMethod("ttest")
```

Default method

```
ttest.default <- function(x, y, mu=0) {
  z <- ttest1(x, y, mu=mu)
  z$name <- "Two-sample t-test"
  class(z) <- "ttest"
  z
}
```

Example - and a formula...

- create testdata

```
xnew <- data.frame(
  x=c(x,y1),
  group=factor(c(rep(1, length(x)),
  rep(2, length(y1))))
)
summary(xnew)
```

```
       x              group
 Min.   :-1.9666   1:20
 1st Qu.: 0.0906   2:15
 Median : 0.8619
 Mean   : 0.8947
 3rd Qu.: 1.7485
 Max.   : 3.2538
```

- Formula based call:

```
ttest(xnew$x ~ xnew$group)
```

```
Two-sample t-test
 The t statistic is: 5.400884
 The p value is:  5.650172e-06
```

Example - S4 classes and methods (1)

- S4-classes are more formal
- allow for better control due to automatic checks
- we need class-definitions, generics and correspondig methods
- Let's define a S4-class

```r
setClass("S4class_htest",
  representation(tstat="numeric",pval="numeric",name="character"),
  prototype(tstat=numeric(1),pval=numeric(1),name=character(1))
)
```

Example - S4 classes and methods (2)

- Lets define a generic method

```
setGeneric("ttestS4", function(x, y, mu) {
  standardGeneric("ttestS4")
})
```

```
[1] "ttestS4"
```

```
setMethod("ttestS4",
  signature(x="numeric", y="numeric"),
  function(x, y, mu=0){
    z <- ttest1(x, y)
    result <- new("S4class_htest")
    result@name <- "Two-sample t-test"
    result@tstat <- z$tstat
    result@pval <- z$pval
    result
  }
)
```

```
[1] "ttestS4"
```

Example - S4 classes and methods (3)

```r
resS4 <- ttestS4(x=x,y=y1)
resS4
```

```
An object of class "S4class_htest"
Slot "tstat":
[1] 5.400884

Slot "pval":
[1] 5.650172e-06

Slot "name":
[1] "Two-sample t-test"
```

Example - S4 classes and methods (4)

implementing a print method

```
setMethod("show", "S4class_htest",
  function(object){
    cat(object@name , "\n")
    cat(" The t statistic is:", object@tstat , "\n")
    cat(" The p value is: ", object@pval , "\n")
  }
)
```

```
[1] "show"
```

```
resS4
```

```
Two-sample t-test
 The t statistic is: 5.400884
 The p value is:  5.650172e-06
```

Access of elements via slot (@) instead of $

```
slotNames(resS4); # resS4@pval
```

```
[1] "tstat" "pval"  "name"
```

Tasks / Exercises

Time for practical training! :)

Please continue to work on Exercises 1x).

Summary (1)

- object-oriented programming style –> userfriendly code
- previous example worked as demo –> use class **htest** for this problem
- S4 is more complicated but users and programmers benefit from automatic generated error messages…

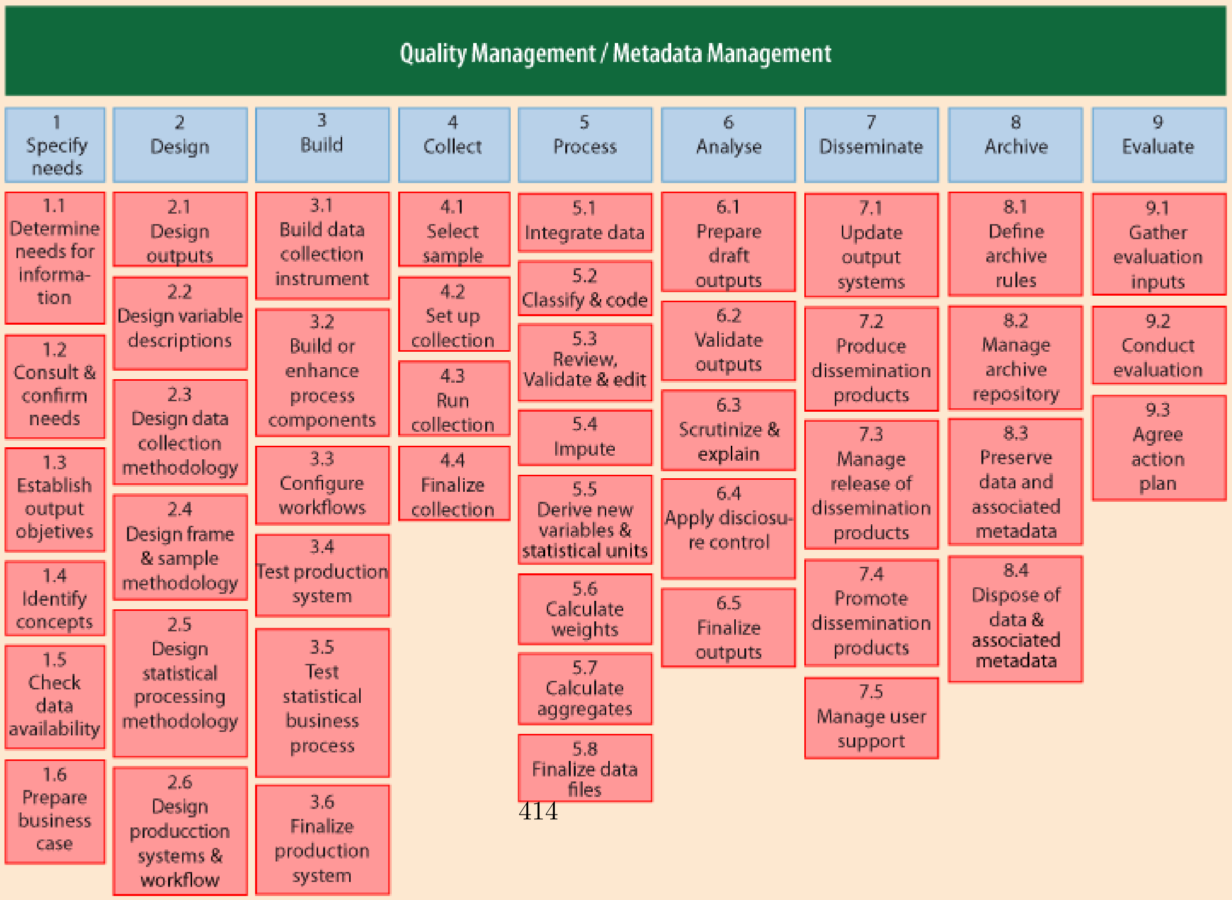
R in Official Statistics and Survey Methodology

Alexander Kowarik, Bernhard Meindl

GSBPM

- GSBPM Model

Figure 3


The Generic Statistical Business Process Model (GSBPM)
Quality Management / Metadata Management
1 Specify needs
1.1 Determine needs for informa-tion
1.2 Consult & confirm needs
1.3 Establish output objetives
1.4 Identify concepts
1.5 Check data availability
1.6 Prepare business case
2 Design
2.1 Design outputs
2.2 Design variable descriptions
2.3 Design data collection methodology
2.4 Design frame & sample methodology
2.5 Design statistical processing methodology
2.6 Design producction systems & workflow
3 Build
3.1 Build data collection instrument
3.2 Build or enhance process components
3.3 Configure workflows
3.4 Test production system
3.5 Test statistical business process
3.6 Finalize production system
4 Collect
4.1 Select sample
4.2 Set up collection
4.3 Run collection
4.4 Finalize collection
5 Process
5.1 Integrate data
5.2 Classify & code
5.3 Review, Validate & edit
5.4 Impute
5.5 Derive new variables & statistical units
5.6 Calculate weights
5.7 Calculate aggregates
5.8 Finalize data files
6 Analyse
6.1 Prepare draft outputs
6.2 Validate outputs
6.3 Scrutinize & explain
6.4 Apply disclosu-re control
6.5 Finalize outputs
7 Disseminate
7.1 Update output systems
7.2 Produce dissemination products
7.3 Manage release of dissemination products
7.4 Promote dissemination products
7.5 Manage user support
8 Archive
8.1 Define archive rules
8.2 Manage archive repository
8.3 Preserve data and associated metadata
8.4 Dispose of data & associated metadata
9 Evaluate
9.1 Gather evaluation inputs
9.2 Conduct evaluation
9.3 Agree action plan

Available Functionality and Packages

- CRAN Task View on Official Statistics and Survey Methodology

What else is important?

- Connection to databases
- Import/Export
- Efficient data manipulation
- Mapping
- Visualisation
- Dynamic reporting

Survey Sampling (1)

- We need a sampling frame containing basic variables like contact addresses, names, ...
- Let us assume we have this in data set `eusilcP` – it serves as our sampling frame and population.

```
require(simFrame)
data(eusilcP)
head(eusilcP[,1:3]) ## usually fewer variables in a sampling frame
```

```
      hid        region hsize
39993   1 Upper Austria     2
39994   1 Upper Austria     2
31004   2        Styria     2
31005   2        Styria     2
29071   3        Styria     1
41322   4 Upper Austria     3
```

- We could do *srs* using **sample()**
- We could reduce costs by drawing the sample in a clever way

Survey Sampling (2)

- draw a sample by *region*, equal sizes within groups

```
require(sampling)
x <- draw(eusilcP[, c("hid", "id", "region")],
  design = "region", grouping = "hid",
  size = rep(650, 9))
dim(x)
```

```
[1] 14076     4
```

```
table(x$region)
```

```

   Burgenland Lower Austria        Vienna     Carinthia       Styria
         1550          1586          1322          1512         1584
Upper Austria      Salzburg         Tyrol    Vorarlberg
         1619          1580          1650          1673
```

```
summary(x$.weight)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  1.229   2.571   2.906   4.183   6.263   9.011
```

Survey Sampling (3)

- draw a sample by *region*, sample sizes are proportional to group size

```
## stratified group sampling, proportional size
tab <- table(eusilcP$region[!duplicated(eusilcP$hid)])
x2 <- draw(eusilcP[, c("hid", "id", "region")],
  design = "region", grouping = "hid",
  size = as.numeric(tab/4))
dim(x2)
```

```
[1] 14589     4
```

```
table(x2$region)
```

```

   Burgenland Lower Austria        Vienna     Carinthia       Styria
          508          2806          2837          1030          2027
Upper Austria      Salzburg         Tyrol    Vorarlberg
         2610           981          1163           627
```

```
summary(x2$.weight)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
      4       4       4       4       4       4
```

Editing (1)

Editing is one approach to correct incorrect data in an automatic manner

- package **deducorrect**
- package **editrules**

(different approach: screening and outlier detection based on non-deterministic data-driven approaches)

Editing (2)

- Package **editrules** works recordwise and checks data constraints including
  - linear (in)equality constraints for numerical data;
  - constraints on value combinations of categorical data;
  - conditional constraints on numerical and/or mixed data
- **editrules** offers
  - **error localization** functionality
  - paradigm (by Fellegi and Holt): determine the smallest (weighted) number of variables to adapt such that no (additional or derived) rules are violated.
- for sign flips, typing errors or rounding errors –> use package **deducorrect**

Editing (3)

- Defining edits can be done:
  - **editfile**: read conditional numerical, numerical and categorical constraints from textfile
  - **editset**: create conditional numerical, numerical and categorical constraints
  - **editmatrix**: create a linear constraint matrix for numerical data
  - **editarray**: create value combination constraints for categorical data

Editing (4)

- an example using **editmatrix()**

```
Error in library(editrules) : there is no package called 'editrules'
```