Using R for Data Analysis and Graphics

Andreas Papritz, Cornelia Schwierz and Martin Mächler

Institut für Terrestrische Ökosysteme
Seminar für Statistik
ETH Zürich

Autumn Semester 2013

[0] based on work by Werner Stahel and Manuel Koller
[0] slides rendered (by LaTeX) on February 19, 2014

0.1 What is R?

- R is a software environment for statistical computing.
- R is based on commands. Implements the S language.
- There is an inofficial menu-based interface to R (R-Commander).
- Drawbacks of menus: difficult to record and document what you do
- Advantage of command scripts:
  - documents an analysis and
  - allows easy repetition with new data, options, ...
- R is free software. `http://www.r-project.org`
  Supported operating systems: Linux, Mac OS X, Windows
- Lingua franca for exchanging statistical methods among researchers

0.2 Other Statistical Software

- S+ (formerly "S-PLUS") same programming language, commercial. Features a GUI.
- SPSS: good for standard procedures.
- SAS: all-rounder, good for large data sets, complicated analyses.
- Systat: Analysis of Variance, easy-to-use graphics system.
- Excel: Good for getting (a small!) dataset ready. Very limited collection of statistical methods. **Not for serious data analysis!**
- Matlab: Mathematical methods. Statistical methods limited. Similar "paradigm", less flexible structure.

0.3 Introductory Examples

- Print a data set that was read before by typing `d.sport`

```
            weit kugel hoch  disc stab speer punkte
OBRIEN      7.57 15.66  207 48.78  500 66.90   8824
BUSEMANN    8.07 13.60  204 45.04  480 66.86   8706
 :            :    :      :   :      :    :       :
 :            :    :      :   :      :    :       :
 :            :    :      :   :      :    :       :
CHMARA      7.75 14.51  210 42.60  490 54.84   8249
```

- Draw a histogram of the scores of variable the `kugel` by typing `hist(d.sport[,"kugel"])`.
- We call here the R function `hist` with the argument `d.sport[,"kugel"]`.
- The function call opens a graphics window and displays the frequency distribution of the scores for `kugel`.

- Scatter plot: type
  `plot(d.sport[,"kugel"], d.sport[,"speer"])`
  - First argument: x coordinates; second: y coordinates
  - Many(!) optional arguments:

```
plot(d.sport[,"kugel"],d.sport[,"speer"],
     xlab="shot put",ylab="javelin",pch=7)
```

- Scatter plot matrix: type
  `pairs(d.sport)`
  Every variable of `d.sport` is plotted against all other variables.
- Get a dataset from a text file on the web and assign a name to it

```
d.sport <- read.table(
"http://stat.ethz.ch/Teaching/Datasets/WBL/sport.dat")
```


0.4 Scripts and Editors

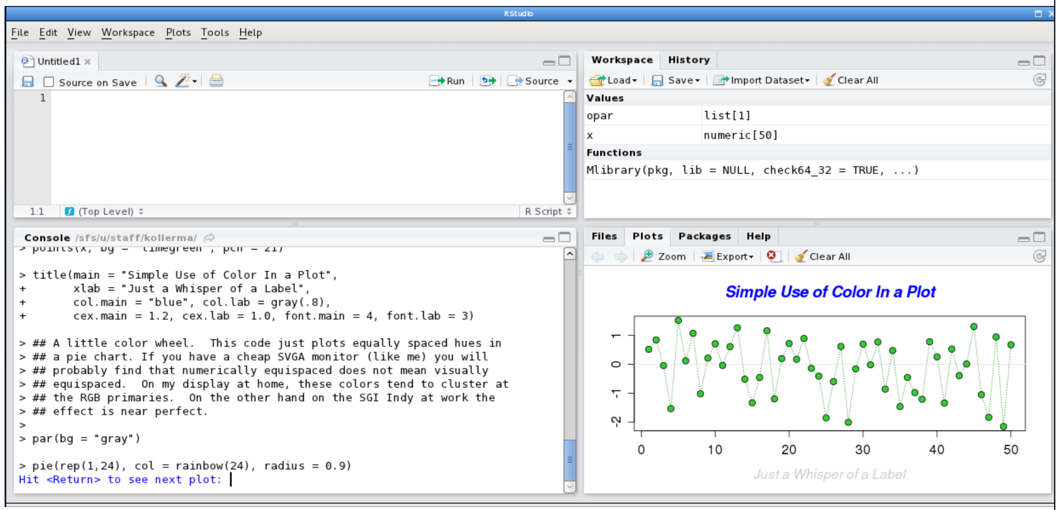
Instead of typing commands into the R console, you can generate commands by an editor and then "send" them to R ... and later modify (correct, expand) and send again.

Text editors with support for R

- R Studio (free software available for all major platforms:
  `http://rstudio.org/`
- Tinn-R (only for Windows):
  `http://www.sciviews.org/Tinn-R/`
- Emacs[1] with ESS: `http://ESS.r-project.org/`[2]
- WinEdt (only for Windows): `http://www.winedt.com/`

[1] `http://www.gnu.org/software/emacs/`
[2] For Windows and Mac, on the Downloads tab, look for the "All-in-one installation" by Vincent Goulet

The R Studio Window



The Window has 2×2 panes; the top left pane will be our "R script file" or "R file", to be saved e.g., as `ex1.R` .

R Studio — Keyboard Shortcuts

Many shortcuts by which you work more efficiently in RStudio.
Menu Help → Keyboard Shortcuts gives two pages of shortcuts.
A few of important ones are[3]:

Description	Key
Indent	Tab (at beginning of line)
Attempt completion	Tab
Cut / Paste / Copy	Ctrl + X / V / C
Insert assignment "arrow" ← (2 letter <–)	Alt + -
Run current line/selection	Ctrl + Enter
Run from document beginning to current line	Ctrl + Shift + B
Move cursor to beginning of line	Home
Move cursor to end of line	End
Save active document ("R file")	Ctrl + S
Show help	F1

[3] on Mac, you can replace `Ctrl` by `Command` (= "Apple key" = ⌘)

0.5 Using R

- In the R console, you will see the prompt ‘ >␣ ’ You can type a command in the console (or better: write it in an R Script and send it from there to the R console) and you will get a result and a new prompt.

```
> hist(d.sport[,"kugel"])
>
```

- An incomplete statement is automatically continued on the the following lines until the statement is syntactically complete (ie., R has found the closing “)”)

```
> plot(d.sport[,"kugel"],
+
+   d.sport[,"speer"])
>
```


R statements

An R statement[4] is typically either

- a name of an object $\longrightarrow$ object is displayed

```
> d.sport
```

- a call to a function $\longrightarrow$ graphical or numerical result is shown

```
> hist(d.sport[,"kugel"])
```

- an assignment

```
> a <- 2*pi/360
```

  or

```
> mn <- mean(d.sport[,"kugel"])
```

  which stores the result the numerical evaluation `2*pi/360` or `mean(d.sport[,"kugel"]` in new objects with the names `a` or `mn` , respectively.

[4] R “statement”: more precisely R “function call”

- a comment that is not processed

```
> # any text after a hash character on a line
> # is considered as comment and is not processed
> # by R
```


Calling R functions

- R functions typically have multiple arguments that all have names. To see the complete list of arguments of a function (and their default values) type `args(`*functionname*`)`

```
> args(var)
function (x, y = NULL, na.rm = FALSE, use)
NULL
```

- argument values may be passed to the function either by name

```
> var(x=d.sport[, "kugel"], na.rm=TRUE)
```

- or by position

```
> var(d.sport[, "kugel"], , TRUE)
```

- convention is to specify values for the first (and maybe second) argument by position and for the remaining arguments by name

```
> var(d.sport[, "kugel"], na.rm=TRUE)
```

0.6 Reading/Writing Data from/to Files

Read a file in table format and create a data frame (= data matrix) from it (with cases corresponding to lines and variables to columns):

- text (ASCII) files:
```
> read.table(file, header = FALSE, sep = "",
+     dec = ".", row.names, col.names,...)
```
- controlling columns delimiters and decimal "points"
```
> read.csv(file, sep = ",", dec=".",...)
> read.csv2(file, sep = ";", dec=",",...)
```
- Get all arguments and defaults by typing
```
?read.table
```


Some Examples

- Get a dataset from a text file on the web and assign a name to it:
```
> d.sport <- read.table(
+   "http://stat.ethz.ch/Teaching/Datasets/WBL/sport.dat",
+   header = TRUE)
```
- For data files with a one-line header (of column names), you need to set the option `header = TRUE`,
```
> d... <- read.table(... , header = TRUE)
```
- To download the file first to the local computer, R provides
```
> download.file(
+   "http://stat.ethz.ch/Teaching/Datasets/WBL/sport.dat",
+   destfile = "sport_data.txt")
```
- Use file browser (of the underlying operating system) to open the file: s
```
> d.sport <- read.table(file.choose(), header = TRUE)
```


Reading Data (continued)

- Tab-separated text files:
```
> read.delim(file, sep = "\t", dec=".",...)
> read.delim2(file, sep = "\t", dec=",",...)
```
- Reading binary Rdata-files:
```
> load(file="myanalysis.Rdata")
> load(file="C:/myanalysis.Rdata")
```


Writing Data to Files

- Text-files:
```
> write.table(x, file = "", append = FALSE,
+     sep = " ",eol = "\n", na = "NA", dec = ".",
+     row.names = TRUE, col.names = TRUE, ...)
```
where `x` is the data object to be stored.
- Text files in CSV format:
```
> write.csv(...)
> write.csv2(...)
```
- binary Rdata-files:
```
> save(..., file, ascii = FALSE,...)
```
Example:
```
> x <- c(1:20)
> y <- d.sport[,"kugel"]
> save(x, y, file = "xy.Rdata")
```

0.7 R Workspace

- R stores all created "objects" in a user workspace. List the objects by either `ls()` or equivalently, `objects()` :

```
> ls()
[1] "a"       "d.sport" "mn"
```

- Objects have names like `a, fun, d.sport`
- Besides, R provides a huge number of functions and other objects
- You can see the function definition ("source") by typing its name without `()`:

```
> read.table
```


0.8 Getting Help

- Documentation on the arguments etc. of a function (or dataset provided by the system):
  `> help(hist)` or `?hist`
  On the help page, the section "See Also..." contains related functions that could help you further.
- Search for a specific keyword:
  `> help.search("matrix")` Lists packages and functions related to or using "matrix".
  Note: Takes a long time when you have many extra R packages installed
- For many functions and data sets, examples are provided on the help page (`?matrix`). You can execute them directly,
  `> example("matrix")`

Resources on the internet

- R's Project page `http://www.r-project.org/`[5]
- CRAN: use Swiss mirror[6] `http://cran.CH.r-project.org/`: Links to **Search** (several search possibilites), **Task Views** (thematic collections of functions), **Contributed** (electronic Documentation, Introductions) and **FAQs**.

The following list could be extended "infinitely":

- `http://search.r-project.org/`: Search specific for R, also accessed via R function `RSiteSearch()` . Functions, Help, etc.
- `http://www.rseek.org/`: A "Google-type" search specific for R. Delivers Functions, Help Forums, etc.

[5] all URLs on this page are "clickable"

[6] the Swiss CRAN mirror is at `stat.ethz.ch`

0.9 Leaving an R Session

- Always save your script (*.R) files first.
- Then quit the R session by
  `> q()`
  in RStudio this is the same as using Ctrl-Q (menu item Quit RStudio)
- You get the question:
  `Save workspace image? [y/n/c]:`
  If you answer "y", your objects will be available for your next session.
- Note that we usually answer "`n`" to have a "clean" workspace when you start again. To recreate your objects execute your R script again.

1. Basics

In this Chapter you will ...

... find out about vectors (numerical, logical, character)

... use R as a calculator

... learn how to select elements from a data set

... learn how to create and manipulate matrices

1.1 Vectors

Functions and operations are usually applied to whole "collections" instead of single items, including "vectors", "matrices", "data.frames" (`d.sport`)

- Numbers can be combined into "vectors" by the function `c()` ("combine"):

```
> v <- c(4,2,7,8,2)
> a <- c(3.1, 5, -0.7, 0.9, 1.7)
> u <- c(v,a)
> u
 [1]  4.0  2.0  7.0  8.0  2.0  3.1  5.0 -0.7  0.9  1.7
```


- Generate a sequence of consecutive integers:

```
> seq(1, 9)
[1] 1 2 3 4 5 6 7 8 9
```

Since such sequences are needed very often, a shorter form is `1:9`.

Sequence of evenly spaced numbers: Use argument `by` (default: 1):

```
> seq(0, 3, by=0.5)
[1] 0.0 0.5 1.0 1.5 2.0 2.5 3.0
```

- Repetition:

```
> rep(0.7, 5)
[1] 0.7 0.7 0.7 0.7 0.7
> rep(c(1, 3, 5), length=8)
[1] 1 3 5 1 3 5 1 3
```


- Basic functions for vectors:

Call, Example	Description
`length(v)`	length of a vector, number of elements
`sum(v)`	sum of all elements
`mean(v)`	arithmetic mean
`var(v)`	sample variance
`range(v)`	range

These functions have additional optional arguments. Check their help pages to find out more.

1.2 Arithmetic

Simple arithmetic is as expected:

- `> 2+5`

```
[1] 7
```

Operations: $+$ $-$ $*$ $/$ ^ (Exponentiation)

See `?Arithmetic`. A list of all available operators is found in the R language definition manual[7].

- Priorities as usual. Use parentheses!

```
> (2:5) ^ 2
[1]  4  9 16 25
```

- These operations are applied to vectors elementwise.

```
> (2:5) ^ c(2,3,1,0)
[1]  4 27  4  1
```

[7] http://cran.r-project.org/doc/manuals/R-lang.html#Operators

- Elements are recycled if operations are carried out with vectors that do not have the same length:

```
> (1:6)*(1:2)
[1]  1  4  3  8  5 12
> (1:5) - (0:1) ## with a warning
[1] 1 1 3 3 5
Warning message:
    longer object length is not a multiple of
    shorter object length in: (1:5) - (0:1)
> (1:6)-(0:1)    ## no warning
[1] 1 1 3 3 5 5
```

Be careful, there is no warning in the last case!

1.3 Character Vectors

- Character strings: `"abc"` , `"nut 999"`
  Combine strings into vector of "mode" character:

```
> names <- c("Urs", "Anna", "Max", "Pia")
```

- Length (in characters) of strings:

```
> nchar(names)
[1] 3 4 3 3
```

- String manipulations:

```
> substring(names,3,4)
[1] "s"  "na" "x"  "a"
> paste(names, "Z.")
[1] "Urs Z."  "Anna Z." "Max Z."  "Pia Z."
> paste("X",1:3, sep="")
[1] "X1" "X2" "X3"
```


1.4 Logical Vectors

- Logical vectors contain elements `TRUE`, `FALSE`, or `NA`

```
> rep(c(TRUE, FALSE), length=6)
[1]  TRUE FALSE  TRUE FALSE  TRUE FALSE
```

- Often result from comparisons with relational operators, see `?Comparison`

`<` `<=` `>` `>=` `==` `!=`

```
> (1:5) >= 3
[1] FALSE FALSE  TRUE  TRUE  TRUE
```

- operations with logical operators, see `?Logic`

`&` (and), | (or), `!` (not)

```
> a
[1]  3.1  5.0 -0.7  0.9  1.7
> i <- (2 < a) & (a < 5)

> i
[1]  TRUE FALSE FALSE FALSE FALSE
```

1.5 Selecting elements

Select elements from vectors or data.frames by `[`i_1`]` and `[`i_1`,`i_2`]`, where i_1 and i_2 are vectors with element indices

```
> v
[1] 4 2 7 8 2
> v[c(1,3,5)]
[1] 4 7 2
> d.sport[c(1,3,5),1:3]
           weit kugel hoch
OBRIEN     7.57 15.66  207
DVORAK     7.60 15.82  198
HAMALAINEN 7.48 16.32  198
```

Drop elements, via *negative* indices:

```
>   d.sport[-(3:14), c("kugel","punkte")]
         kugel punkte
OBRIEN   15.66   8824
BUSEMANN 13.60   8706
CHMARA   14.51   8249
```


Elements of data.frames can be selected by names of columns or rows:

```
> d.sport[c("OBRIEN","DVORAK"),          # 2 rows
+         c("kugel","speer","punkte")] # 3 columns
       kugel speer punkte
OBRIEN 15.66 66.90   8824
DVORAK 15.82 70.16   8664
```

One can also select elements by logical vectors:

```
> a
[1]  3.1  5.0 -0.7  0.9  1.7
> a[c(TRUE,FALSE,TRUE,TRUE,FALSE)]
[1]  3.1 -0.7  0.9
```

Similarly use logical operations to select from a data.frame

```
> d.sport[d.sport[,"kugel"] > 16, c(2,7)]
           kugel punkte
HAMALAINEN 16.32   8613
PENALVER   16.91   8307
SMITH      16.97   8271
```


1.6 Matrices

Matrices are “data tables” like data.frames, but they can only contain data of a single type (numeric, character, logical, . . .)

- Generate a matrix (method 1):

```
> m1 <- matrix(1:6, nrow=2, ncol=3); m1
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> m2 <- matrix(1:6, ncol=2, byrow=TRUE); m2
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
```

- Transpose: `t(m1)` equals `m2`.
- Selection of elements as with data.frames:

```
> m1[2, 2:3]
[1] 4 6
```


- Generate a matrix (method 2):

```
> rbind(m1, -(1:3)) ## add row
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
[3,]   -1   -2   -3
> cbind(m2, 100) ## add column
     [,1] [,2] [,3]
[1,]    1    2  100
[2,]    3    4  100
[3,]    5    6  100
```

- Matrix multiplication:

```
> A <- m1 %*% m2; A
     [,1] [,2]
[1,]   35   44
[2,]   44   56
```

- Functions for linear algebra are available, e.g., $x = A^{-1}b$

```
> b <- 2:3
> x <- solve(A, b) ; x
[1] -0.83333  0.70833
> A %*% x # == b  -- as 1-col. matrix (!)
     [,1]
[1,]    2
[2,]    3
```

see `?solve`, `?crossprod`, `?qr`, `?eigen`, `?svd`,...[8].

[8] or e.g. http://www.statmethods.net/advstats/matrix.html

2. Simple Statistics

In this Chapter you will ...

... learn how to obtain information on R objects

... repeat simple functions for descriptive statistics

... learn about factor variables

... compare groups of data

... perform a simple hypothesis test

2.1 Useful summary functions for objects

To get an overview of a data set and a summary of its variables:

- Dimension of data set

```
> dim(d.sport)
[1] 15  7
> nrow(d.sport); ncol(d.sport)
[1] 15
[1] 7
```

- First/Last few lines of a data set

```
> head(d.sport,n=2) ## default is n=6
         weit kugel hoch  disc stab speer punkte
OBRIEN   7.57 15.66  207 48.78  500 66.90   8824
BUSEMANN 8.07 13.60  204 45.04  480 66.86   8706
> tail(d.sport,n=1) ## default is n=6
       weit kugel hoch disc stab speer punkte
CHMARA 7.75 14.51  210 42.6  490 54.84   8249
```


- Get the names of the variables of a data.frame

```
> names(d.sport)
[1] "weit"   "kugel"  "hoch"   "disc"   "stab"   "speer"
[7] "punkte"
```

- Show the structure of an R object

```
> str(d.sport)
'data.frame': 15 obs. of  7 variables:
 $ weit  : num  7.57 8.07 7.6 7.77 7.48 7.88 7.64 7.61 7.27 7
 $ kugel : num  15.7 13.6 15.8 15.3 16.3 ...
 $ hoch  : int  207 204 198 204 198 201 195 213 207 204 ...
 $ disc  : num  48.8 45 46.3 49.8 49.6 ...
 $ stab  : int  500 480 470 510 500 540 540 520 470 470 ...
 $ speer : num  66.9 66.9 70.2 65.7 57.7 ...
 $ punkte: int  8824 8706 8664 8644 8613 8543 8422 8318 8307 8
> str(d.sport[, "kugel"])
 num [1:15] 15.7 13.6 15.8 15.3 16.3 ...
> str(hist)
function (x, ...)
```

- Show a summary of the values of the variables in a data.frame (min, quartiles and max for numeric variables, counts for factors – see below)

```
> summary(d.sport)
      weit            kugel            hoch             disc
 Min.   :7.25    Min.   :13.5    Min.   :195    Min.   :42.6
 1st Qu.:7.47    1st Qu.:14.6    1st Qu.:196    1st Qu.:44.3
 Median :7.60    Median :15.3    Median :204    Median :45.9
 Mean   :7.60    Mean   :15.2    Mean   :202    Mean   :46.4
 3rd Qu.:7.76    3rd Qu.:15.7    3rd Qu.:206    3rd Qu.:48.9
 Max.   :8.07    Max.   :17.0    Max.   :213    Max.   :49.8
      stab           speer           punkte
 Min.   :470    Min.   :52.2    Min.   :8249
 1st Qu.:480    1st Qu.:57.4    1st Qu.:8278
 Median :500    Median :64.3    Median :8318
 Mean   :498    Mean   :62.0    Mean   :8445
 3rd Qu.:510    3rd Qu.:66.5    3rd Qu.:8628
 Max.   :540    Max.   :70.2    Max.   :8824
```


2.2 Simple Statistical Functions

- Estimation of a “location parameter”: `mean(x)` `median(x)`

```
> mean(d.sport[,"kugel"])
[1] 15.199
> median(d.sport[,"kugel"])
[1] 15.31
```

- Quantiles `quantile(x)`

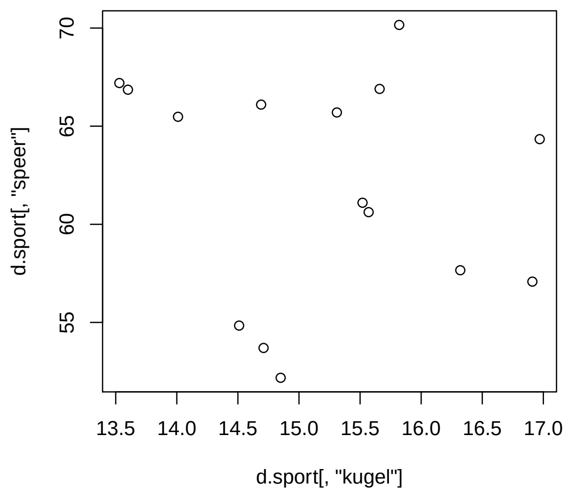
```
> quantile(d.sport[,"kugel"])
   0%   25%   50%   75%  100%
13.53 14.60 15.31 15.74 16.97
```

- Variance: `var(x)`

```
> var(d.sport[,"kugel"])
[1] 1.1445
```


- Correlation: `cor(x,y)` – **Look at a plot before!**

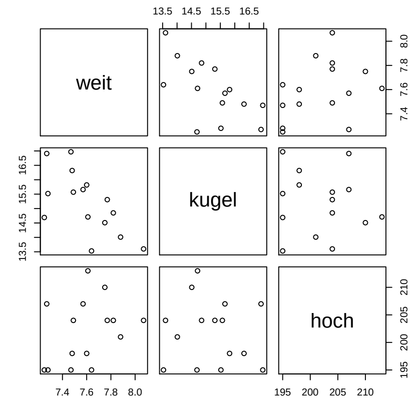
```
> plot(d.sport[,"kugel"], d.sport[,"speer"])
```




```
> cor(d.sport[,"kugel"], d.sport[,"speer"])
[1] -0.14645
```


- Correlation matrix:

```
> pairs(d.sport[,1:3])
```




```
> cor(d.sport[,1:3])
          weit     kugel      hoch
weit   1.00000 -0.630171  0.337752
kugel -0.63017  1.000000 -0.092819
hoch   0.33775 -0.092819  1.000000
```

2.3 Factors

Groups, or categorial variables are represented by factors, e.g. ID of a measurement station, type of species, type of treatment, etc.

In statistical analyses categorical variables MUST be coded as factors to produce correct results (e.g. in analysis of variance or for regression).

⟶ ALWAYS check your data (by `str()`) before starting an analysis.

To produce a factor variable:

- use `c()`, `rep()`, `seq()` to define a numeric or character vector
- and then the function `as.factor()`.

An example: Suppose the athletes listed in `d.sport` belong to 3 teams:

```
> teamnum <- rep(1:3,each=5)

> d.sport[,"team"] <- as.factor(teamnum)
> str(d.sport)
'data.frame': 15 obs. of  8 variables:
 $ weit  : num  7.57 8.07 7.6 7.77 7.48 7.88 7.64 7.61 7.27 7.4..
 $ kugel : num  15.7 13.6 15.8 15.3 16.3 ...
 $ hoch  : int  207 204 198 204 198 201 195 213 207 204 ...
 $ disc  : num  48.8 45 46.3 49.8 49.6 ...
 $ stab  : int  500 480 470 510 500 540 540 520 470 470 ...
 $ speer : num  66.9 66.9 70.2 65.7 57.7 ...
 $ punkte: int  8824 8706 8664 8644 8613 8543 8422 8318 8307 83..
 $ team  : Factor w/ 3 levels "1","2","3": 1 1 1 1 1 2 2 2 2 2 ..

> levels(d.sport[,"team"])
[1] "1" "2" "3"

> levels(d.sport[,"team"]) <-
+      c("Zurich","New York","Tokyo")
```



```
> head(d.sport,n=10)
           weit kugel hoch  disc stab speer punkte     team
OBRIEN     7.57 15.66  207 48.78  500 66.90   8824   Zurich
BUSEMANN   8.07 13.60  204 45.04  480 66.86   8706   Zurich
DVORAK     7.60 15.82  198 46.28  470 70.16   8664   Zurich
FRITZ      7.77 15.31  204 49.84  510 65.70   8644   Zurich
HAMALAINEN 7.48 16.32  198 49.62  500 57.66   8613   Zurich
NOOL       7.88 14.01  201 42.98  540 65.48   8543 New York
ZMELIK     7.64 13.53  195 43.44  540 67.20   8422 New York
GANIYEV    7.61 14.71  213 44.86  520 53.70   8318 New York
PENALVER   7.27 16.91  207 48.92  470 57.08   8307 New York
HUFFINS    7.49 15.57  204 48.72  470 60.62   8300 New York

> nlevels(d.sport[,"team"])
[1] 3
```


2.4 Simple Statistical Functions (cont'd)

```
> summary(d.sport)
      weit           kugel            hoch            disc
 Min.   :7.25   Min.   :13.5   Min.   :195   Min.   :42.6
 1st Qu.:7.47   1st Qu.:14.6   1st Qu.:196   1st Qu.:44.3
 Median :7.60   Median :15.3   Median :204   Median :45.9
 Mean   :7.60   Mean   :15.2   Mean   :202   Mean   :46.4
 3rd Qu.:7.76   3rd Qu.:15.7   3rd Qu.:206   3rd Qu.:48.9
 Max.   :8.07   Max.   :17.0   Max.   :213   Max.   :49.8
      stab          speer          punkte           team
 Min.   :470   Min.   :52.2   Min.   :8249   Zurich  :5
 1st Qu.:480   1st Qu.:57.4   1st Qu.:8278   New York:5
 Median :500   Median :64.3   Median :8318   Tokyo   :5
 Mean   :498   Mean   :62.0   Mean   :8445
 3rd Qu.:510   3rd Qu.:66.5   3rd Qu.:8628
 Max.   :540   Max.   :70.2   Max.   :8824
```

- Count number of cases with same value:

```
> table(d.sport[,"team"])
 Zurich New York    Tokyo
      5        5        5
```

- Cross-table

```
> table(d.sport[,"kugel"],d.sport[,"team"])
        Zurich New York Tokyo
 13.53       0        1     0
 13.6        1        0     0
 14.01       0        1     0
 14.51       0        0     1
 14.69       0        0     1
  ...
```

$\longrightarrow$ The table function is not useful for numerical variables. Use `cut()` (see next slide).

- Subdivide a numerical variable into intervals, e.g. for cross-tables or plots: `cut()`

```
> table( cut( d.sport[,"kugel"], breaks=4 ),
+     d.sport[,"team"] )
              Zurich New York Tokyo
  (13.5,14.4]      1        2     0
  (14.4,15.2]      0        1     3
  (15.2,16.1]      3        1     1
  (16.1,17]        1        1     1
```


2.5 Comparison of Groups

Often in statistics, we want to compare measurements for different groups.

`d.sport` now contains data for 3 different teams with 5 people each.

Let's store the kugel results for each group separately:

```
> y1 <- d.sport[d.sport[,"team"]=="Zurich","kugel"]; y1
[1] 15.66 13.60 15.82 15.31 16.32

> y2 <- d.sport[d.sport[,"team"]=="New York","kugel"]
> y3 <- d.sport[d.sport[,"team"]=="Tokyo","kugel"]
```

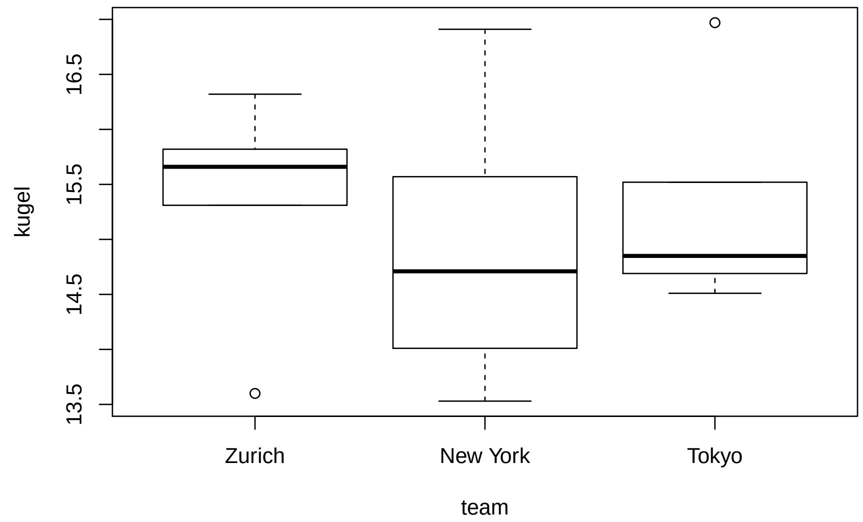
Comparison of the different groups:

- look at a cross-table (see above)
- plot the distribution of the results in each group (better!)
- use a statistical test to compare groups

$\longrightarrow$ Build hypotheses based on plots and prior knowledge!

Boxplot for several groups

```
> boxplot(y1,y2,y3, ylab="kugel", xlab="team",
+     names=levels(d.sport[,"team"]))
```



2.6 Hypothesis Tests

Do two groups differ in their ”location”? (t-test in Exercises)
No assumption about distribution of data:
$\longrightarrow$ Wilcoxon’s Rank Sum Test

```
> wilcox.test(y1,y3,paired=FALSE)
Wilcoxon rank sum test

data:  y1 and y3
W = 15, p-value = 0.6905
alternative hypothesis: true location shift is not equal to 0
> wilcox.test(y1,y2,paired=FALSE)
Wilcoxon rank sum test

data:  y1 and y2
W = 16, p-value = 0.5476
alternative hypothesis: true location shift is not equal to 0
```


3. Missing Values

In this Chapter you will ...

... see how missing values are specified
... learn how functions deal with missing values
... find out how to properly read in data with missing values

3.1 Identifying Missing Values

In practice, some data values may be missing.

- Here, we fake this situation

```
> kugel <- d.sport[,"kugel"]
> kugel[2] <- NA
> kugel
 [1] 15.66    NA 15.82 15.31 16.32 14.01 13.53 14.71 16.91
[10] 15.57 14.85 15.52 16.97 14.69 14.51
```

`NA` means ‘**N**ot **A**vailable’ and typically indicates missing data.

—

- Which elements of `kugel` are missing?

```
> kugel == NA
 [1] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
```

This is not what we expected, we have to use `is.na()` instead

```
> is.na(kugel)
 [1] FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[10] FALSE FALSE FALSE FALSE FALSE FALSE
```


3.2 Missing Values and Function Calls

- Applying functions to vectors with missing values:

```
> mean(kugel)
[1] NA
> mean(kugel, na.rm=TRUE)
[1] 15.313
```

- Other simple functions also have the `na.rm` argument
- For more sophisticated functions (e.g. `wilcox.test`), the argument `na.action` defines how missing values are handled. `na.action=na.omit`: omit cases with NAs
- Plotting functions normally work with NAs.

- Manually dropping the NA elements:

```
> kugel[!is.na(kugel)]
 [1] 15.66 15.82 15.31 16.32 14.01 13.53 14.71 16.91 15.57
[10] 14.85 15.52 16.97 14.69 14.51
```

- more general method

```
> na.omit(kugel)
```

`na.omit(df)` drops rows of a data.frame `df` that contain missing value(s).

3.3 Reading data sets with coded missing values

- How to specify missings when reading in data:

```
> d.dat <- read.table(..., na.strings=c(".","-999"))
```

Default: empty fields are taken as NA for numerical variables.

- ... or clean your data later:

```
> d.dat[d.dat[, "x"]==-999, "x"] <- NA
```


4. Write your own Function

In this chapter you will ...

... learn how to write your own functions

... and use them in other functions

... see a simple function example

Syntax:

fnname <- function(arg(s)) { statements }

A simple function: Get the maximal value of a vector and its index.

```
> f.maxi <- function(data) {
+   mx <- max(data, na.rm=TRUE) # get max element
+   i <- match(mx, data)        # position of max in data
+   c(max=mx, pos=i)            # result of function
+ }
```

Output of f.maxi is a named vector. The use of `return()` is optional.

```
> f.maxi(c(3,4,78,2))
max pos
 78   3
```

(Note: R provides the function `which.max`)

This function can now be used in `apply` :

```
> apply(d.sport, 2, f.maxi)
    weit kugel hoch  disc stab speer punkte
max 8.07 16.97  213 49.84  540 70.16   8824
pos 2.00 13.00    8  4.00    6  3.00      1
```

Note: Use functions when you can. They make your code more legible and simplify the analysis.

You can include the functions at the end of your main programme, or collect all your functions in one R-script (e.g. `myfunctions.R`) and make the functions available by

```
> source("myfunctions.R")
```

More about best-practices in programming will follow in the last block of this lecture course.

R is open-source: Look at, and learn from, the existing functions!

5. Scatter- and Boxplots

In this lecture you will ...

... get a flavour of graphics systems available in R

... learn how to create scatter- and boxplots

... learn how to use formulae in plots

... learn how to add axis labels and titles to plots

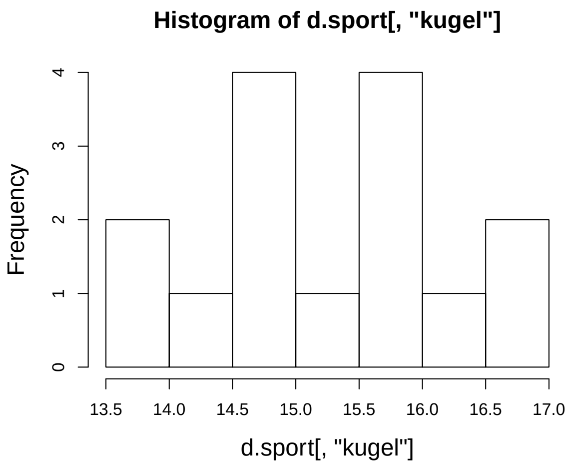
... learn to select color, type and size of symbols

... learn how to control the scales of axes

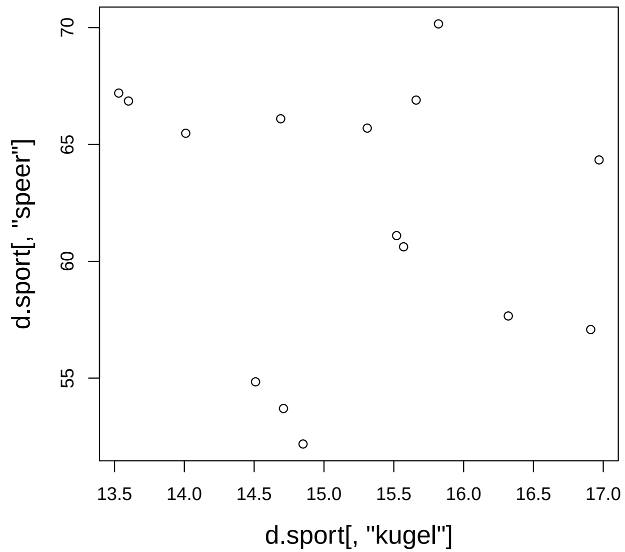
5.1 Overview

Several R graphics functions have been presented so far:

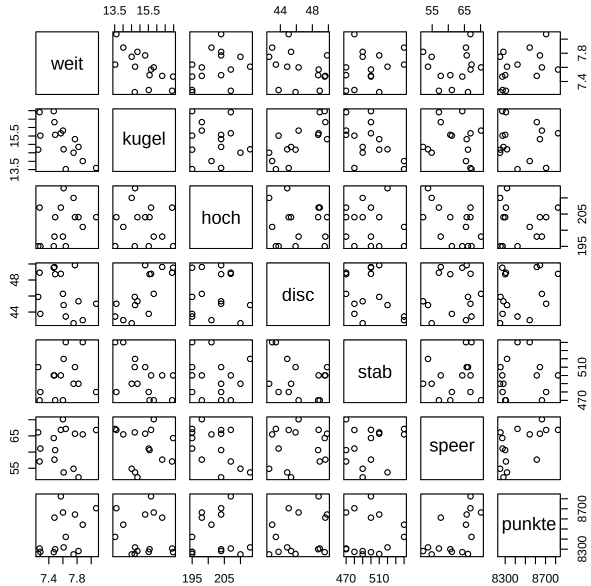
```
> hist(d.sport[,"kugel"])
```



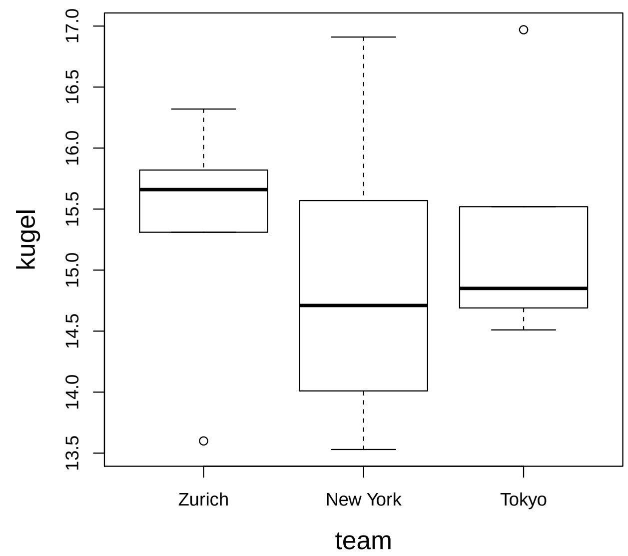

```
> plot(d.sport[,"kugel"], d.sport[,"speer"])
```



```
> pairs(d.sport)
```




```
> boxplot(y1,y2,y3,ylab="kugel",xlab="team")
```



Many more “standard” graphics functions to come:

`scatter.smooth , matplot , image ,...`

`lines , points , text ,...`

`par , identify , pdf , jpeg ,...`

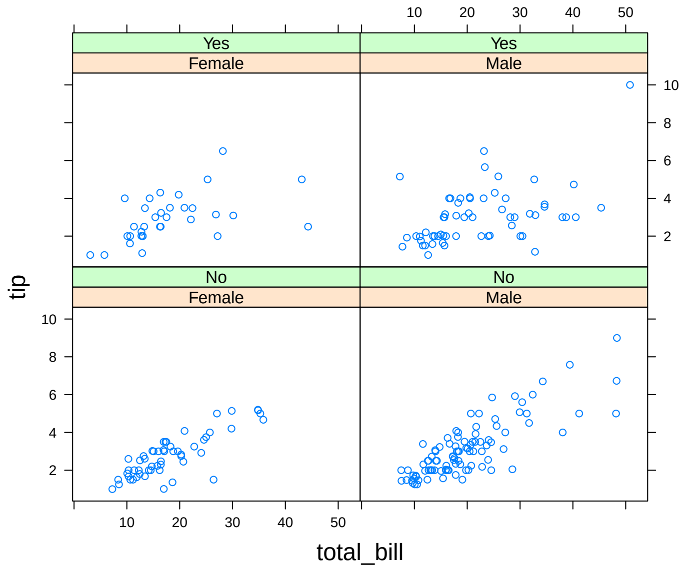
Alternatives to “standard” graphics functions

⇒ functions of package lattice

⇒ functions of package ggplot2

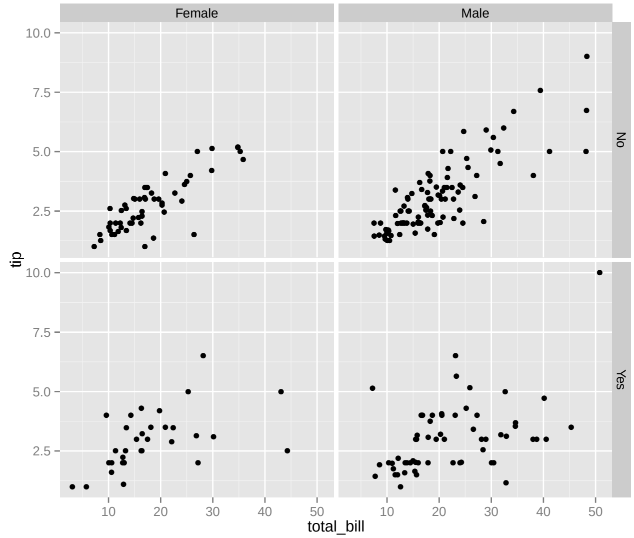
An example using function `xyplot` of package lattice

```
> data(tips, package="reshape"); library(lattice)
> xyplot(tip~total_bill|sex+smoker, data=tips)
```



Same plot using function `qplot` of package ggplot2

```
> library(ggplot2)
> qplot(x=total_bill, y=tip, data=tips,
+       facets=smoker~sex)
```



Five kinds of standard R graphics functions:

- High-level plotting functions such as `plot`
  ⇒ *to generate a new graphical display of data.*
- Low-level plotting functions such as `lines`
  ⇒ *to add further graphical elements to an existing graph.*
- "Interactive" functions such as `identify`
  ⇒ *to amend or collect information interactively from a graph.*
- "Device" control functions such as `pdf`
  ⇒ *to manipulate windows and files that display or store graphs.*
- "Control" functions such as `par`
  ⇒ *to control the appearance of graphs.*

5.2 Scatterplot

Display of the values of two variables plotted against each other.

Syntax:

`plot(`x`, `y`, main=`c_1`, xlab=`c_2`, ylab=`c_3`, ...)`

x,y: two numeric vectors (must have same length)

c_1, c_2, c_2: any character strings (must be quoted)

For the meaning of `...` : ⇒ cf. `?plot`

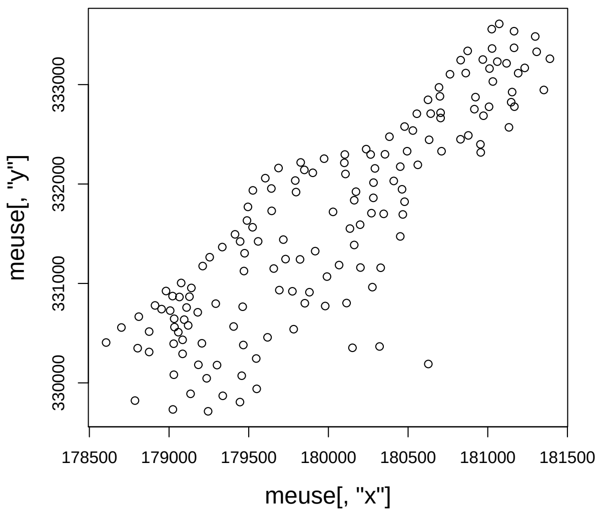
Example: Exploring Meuse data on heavy metals in soil

```
> library(sp); data(meuse)
> str(meuse)
```

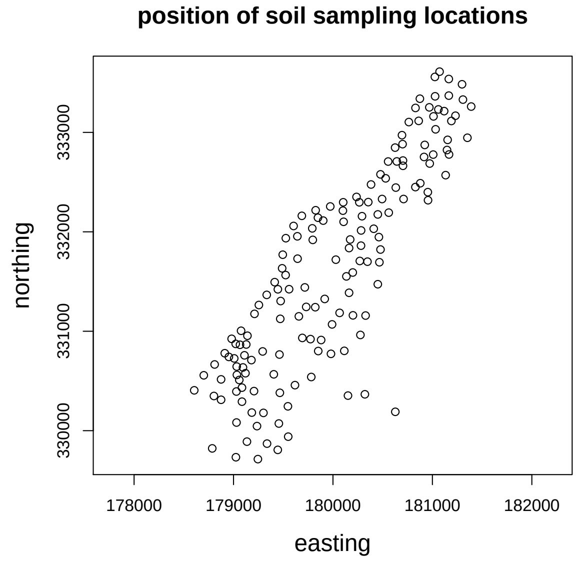


```
'data.frame': 155 obs. of  14 variables:
 $ x      : num  181072 181025 181165 181298 181307 ...
 $ y      : num  333611 333558 333537 333484 333330 ...
 $ cadmium: num  11.7 8.6 6.5 2.6 2.8 3 3.2 2.8 2.4 1.6 ...
 $ copper : num  85 81 68 81 48 61 31 29 37 24 ...
 $ lead   : num  299 277 199 116 117 137 132 150 133 80 ...
 $ zinc   : num  1022 1141 640 257 269 ...
 $ elev   : num  7.91 6.98 7.8 7.66 7.48 ...
 $ dist   : num  0.00136 0.01222 0.10303 0.19009 0.27709 ...
 $ om     : num  13.6 14 13 8 8.7 7.8 9.2 9.5 10.6 6.3 ...
 $ ffreq  : Factor w/ 3 levels "1","2","3": 1 1 1 1 1 1 1 ..
 $ soil   : Factor w/ 3 levels "1","2","3": 1 1 1 2 2 2 2 ..
 $ lime   : Factor w/ 2 levels "0","1": 2 2 2 1 1 1 1 1 ..
 $ landuse: Factor w/ 15 levels "Aa","Ab","Ag",..: 4 4 4 1..
 $ dist.m : num  50 30 150 270 380 470 240 120 240 420 ...
```

```
> plot(x=meuse[,"x"], y=meuse[,"y"])
```



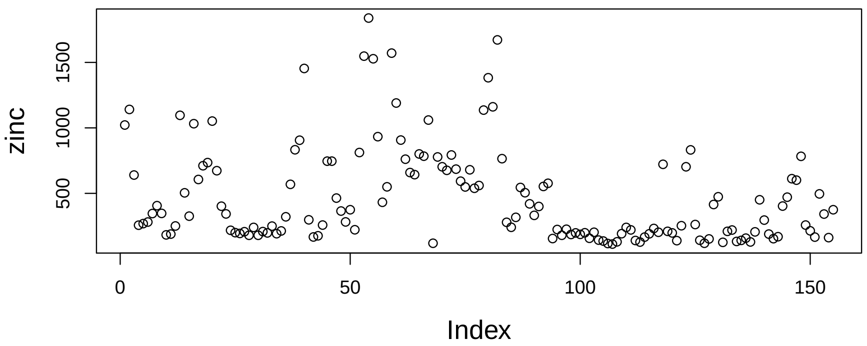

```
> plot(x=meuse[,"x"], y=meuse[,"y"], asp=1,
+       xlab="easting", ylab="northing",
+       main="position of soil sampling locations")
```



Three additional variants ways to invoke `plot` :

- Plot of the values of a single vector against the indices of the vector elements

```
> plot(meuse[,"zinc"], ylab="zinc")
```

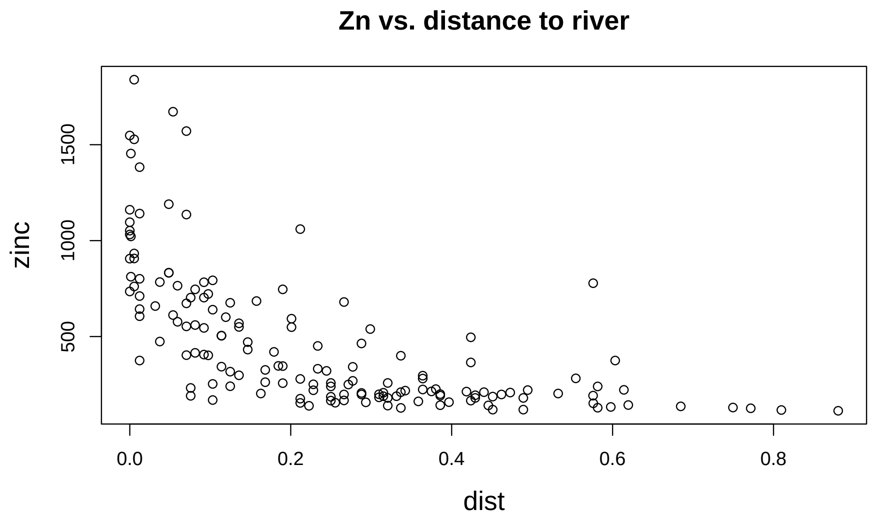


- Scatterplot of two columns of a matrix or a dataframe

```
> plot(meuse[,c("x","y")], asp=1)
```


- Use of a formula, e.g. $y \tilde{} x$, to specify the x- and y-variable out of a data frame (cf. `?plot.formula`)

```
> plot(zinc~dist, data=meuse,
+       main="Zn vs. distance to river")
```



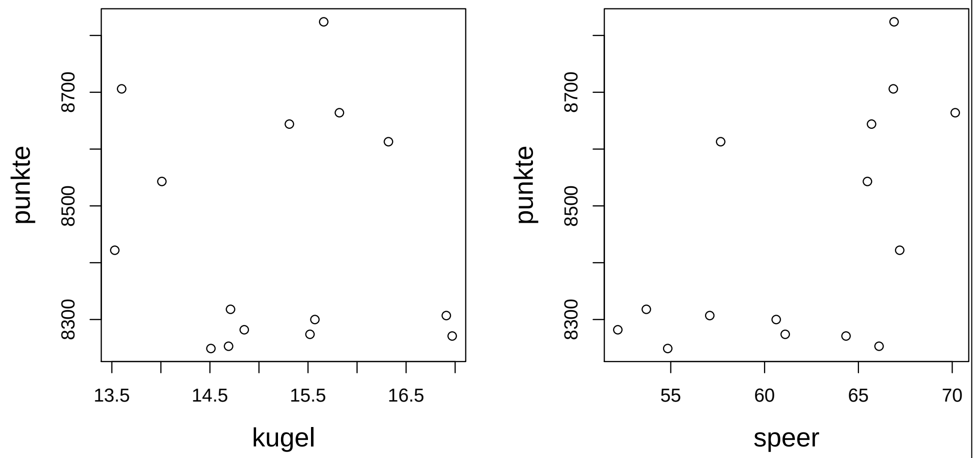
5.3 Digression: Statistical Models, Formula Objects

Statistics is concerned with relations between “variables”.
Prototype: Relationship between target variable `Y`
and explanatory variables `X1, X2, ...` ⇒ Regression.

- The symbolic notation of such a relation: `Y ~ X1 + X2`
  reads as “ `Y` is modelled as an (additive) function of `X1` and `X2` .
  This symbolic notation is also an R object (of class `formula`)
  (The notation is also used in other statistical packages.)

- Further example for use of a formula:

```
> plot(punkte~kugel+speer, data=d.sport)
```



gives 2 scatterplots with `punkte` (on vertical axis) plotted against `kugel` and `speer` (on horizontal axes), respectively.

5.4 Arguments common to many graphics functions

- `main="..."` , `xlab="..."` , `ylab="..."`

  `"..."` : any character string (must be quoted!)

  ⇒ to set title and labels of axes (cf. `?title`)

- `log="x"` , `log="y"` , `log="xy"`

  ⇒ for logarithmic scaling of axes (cf. `?plot.default`)

- `xlim=c(`$x_{\min}$`,`$x_{\max}$`)` , `ylim=c(`$y_{\min}$`,`$y_{\max}$`)` ,

  $x_{\min}$, $x_{\max}$, $y_{\min}$, $y_{\max}$: numeric scalars

  ⇒ to set range of values displayed (cf. `?plot.default`)

- `asp=`*n*

  n : numeric scalar

  ⇒ to set aspect ratio of axes (cf. `?plot.window`

Common arguments of `plot` (continued):

- `type=`*c*

  c: a single character such as `"p"` for points, `"l"` for lines, `"b"` for points **and** lines, ”n” for an “empty” plot, etc.

  ⇒ for selecting type of plot (cf. `?plot`)

- `pch=`*i* or `pch=`*c*

  i : an integer (vector); *c*: a single character such as `"a"` (or a vector of single-character strings)

  ⇒ for choosing symbols (cf. `?points`)

- `cex=`*n*

  n : a numeric (vector)

  ⇒ for choosing size of symbols (cf. `?plot.default`)

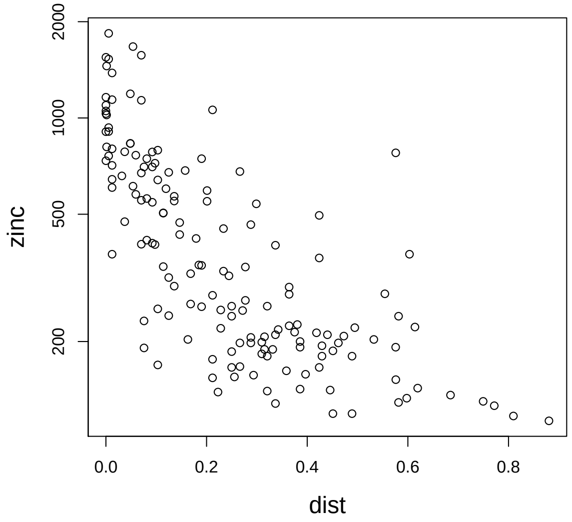
- `col=`*i* or `col=`*color*

  color: (vector with) keyword(s) such as `"red"` , `"blue"` , etc

  ⇒ for choosing color of symbols (cf. `?plot.default` and `colors()`)

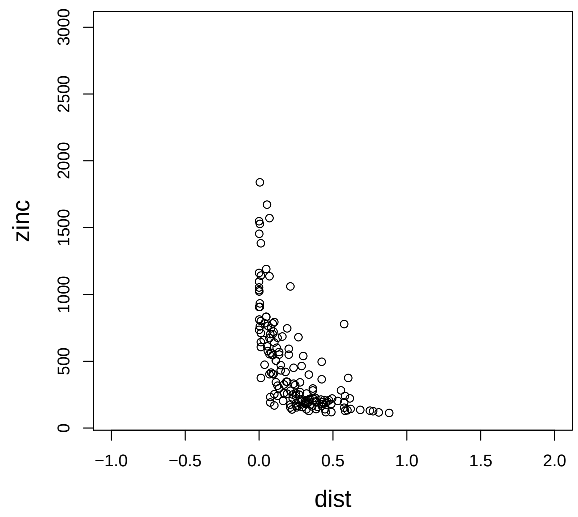
Example: logarithmic axes scale

```
> plot(zinc~dist, data=meuse, log="y")
```



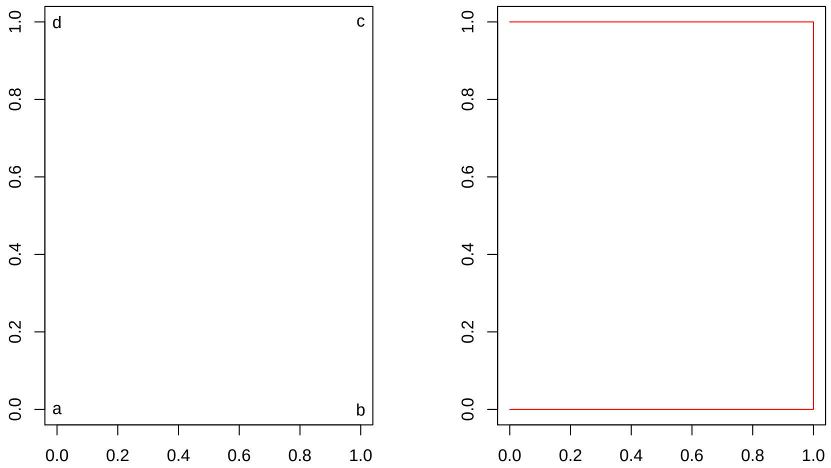
Example: setting the range of axes

```
> plot(zinc~dist, data=meuse,
+       xlim=c(-1,2), ylim=c(100,3000))
```



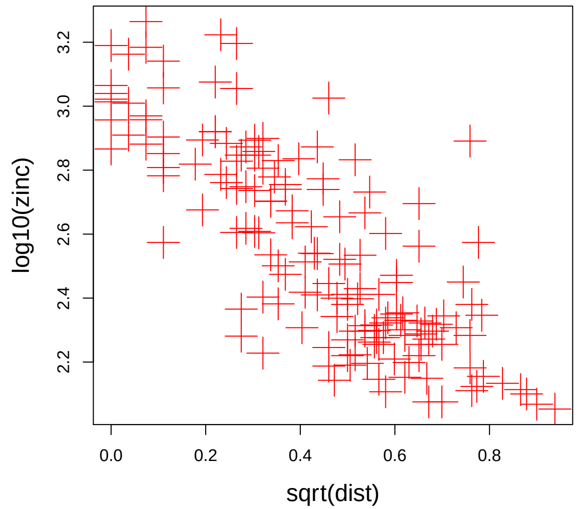
Example: connecting points by lines (cf. `?plot`)

```
> x <- c(0,1,1,0); y <- c(0,0,1,1)
> plot(x=x,y=y,type="p",xlab="",ylab="",pch=letters[1:4])
> plot(x=x,y=y,type="l",xlab="",ylab="",col="red")
```



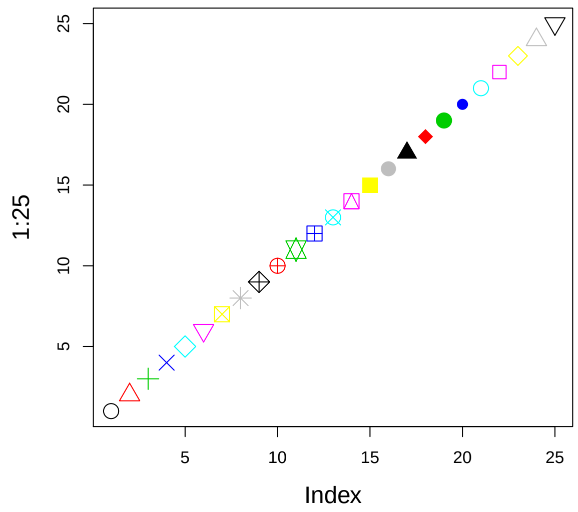
Example: choosing symbol type, color and size (cf. `?points`)

```
> plot(log10(zinc)~sqrt(dist), data=meuse,
+       pch=3, col="red", cex=3)
```



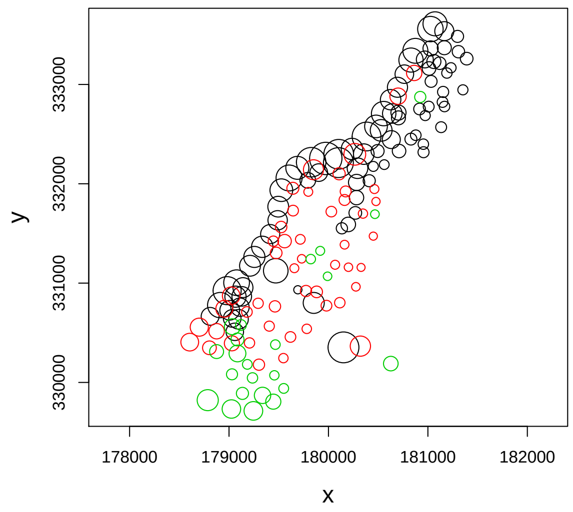
Example: choosing symbol type, color and size

```
> plot(1:25, pch=1:25, cex=2, col=1:8)
```



Example: choosing symbol type, color and size

```
> plot(y~x, data=meuse, asp=1, ## [asp]ect ratio := 1
+      col=ffreq, cex=sqrt(zinc)/10)
```



5.5 Boxplot

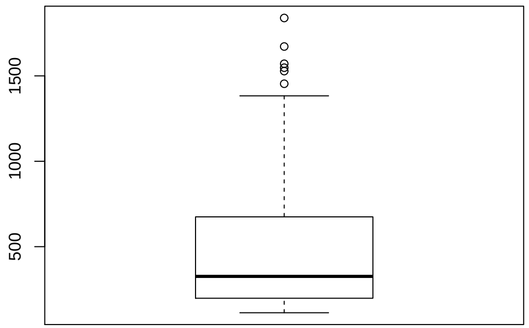
Syntax:

boxplot(x_1, x_2, ..., notch=l_1, horizontal=l_2, ...)

- $x_1, x_2, \ldots$: numeric vectors
- l_1 (logical): controls whether "notches" are added to (coarsely) test whether medians of $x_1, x_2, \ldots$ significantly differ (5% significance level)
- l_2 (logical): controls whether horizontal boxplots are generated
- ... : many more arguments (cf. `?boxplot`)

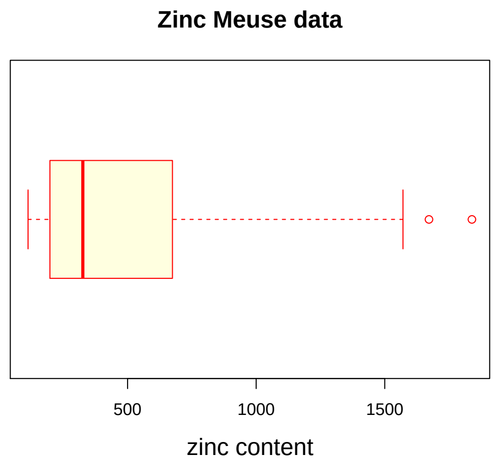
Example: a single boxplot

```
> boxplot(meuse[,"zinc"])
```



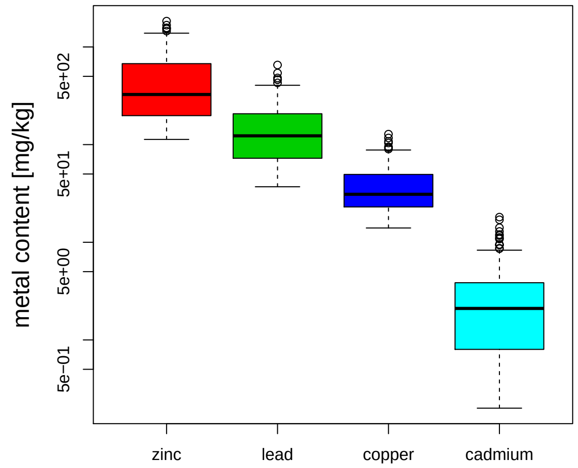
Example: a single boxplot with some decoration

```
> boxplot(x=meuse[,"zinc"], horizontal=TRUE, range=2,
+         col="lightyellow", border="red",
+         xlab="zinc content", main="Zinc Meuse data")
```



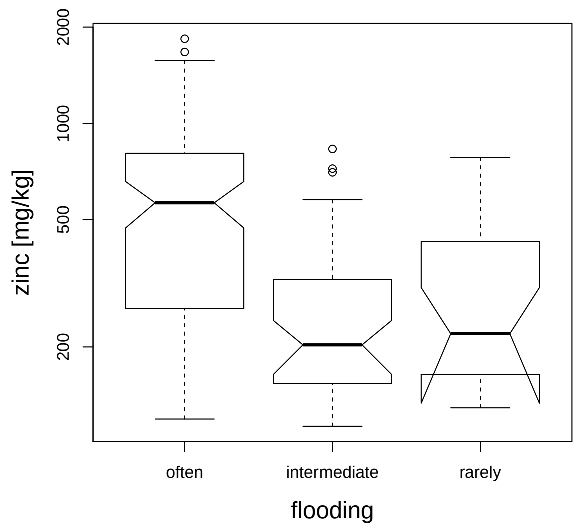
Example: variant to generate boxplots of several variables

```
> boxplot(meuse[,c("zinc","lead","copper","cadmium")],
+      log="y", ylab="metal content [mg/kg]", col = 2:5)
```



Example: boxplot of one variable for several levels of a factor

```
> boxplot(zinc~ffreq, data=meuse, log="y", notch=TRUE,
+         names= c("often", "intermediate", "rarely"),
+         xlab= "flooding", ylab= "zinc [mg/kg]")
```



In this lecture you have ...

... got a flavour of graphics systems available in R
 ⇒ **"standard" graphics**, **`lattice`** , **`ggplot2`**

... learnt how to create scatterplots and boxplots
 ⇒ functions **`plot`** , **`boxplot`**

... learnt how to use formulae for generating plots

... learnt how to connect points in a scatterplot by lines
 ⇒ argument **`type`**

... learnt how to add axis labels and titles to plots
 ⇒ arguments **`main`** , **`xlab`** , **`ylab`**

... learnt to select color, type and size of symbols
 ⇒ arguments **`col`** , **`pch`** , **`cex`**

... learnt how to control the scales of axes
 ⇒ arguments **`asp`** , **`log`** , **`xlim`** , **`ylim`**

6. Controlling the visual aspects of a graphic

In this lecture you will learn ...

... how to add points and lines to an existing plot,

... how to amend a plot by additional text and a legend,

... about the `par` function for fine-tuning your graphics,

... how to arrange several plots in one graphic,

... how to manage colors,

and in this week's exercise series you will explore additional high-level plotting functions

6.1 Adding further points and lines to a graphic

Use `points` to add further points to a graph created before by a high-level plotting function such as `plot`.

Syntax:

`points(x=`*x*`, y=`*y*`, pch=`i_1`, col=`i_2 or `col=`*color*`, cex=`*n*`)`

x, *y*: two numeric vectors

i_1, i_2: integers (scalars or vectors)

color: color name (scalar or vector)

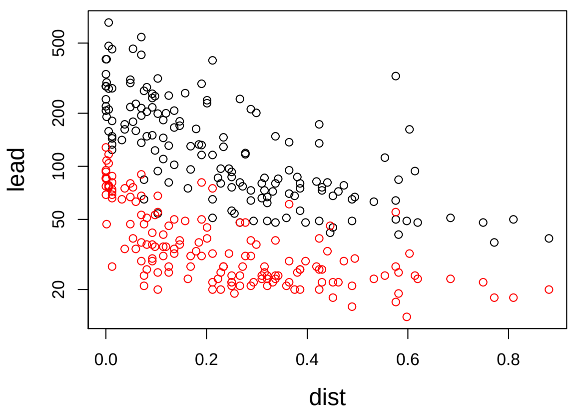
n: numeric (scalar or vector)

Remarks:

- ± same arguments as for `plot`
- `points` can also be used with `formula` and `data` arguments (cf. `?points.formula`)

Example: adding Cu data to a plot of `lead~dist` for Meuse data

```
> plot(lead~dist, data=meuse, log="y",
+      ylim=range(c(copper,lead)))
> points(copper~dist, data=meuse, col="red")
```



Use `lines` to add lines that connect successive points to an existing plot.

Syntax:

`lines(x=`*x*`, y=`*y*`, lty=`*i* or `lty=`*line_type*`, lwd=`*n*`, ...)`

x, *y*: two numeric vectors

i: integer (scalar) to select line type (cf. `?par`)

line_type: integer or keyword such as `"dotted"` to select line type (cf. `?par`)

n: numeric scalar to select line width

...: further arguments such as `col` to select line color

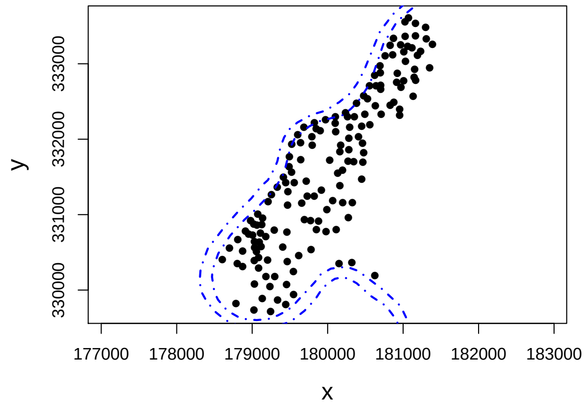
Remarks:

- ± same arguments as for `plot` and `points`
- `lines` can also be used with `formula` and `data` arguments (cf. `?lines.formula`)

Example: adding outline of river Meuse to plot of sampling locations

```
> data(meuse.riv)
> str(meuse.riv)
 num [1:176, 1:2] 182004 182137 182252 182314 182332 ...

> plot(y~x, data=meuse, asp=1, pch=16)
> lines(meuse.riv, lty="dotdash", lwd=2, col="blue")
```



Use `abline` to add straight lines to an existing plot.

Syntax:

```
abline(v=x, ...)
abline(h=y, ...)
abline(a=n1, b=n1, ...)
```

x: coordinate(s) where to draw vertical straight line(s) (scalar or vector)

y: coordinate(s) where to draw horizontal straight line(s) (scalar or vector)

n_1, n_2: numeric scalars for intercept and slope of straight line

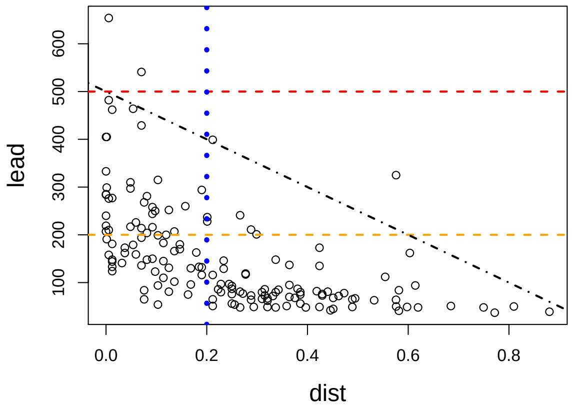
`...`: further arguments such as `col`, `lty`, `lwd`

Remark:

- the straight lines extend over the entire plot window

Example: adding straight lines to a plot

```
> plot(lead~dist, data=meuse)
> abline(h=c(200, 500), col=c("orange", "red"),
+        lty="dashed", lwd=2)
> abline(v=0.2, col=4, lty=3, lwd=5)
> abline(a=500, b=-500, lty="dotdash", lwd=2,
+        col="black")
```



Further useful low-level plotting functions

- `segments` adds arbitrary line segments to an existing plot, cf. `?segments`
- `arrows` adds arrows to a plot (± same syntax as `segments`, cf. `?arrows`)
- `polygon` adds a polygon to an existing plot, cf. `?polygon`

6.2 Amending plots by additional text and legends

Points in a scatterplot are labelled by `text` .

Syntax:

`text(x=`*x*`, y=`*y*`, labels=`*c* `, pos=`*i*`, ...)`

x, *y*: two numeric vectors

c: vector of character strings with the text to label the points

i: integer (vector) to control whether labels are plotted below (1), to the left (2), above (3) or to the right (4) of the points (scalar or vector)

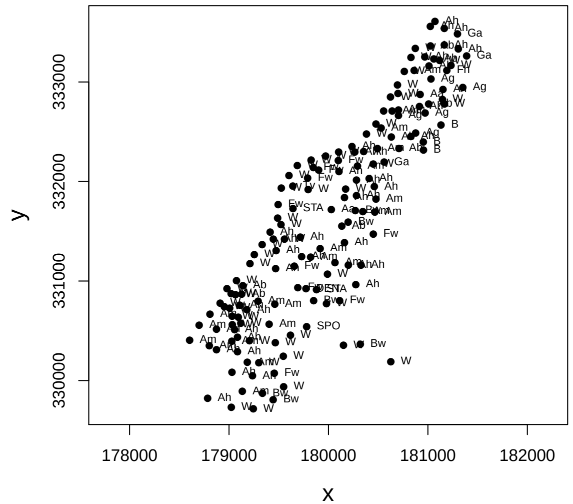
`...`: further arguments such as `col` and `cex`

Remarks:

- *x* and *y* may specify arbitrary coordinates within the plot window
- one can also use `formula` (along with a `data` argument) in `text` (cf. `plot.formula`)

Example: labelling sample points of Meuse data by landuse info

```
> plot(y~x, data=meuse, asp=1, pch=16)
> text(y~x, data=meuse, labels=landuse, pos=4, cex=0.7)
```



More sophisticated text annotation is added by `legend` to a plot.

Syntax:

`legend(x=`*x*`, y=`*y*`, legend=`*c*`, pch=`i_1`, lty=`i_2 `,...)`

x, *y*: coordinates where the legend should be plotted

c: vector of character strings with labels of categories

i_1, i_2: vector of integers with type of plotting symbol **or** line type for categories

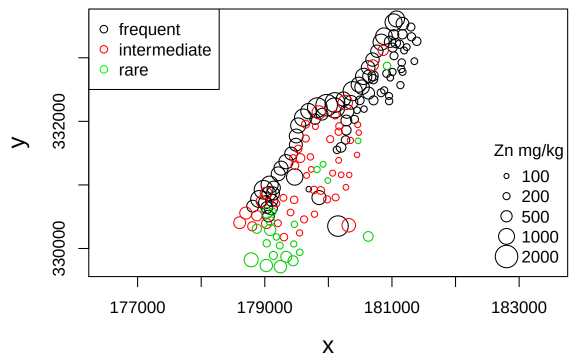
`...`: further arguments such as `col` and `cex`

Remarks:

- The position of the legend is either specified by *x* and *y* **or** by a keyword such as `"topright"` , `"bottomleft"` , etc. (cf. `legend` for allowed keywords)

Example: legends annotating flooding frequency and zinc concentration for Meuse data

```
> plot(y~x, data=meuse, asp=1, col=ffreq,
+     cex=sqrt(zinc)/15)
> legend("topleft", pch=1, col=c("black","red","green"),
+     legend=c("frequent","intermediate","rare"))
> legend("bottomright", pch=1,  title="Zn mg/kg",
+     legend=zn.label <- c(100,200,500,1000,2000),
+     pt.cex=sqrt(zn.label)/15, bty="n")
```



6.3 Controlling the visual aspects of a graphic

- So far we have used the arguments `pch`, `col`, `cex`, `lty` and `lwd` to tailor the visual appearance of graphics when calling high- and low-level plotting functions.
- There are many more arguments to control the visual aspects of graphics: `adj`, `ann`, ..., `yaxt`, cf. help page of `par`.
- Default values of these arguments are queried for the active graphics device by

```
> par()
$adj
[1] 0.5

$ann
[1] TRUE

...

$ylbias
[1] 0.2
```


- Most of the arguments of `par` are effective in high-level plotting function calls.
- Many work also for low-level plotting functions.
- New default values of nearly all arguments are set for the active device by `par`:

```
> par(c("pch", "col"))
$pch
[1] 1

$col
[1] "black"

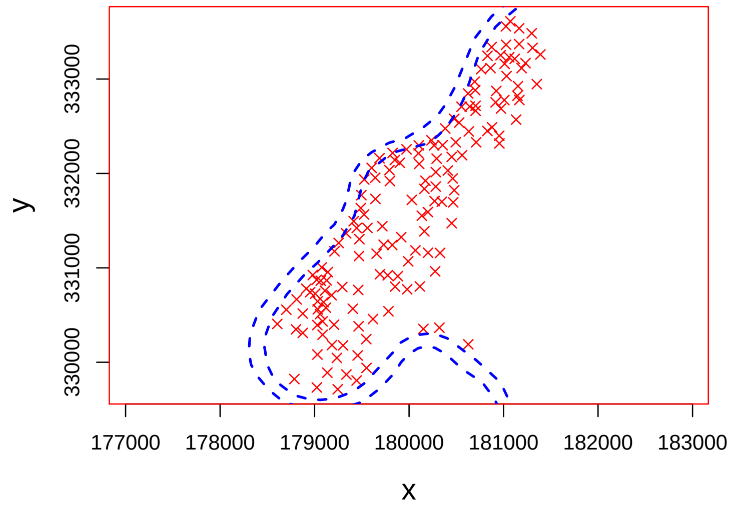
> par(pch=4, col="red")

> par(c( "pch", "col"))
$pch
[1] 4

$col
[1] "red"
```


and they remain effective as long as they are not changed

```
> plot(y~x, data=meuse, asp=1)
> lines(meuse.riv, lwd=2, col="blue")
```



Arguments and functions for the following tasks will be considered in more detail:

- placing several graphs onto a graphics device
- controlling color

For other aspects of tailoring the visual appearance of graphs (choice of text font, ...), see help page of `par`.

6.4 Placing several figures in one graphic

The arrangement of multiple plots in one graphic can be controlled by the arguments `mfrow` and `mfcol` of `par`.

Syntax:

`par(mfrow=c(`i_1`,`i_2`))` **or** `par(mfcol=c(`i_1`,`i_2`))`

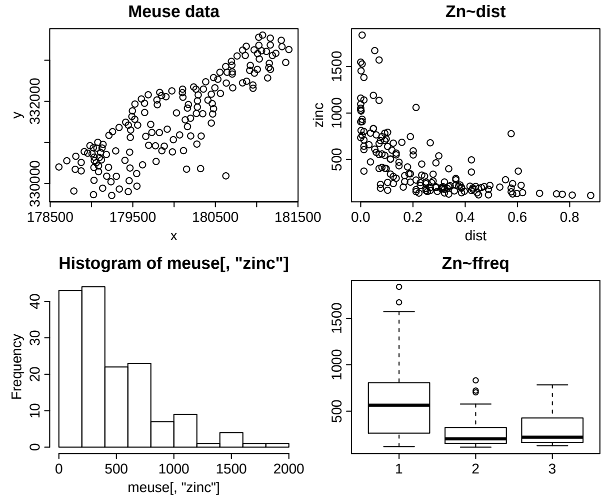
i_1, i_2: two integer scalars for the number of rows and columns into which the graphic device is split

Remarks:

- the graphics device is split into a matrix of $i_1 \times i_2$ figure regions; "rows" and "columns" have constant height and width
- successive calls of high-level plotting function populate the figure regions sequentially by plots
- sequence of plotting is either by rows (`mfrow`) or by columns (`mfcol`)
- alternatives: functions `layout` or `split.screen`

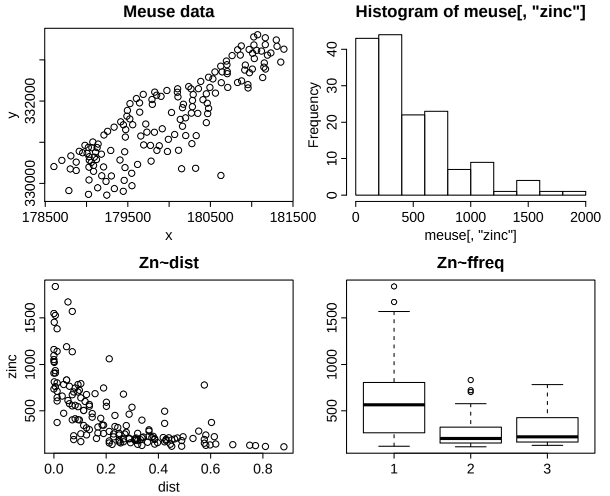
Example: multiple plots in same graphics (by rows)

```
> par(mfrow=c(2,2))
> plot(y~x, data=meuse, main="Meuse data")
> plot(zinc~dist, data=meuse, main="Zn~dist")
> hist(meuse[,"zinc"])
> boxplot(zinc~ffreq, data=meuse, main="Zn~ffreq")
```



Example: multiple plots in same graphics (by columns)

```
> par(mfcol=c(2,2))
> plot(y~x, data=meuse, main="Meuse data")
> plot(zinc~dist, data=meuse, main="Zn~dist")
> hist(meuse[,"zinc"])
> boxplot(zinc~ffreq, data=meuse, main="Zn~ffreq")
```



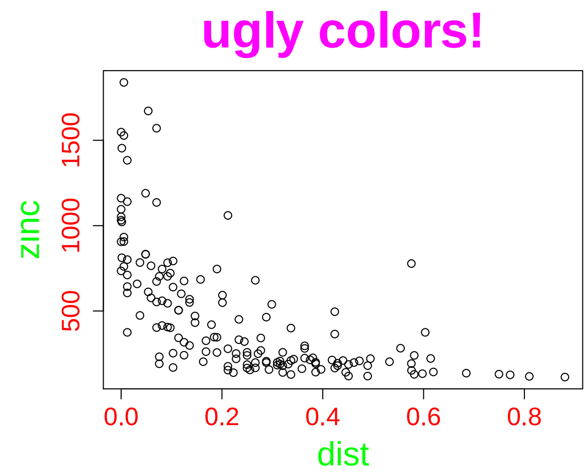
6.5 More on colors (and size)

The color (and size) of title, axes labels and tick mark labels is controlled by separate `col.`*xxx* (and `cex.`*xxx*) arguments passed to high-level functions or to `par`.

	Color	Size
title	`col.main`	`cex.main`
axes labels	`col.lab`	`cex.lab`
tick mark labels	`col.axis`	`cex.axis`

Example: settting the color and the size of text annotation

```
> par(col.main="magenta", cex.main=3,
+     col.lab="green", cex.lab=2,
+     col.axis="red", cex.axis=1.5)
> plot(zinc~dist, meuse, main="ugly colors!")
```



The background and foreground colors of a plot are queried and set by the arguments `bg` and `fg` of `par`.

Syntax:

`par(fg=`*color*`,bg=`*color*`)`

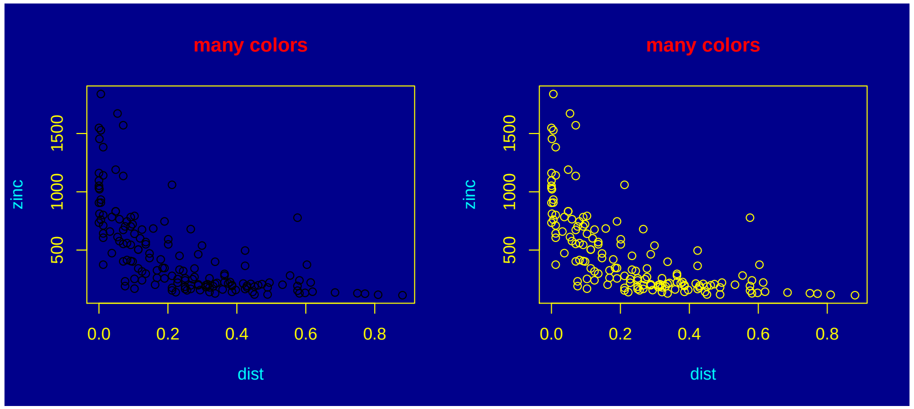
color: valid colors (integer scalar or keyword)

Remarks:

- the device region is colored by the background color; the background color can be set only by `par(bg=`*color*`)`
- `fg=`*color* can be used as argument for high-level plotting functions to set the color of the axes and the box around the plot region
- `par(fg=`*color*`)` sets in addition also the default color for points and lines plotted subsequently in the plot region
- `par(fg=`*color*`)` does not affect the color of text annotation; these colors must be set by the arguments `col.main`, `col.axis`, `col.lab`

Example: setting fore- and background colors

```
> par(mfrow=c(1,2))
> par(bg="darkblue", col.main="red", col.lab="cyan",
+     col.axis="yellow")
> plot(zinc~dist, meuse, main="many colors", fg="yellow")
> par(fg="yellow")
> plot(zinc~dist, meuse, main="many colors")
```



Colors can be either specified by integer or keywords. The color scale, i.e., the mapping of the integer numbers to particular colors, are queried and set by the function `palette`.

Syntax:

`palette(`*colorscale*`)`

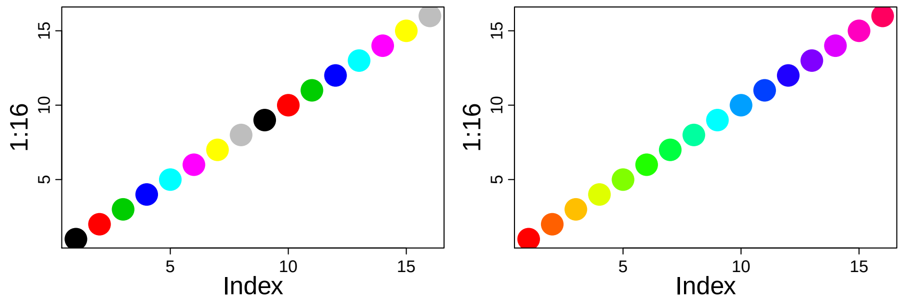
colorscale: an optional character vector with valid colors

Remarks:

- `palette()` shows the current color scale
- color vectors are preferably constructed by the built-in functions such as `rainbow`, `heat.colors`, ...(cf. `?rainbow`) or by the more flexible function `colorRampPalette` (cf. `?colorRamp`).
- `palette("default")` restores the default color scale

Example: querying and setting color scales

```
> palette()
[1] "black"  "red"    "green3" "blue"   "cyan"   ...
> par(mfrow=c(1,2))
> plot(1:16, col=1:16, pch=16, cex=3)
> palette(rainbow(16))
> plot(1:16, col=1:16, pch=16, cex=3)
```




```
> palette("default"); palette()
[1] "black"  "red"    "green3" "blue"   "cyan"   ...
```


Good R programming practice:

reset argument controlling the visual appearance of a graphics at end to the previous values,

```
> old.par <- par(mfrow = c(2,2), mgp = c(2,1,0))
> for(i in 1:4){
+   curve(sin(i * pi* x), main = paste("sin(",i,"pi x)"))
+ }
> par(old.par)
> par("mfrow")# areback to  (1, 1)
[1] 1 1
```


In this lecture you have learnt . . .

. . . how to add additional data to an existing plot by
 ⇒ functions **points** and **lines**

. . . how to draw horizontal and vertical straight lines by
 ⇒ function **abline**

. . . how to annotate points in a scatterplot by
 ⇒ function **text**

. . . how to add a legend by
 ⇒ function **legend**

. . . to query and set default values for arguments controlling the visual aspects of a graphic
 ⇒ function **par**

. . . that most of the **par** arguments can be specified “on the fly” in high-level and low-level plotting functions

. . . how to arrange several plots in one graphic
 ⇒ arguments **mfrow** , **mfcol** of function **par**

. . . how to control color
 ⇒ arguments **col.***xxx* , **fg** , **bg**
 ⇒ functions **palette** , **rainbow** , etc.

Introduction Part 2

in the second part of the Lecture “Using R ...” we

- ... deepen understanding for using functions
- ... learn about loops and control structures
- ... get to know further R building blocks (objects, classes, attributes)
- ... work with lists and apply
- ... see how to tailor the behaviour of R
- ... find out about packages and where to get help

7. Programming in R - Functions and Control Structures

In this chapter you will learn about . . .

- . . . How to write a function (repetition from part I)
- . . . Error messages, debugging etc
- . . . Control structures, i.e. loops, if–else, etc.

7.1 Writing Functions

Syntax: `fname <- function( arg(s) ) { R statements }`

A simple function: Get the maximum value of a vector and its index.

```
> f.maxi <- function(data) {
+   mx <- max(data, na.rm=TRUE) # get max element
+   i <- match(mx, data)        # position of max in data
+   c(max=mx, pos=i)            # result of function
+ }
```

Output of f.maxi is a named vector. By default, the the result of the last evaluated R statement is returned by a function call. Use of an explicit `return()` statement is optional.

```
> f.maxi(c(3,4,78,2))
max pos
 78   3
```

(Note: R provides the function `which.max`)

Optional arguments and argument default values

Many functions have optional arguments with default values.

For instance look at function code of `hist()` or `?hist` :

```
function (x, breaks = "Sturges", freq = NULL, probability = !freq,
    include.lowest = TRUE, right = TRUE, density = NULL, angle = 45,
    col = NULL, border = NULL, main = paste("Histogram of", xname),
    xlim = range(breaks), ylim = NULL, xlab = xname, ylab, axes = TRUE,
    plot = TRUE, labels = FALSE, nclass = NULL, warn.unused = TRUE,
    ...)
{
```

Example Use optional argument `my.names` to specify names of result vector of `f.maxi`

```
> f.maxi.names <- function(data,my.names=c("max","pos")){
+   mx <- max(data, na.rm=TRUE) # get max element
+   i <- match(mx, data)        # position of max in data
+   res <- c(mx, i)             # result of function
+   names(res) <- my.names      # naming of result
+   res                         # or return(res)
+ }
```

Default values are used if actual argument `my.names` not specified

```
> f.maxi.names(c(3,4,78,2))
max pos
 78   3
```

but may be over-written by specifying values for `my.names`

```
> f.maxi.names(c(3,4,78,2), my.names=c("Maximum","Index"))
Maximum   Index
     78       3
```


Querying if a formal argument has be specified

Use `missing()` to query if a formal argument has been specified.

Example

```
> f.maxi.names2 <- function(data,my.names=c("max","pos")){
+   cat("'my.names' missing?", missing(my.names), "\n")
+   mx <- max(data, na.rm=TRUE)
+   res <- c(mx, match(mx, data))
+   names(res) <- my.names;  res  }

> f.maxi.names2(c(3,4,78,2))
'my.names' missing? TRUE
max pos
 78   3

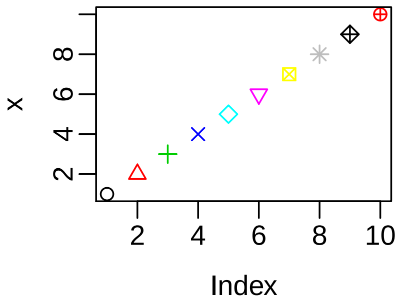
> f.maxi.names2(c(3,4,78,2), my.names=c("Maximum","Index"))
'my.names' missing? FALSE
Maximum   Index
     78       3
```


Unspecified list of arguments of a function

A function may accept arbitrary arguments if the function definition contains `...` as formal argument.

Example:

```
> myplot <- function(x, y, ...){
+   plot(x, ...)
+ }
> myplot(1:10, col=1:10, pch=1:10)
```



Printing the definition of a function

- Type the name of the function (without parentheses) for its definition

```
> f.maxi
function(data) {
  mx <- max(data, na.rm=TRUE) # get max element
  i <- match(mx, data)        # position of max in data
  c(max=mx, pos=i)            # result of function
}
```

- Of course, this works for all “built-in” R functions

```
> sd
function (x, na.rm = FALSE)
sqrt(var(if (is.vector(x)) x else as.double(x), na.rm = na.rm))
<bytecode: 0x37249f0>
<environment: namespace:stats>
```

- or to see only its formal arguments type `str(`*fname*`)`

```
> str(sd)
function (x, na.rm = FALSE)
```

- Function arguments and their defaults are also shown on `help(.)` page, in section `Usage:`.
  Try `?sd`

Summary: R functions

- with several argument often have *defaults*,
- < *argname* > = < *default* >
- "visible" from the help page's `Usage:` section or `str()`.
- Functions *return* the last evaluated expression, typically, the last line.
- `return()` is hence optional and not often used.
- look at the function definition by just (auto-) `print()` ing it

7.2 Error Handling

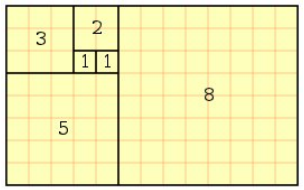
- Error messages are often helpful ...
  sometimes, you have no clue – mostly, if they occur in a function that was called by a function ...
- Show the "stack" of function calls:
  `> traceback()`
- Ask an experienced user ...
- If you write your own functions:
  - use `print` statements (if simple code)
  - `?browser`
  - `?debug`
  - `options(error=recover)` calls `browser` when an error occurs.
  - `browser()` as a statement in the function: stops execution and lets you inspect all variables.

7.3 Control Structures: Loops

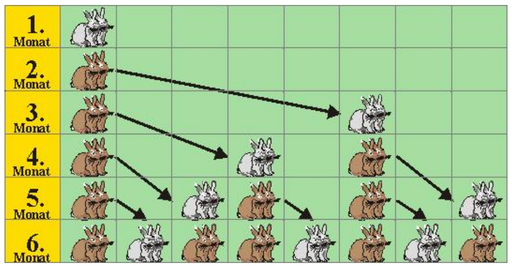
Loops are basic for programming. Most important one: `for`

Syntax: `for ( i in ... ){ commands}`

Example: The Fibonacci series. Illustration of the first 6 elements:



and applications:



Example: Fibonacci Series

Goal: Calculate the first twelve elements of the Fibonacci series.

```
> fib <- c(1,1)
> for(i in 1:10)
+      fib <- c(fib, fib[i]+fib[i+1])
> fib
 [1]   1   1   2   3   5   8  13  21  34  55  89 144
```

```
> fib <- c(1,1)
> for(i in 1:6){
+      fib <- c(fib, fib[i]+fib[i+1])
+      print(fib)
+    }
[1] 1 1 2
[1] 1 1 2 3
[1] 1 1 2 3 5
[1] 1 1 2 3 5 8
[1]  1  1  2  3  5  8 13
[1]  1  1  2  3  5  8 13 21
```

Note

Instead of `for` loops, you can (and should!) often use more elegant and efficient operations,

- e.g., instead of

  ```
  > n <- length(x); y <- x
  > for(i in 1:n)
  +     y[i] <- x[i] * sin(pi * x[i])
  ```

  use simply

  ```
  > Y <- x * sin(pi * x)
  ```

  Of course, that's equivalent:

  ```
  > identical(Y, y)
  [1] TRUE
  ```

- In more complicated cases, it is often advisable to `apply()` functions instead of `for(.){...}`, see next week!

7.4 Control Structures: if – else

- Conditional evaluation: `if(.){...}  [ else{...} ]`
  Syntax:

  `if(`*logical*`)` *A* or
  `if(`*logical*`)` A_1 `else` A_2

  E.g., For the Fibonacci construction loop,

  ```
  > fib <- c(1,1)
  > for( i in 1: 100 ) {
  +     fib <- c(fib, fib[i]+fib[i+1])
  +     if ( fib[i+2] > 5000 ) break
  + }
  > fib
   [1]    1    1    2    3    5    8   13   21   34   55   89
  [14]  377  610  987 1597 2584 4181 6765
  ```

- with optional `else`

  ```
  >  if(sum(y) > 0) log(sum(y)) else "negative sum"
  [1] "negative sum"
  ```


Digression: other loop constructs – break

Use `break` for forced leaving of a loop

```
> plot(1:10)
> ## "left-click" to read coordinates and
> ## plot further points, stop by "right-clicking"
> for(i in 1:10000) {
+   loc <- locator(1,type="l")
+   if(length(loc) < 1)
+     break          ## "right-clicking" leaves loop
+   points(loc, pch=19)
+ }
```


Control Structures: if – else (continued)

`if(cond) A` *always* returns a value:

```
> u <- 1
> x1 <- if(u^2 == u) "are the same" ; x1
[1] "are the same"
> u <- 2
> x2 <- if(u^2 == u) "are the same" ; x2
NULL
```

$\boxed{\texttt{if(}\textit{cond}\texttt{) A} \text{ when } \textit{cond} \text{ is false, has value } \texttt{NULL}}$

What is "NULL" ?? Not the same as '0':

```
>  length(NULL) ## has length zero
[1] 0
> is.null(NULL) ## query whether an output is NULL
[1] TRUE
> c(2,NULL,pi) ## does not show up in vectors
[1] 2.0000 3.1416
```

Examples

- A (simplistic!) example of computing “significance stars” from P-values:

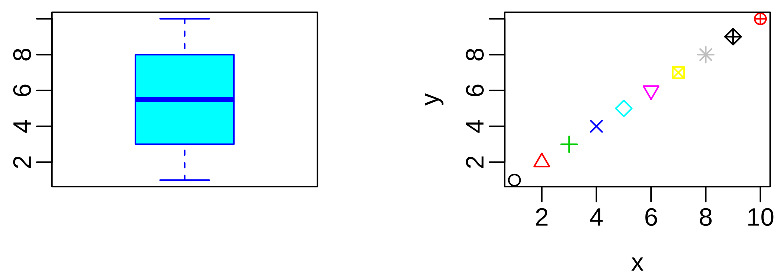
```
> myStar <- function(x) { if(x < .01) "**" else
+                         if(x < .05) "*" else "" }
> myStar(0.024)
[1] "*"
> myStar(0.2)
[1] ""
> myStar(0.002)
[1] "**"
```


-
```
> tst3 <- function(x) {
+    if(x %% 3 == 0) paste("HIT:", x) else format(x %% 3)
+ }
> c(tst3(17), tst3(27))
[1] "2"        "HIT: 27"
```

-
```
> tst4 <- function(x) {
+     if(x < -2) "pretty negative"
+     else if(x < 1) "close to zero"
+     else if(x < 3) "in [1, 3)" else "large"
+ }
     x    tst4(x)
[1,] "-5" "pretty negative"
[2,] "-1" "close to zero"
[3,] "0"  "close to zero"
[4,] "1"  "in [1, 3)"
[5,] "2"  "in [1, 3)"
[6,] "3"  "large"
[7,] "4"  "large"
```


Another example using `missing` and `...`

```
> myplot <- function(x, y=NA, ...){
+   if( missing(y) ) boxplot(x, ...)
+   else plot(x, y, ...)
+ }
> par(mfcol=c(1, 2))
> myplot(1:10, border="blue", col="cyan")
> myplot(1:10, 1:10, col=1:10, pch=1:10)
```



Using R for Data Analysis and Graphics

8. Objects in R

In this chapter you will learn about . . .

. . . different R objects and classes

. . . handling Dates and Times

. . . manipulating strings

8.1 R Objects

The basic building blocks of R are called "objects". – They come in "classes":

- numeric, character, factor ... one-dim. sequence of numbers, strings, ... called *atomic*[9] vectors
- matrix — two dimensional array of numbers, strings, ...
- array — (1–, 2–, 3–, ...)dimensional; 2-dim. array =: matrix.
- data.frame — two dimensional, (numbers, "strings", factors, ...)
- formula — specifying (regression, plot, ...) "model"
- function — also an object!
- list — very *general* collection of objects, → see below
- call, ... — and more

Determine class with `class()`.

[9] see help page `?is.atomic` for more

`array` — *k*-dimensional matrix

Matrices are 2-dimensional, an `array` can be k-dimensional ($k \geq 1$). E.g., 3-dimensional, a "stack of matrices":

```
> a <- array(1:30, dim=c(3,5,2))
> a
, , 1

     [,1] [,2] [,3] [,4] [,5]
[1,]    1    4    7   10   13
[2,]    2    5    8   11   14
[3,]    3    6    9   12   15

, , 2

     [,1] [,2] [,3] [,4] [,5]
[1,]   16   19   22   25   28
[2,]   17   20   23   26   29
[3,]   18   21   24   27   30
```


`array` — (2)

```
> a <- array(1:30, dim=c(3,5,2))
> class(a)
[1] "array"
```

Query the class of an object with `is.`..., result is `logical`:

```
> is.array(a)
[1] TRUE
> dim(a[ 1, , ])    # the first slice of a[]
[1] 5 2
> m <- a[ , 2, ] ; m
     [,1] [,2]
[1,]    4   19
[2,]    5   20
[3,]    6   21
> is.matrix(m)      # a "slice" of a 3-d array is a matrix
[1] TRUE
```


There are specific functions to examine the kind of an object.In particular the "inner" structure of an object, is available by `str()` :

```
> str(d.sport)
'data.frame': 15 obs. of  7 variables:
 $ weit   : num  7.57 8.07 7.6 7.77 7.48 7.88 7.64 7.61 7.2..
 $ kugel  : num  15.7 13.6 15.8 15.3 16.3 ...
 $ hoch   : int  207 204 198 204 198 201 195 213 207 204 ...
 $ disc   : num  48.8 45 46.3 49.8 49.6 ...
 $ stab   : int  500 480 470 510 500 540 540 520 470 470 ...
 $ speer  : num  66.9 66.9 70.2 65.7 57.7 ...
 $ punkte: int  8824 8706 8664 8644 8613 8543 8422 8318 83..
> class(d.sport[,"weit"])
[1] "numeric"
> str(m)
 int [1:3, 1:2] 4 5 6 19 20 21
> str(a)
 int [1:3, 1:5, 1:2] 1 2 3 4 5 6 7 8 9 10 ...
```

8.2 Apply on Dataframes and Arrays

Loops can and should be avoided in many cases!

- Apply a function to each column (or row) of a data.frame or matrix or array:

```
> apply(d.sport, 2, mean)
     weit     kugel      hoch      disc      stab     speer
   7.5967   15.1987  202.0000   46.3760  498.0000   61.9947
   punkte
8444.6667
```

Second argument: `1` for "summary" of rows, `2` for columns, `3` for 3rd dimension, ...

- If the function needs more arguments, they are provided as additional arguments:

```
> apply(d.sport, 2, mean, trim=0.3)
     weit     kugel      hoch      disc      stab     speer
   7.5914   15.1871  201.8571   46.4171  495.7143   63.0000
   punkte
8397.8571
```

```
> apply(a,3,mean)
[1]  8 23
```

More on `apply` next week.

8.3 Factors (repeated from part I)

Groups, or categorial variables are represented by factors.
Examples: IDs of measurement stations, types of species, types of treatment, etc.

To produce a factor variable:

- use `c()`, `rep()`, `seq()` to define a numeric or character vector
- and then the function `as.factor()`
- Note: internally factors use integers as grouping-ID, but levels can be defined, to label the groups.

An example: Suppose the athletes listed in `d.sport` belong to 3 teams:

```
> teamnum <- rep(1:3,each=5)
```

```
> d.sport[,"team"] <- as.factor(teamnum)
> str(d.sport)
'data.frame': 15 obs. of  8 variables:
 $ weit  : num  7.57 8.07 7.6 7.77 7.48 7.88 7.64 7.61 7.2..
 $ kugel : num  15.7 13.6 15.8 15.3 16.3 ...
 $ hoch  : int  207 204 198 204 198 201 195 213 207 204 ...
 $ disc  : num  48.8 45 46.3 49.8 49.6 ...
 $ stab  : int  500 480 470 510 500 540 540 520 470 470 ...
 $ speer : num  66.9 66.9 70.2 65.7 57.7 ...
 $ punkte: int  8824 8706 8664 8644 8613 8543 8422 8318 83..
 $ team  : Factor w/ 3 levels "1","2","3": 1 1 1 1 1 2 2 2..
```

```
> class(d.sport[,"team"])
[1] "factor"
```

```
> levels(d.sport[,"team"])
[1] "1" "2" "3"

> nlevels(d.sport[,"team"])
[1] 3

> levels(d.sport[,"team"]) <-
+     c("Zurich","New York","Tokyo")

> head(d.sport,n=10)
           weit kugel hoch  disc stab speer punkte     team
OBRIEN     7.57 15.66  207 48.78  500 66.90   8824   Zurich
BUSEMANN   8.07 13.60  204 45.04  480 66.86   8706   Zurich
DVORAK     7.60 15.82  198 46.28  470 70.16   8664   Zurich
FRITZ      7.77 15.31  204 49.84  510 65.70   8644   Zurich
HAMALAINEN 7.48 16.32  198 49.62  500 57.66   8613   Zurich
NOOL       7.88 14.01  201 42.98  540 65.48   8543 New York
ZMELIK     7.64 13.53  195 43.44  540 67.20   8422 New York
GANIYEV    7.61 14.71  213 44.86  520 53.70   8318 New York
PENALVER   7.27 16.91  207 48.92  470 57.08   8307 New York
HUFFINS    7.49 15.57  204 48.72  470 60.62   8300 New York
```

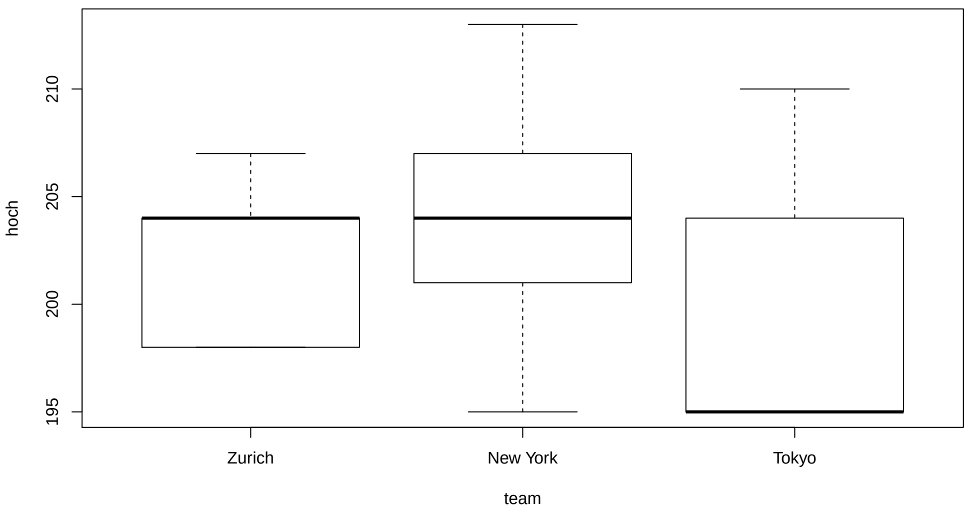


```
> #How many cases per factor level?
> table(d.sport[,c("team")])
  Zurich New York    Tokyo
       5        5        5

> #aggregate according to factor
> aggregate(punkte~team,d.sport,sum)
      team punkte
1   Zurich  43451
2 New York  41890
3    Tokyo  41329
```


Functions handle factors differently to numeric variables. Example: plot() generates boxplot:

```
> plot(hoch~team,d.sport)
```



Note: In statistical analyses categorical variables MUST be coded as factors to produce correct results (e.g. in analysis of variance or for regression).
$\longrightarrow$ ALWAYS check your data (by `str()`) before starting an analysis.

8.4 Dates and Times

Dates and Times are also R objects with specific classes. Get the System Date

```
> (dd <- Sys.Date())
[1] "2014-02-19"
> class(dd)
[1] "Date"
```

and System Time

```
> (tt <- Sys.time())
[1] "2014-02-19 11:12:45 CET"
> str(tt)  #in seconds
 POSIXct[1:1], format: "2014-02-19 11:12:45"
> class(tt)
[1] "POSIXct" "POSIXt"
```


Classes "Date", "POSIXlt" and "POSIXct" represent calendar dates and times (to the nearest second).

Class "POSIXct" represents the (signed) number of seconds since the beginning of 1970 (in the UTC timezone) as a numeric vector.

Class "POSIXlt" is a named list of vectors representing sec, min, hour, mday, mon, year, . . .

More information on ?DateTimeClasses.

Conversion between time zones:

```
> #Note: need to supply suitable file path
> # "/usr/share/zoneinfo/zone.tab" first
> as.POSIXlt(Sys.time(), "GMT")
[1] "2014-02-19 10:12:45 GMT"
> #what time in time zone of Seattle?
> as.POSIXlt(Sys.time(), , tz = "PST8PDT")
[1] "2014-02-19 02:12:45 PST"
> #and in Denver?
> as.POSIXlt(Sys.time(), , tz = "America/Denver")
[1] "2014-02-19 03:12:45 MST"
```


Special operations and functions are defined for Dates and Times. See ?Ops.Date or ?Ops.POSIXt. Some examples:

```
> dd +20 # 20 days from now
[1] "2014-03-11"
> tt + 3600 # an hour later
[1] "2014-02-19 12:12:45 CET"
> #How many days to christmas?
> difftime(dd,"2013-12-25 8:00")
Time difference of 55.708 days
> #convert character to Date/Time
> (xx <- strptime("2100-12-25",format="%Y-%m-%d"))
[1] "2100-12-25"
> #Which day of the week is Christmas 2100?
> weekdays(xx)
[1] "Saturday"
```

8.5 Manipulating strings

Often string manipulation is necessary or desireable. A list from Uwe Ligges's book[10] below shows some of the available functions. Look at the respective help pages for more information. A few examples follow next.

Function / Operator	Description
`cat()`	print text in console or to file
`deparse()`	convert an *expression* to a string
`formatC`	very general formatting possibilities
`grep()`	search for (sub-)string in vectors
`match(), pmatch()`	search for string matches
`nchar()`	get number of characters in a string
`parse()`	convert to an *expression*
`paste()`	paste several strings together
`sub(), gsub()`	replace (parts of) strings
`substring()`	extract sub-strings
`toupper(),tolower()`	change to upper or lower case letters
`strsplit()`	split strings, result is a list

[10] Uwe Ligges: Programmieren in R, Springer.

Examples String Manipulation

Combine numeric and text output for messages or to write to files:

```
> pp <- round(2*pi,2)
> cat("Two times Pi is:", pp, "\n", sep = "\t")
Two times Pi is: 6.28
> cat("Two times Pi is:", pp, "\n", sep = "\t",
+     file = "myOutputMessage.txt")
```

Useful string manipulations:

```
> nam  <- "Peter Pan" # create string
> nchar(nam) # how many letters
[1] 9
> ## substitute parts of strings (useful for Umlauts etc):
> (nam2 <- gsub("Peter","Pete",nam) )
[1] "Pete Pan"
> toupper(nam2) # convert to upper case
[1] "PETE PAN"
```


Examples String Manipulation (cont'd)

Create numbered filenames:

```
> filenames <- paste("File", 1:3, ".txt", sep = "")
```

Exchange a matching string with a replacement. The original ist NOT overwritten. Note the "protection" (escape) `"\\"` for special characters such as "."

```
> sub("File","Datei",filenames)
[1] "Datei1.txt" "Datei2.txt" "Datei3.txt"
> sub("\\.","_",filenames)
[1] "File1_txt" "File2_txt" "File3_txt"
```

Find which entries in a vector match a given string:

```
> grep("ile",filenames)
[1] 1 2 3
> grep("3",filenames)
[1] 3
> filenames[grep("3",filenames)]
[1] "File3.txt"
```


Using R for Data Analysis and Graphics

9. Lists and Apply

In this chapter you will learn about . . .

. . . how to work with lists

. . . the efficient use of `apply`

9.1 Lists

Objects of any kind can be collected into a list:

```
> v <- c(Hans=2, Fritz=-1, Elsa= 9, Trudi=0.4, Olga=100.)
> list(v, you="nice")
[[1]]
 Hans Fritz  Elsa Trudi  Olga
  2.0  -1.0   9.0   0.4 100.0

$you
[1] "nice"
```

As with `c(...)` , all arguments are collected, names can be given to the components.

Lists are an important (additional) `class` of objects, since most statistical functions produce a list that collects the results.

```
> hi.k <- hist(d.sport[,"kugel"], plot=FALSE)
> hi.k
$breaks
[1] 13.5 14.0 14.5 15.0 15.5 16.0 16.5 17.0

$counts
[1] 2 1 4 1 4 1 2

$density
[1] 0.26667 0.13333 0.53333 0.13333 0.53333 0.13333 0.26667

$mids
[1] 13.75 14.25 14.75 15.25 15.75 16.25 16.75

$xname
[1] "d.sport[, \"kugel\"]"

$equidist
[1] TRUE

attr(,"class")
```


- Get a sublist of the list: `[ ]`

  ```
  > hi.k[2:3]
  $counts
  [1] 2 1 4 1 4 1 2

  $density
  [1] 0.26667 0.13333 0.53333 0.13333 0.53333 0.13333 0.26667
  ```

  or `hi.k[c("breaks","intensities")]`

  Note: `hi.k["counts"]` is a list with one component.

- Get a component: `[[ ]]`

  ```
  > hi.k[[2]]
  [1] 2 1 4 1 4 1 2
  > identical(hi.k[[2]], hi.k[["counts"]])
  [1] TRUE
  ```

  or also `hi.k$counts` . These components are all vectors.

- Hint: A data.frame is a list with additional attributes.
  $\longrightarrow$ Single columns (variables) can be selected by `$` :

  ```
  > k <- d.sport$kugel
  > ## select elements from it:
  > d.sport$kugel[4:6]     # but preferrably
  [1] 15.31 16.32 14.01
  > d.sport[4:6, "kugel"] # treat it like a matrix
  [1] 15.31 16.32 14.01
  ```

- Make a list of subsets of a vector:

```
> split(1:7, c(1, 1, 2, 3, 3, 2, 1))
$`1`
[1] 1 2 7

$`2`
[1] 3 6

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

- `unlist` concatenates all elements of all components into a single vector.

```
> unlist(hi.k[1:2])
breaks1 breaks2 breaks3 breaks4 breaks5 breaks6 breaks7
   13.5    14.0    14.5    15.0    15.5    16.0    16.5
breaks8 counts1 counts2 counts3 counts4 counts5 counts6
   17.0     2.0     1.0     4.0     1.0     4.0     1.0
counts7
    2.0
```


Examples String Manipulation (cont'd)

Create numbered filenames:

```
> filenames <- paste("File", 1:3, ".txt", sep = "")
```

Split the string at specified separator; Note the "protection" (escape) `"\\"` for special characters such as "."

```
> unlist(strsplit(filenames[1],"\\."))
[1] "File1" "txt"
```

Personalize file names - for user name "Pete Pan", see last lecture:

```
> (nn <- unlist(strsplit(nam2, " ")))# split string at " "
[1] "Pete" "Pan"
> # get first letters as new string:
> (nn2 <- paste(sapply(nn, function(x) substring(x,1,1)),
+                collapse = ""))
[1] "PP"
> (myfiles <- paste(unlist(strsplit(filenames,".txt")),
+                       "_", nn2, ".txt", sep=""))
[1] "File1_PP.txt" "File2_PP.txt" "File3_PP.txt"
```


Functions for vectorized Programming

Function / Operator	Description
`%*%`	Vector product / matrix multiplication
`%x%, kronecker(X,Y, FUN="*")`	Kronecker product; the latter applies an arbitrary bivariate function `FUN`
`%o%, outer(X,Y, FUN="*")`	"outer" product; the latter applies any `FUN()`.
`sum(v),prod(v),all(L),...`	Sum, product, ... of all elements
`colSums(),rowSums()`	Fast column / row sums
`colMeans(),rowMeans()`	Fast column / row means
`apply()`	column- or row-wise application of function on matrices and arrays
`lapply()`	elementwise application of function on lists, data frames, vectors
`sapply()`	**s**implified `lapply`: returns simple vector, matrix, ... (if possible)
`tapply()`	**t**able producing `*apply`, grouped by factor(s)
`vapply()`	(more robust, slightly faster) version of `sapply`
`rapply()`	**r**ecursive version of `lapply`
`mapply()`	**m**ultivariate version of `lapply`

List–Apply: `lapply()` — *the* most important one

- Compute the list mean for each list element

```
> # generate list
> x <- list(a = 1:10, beta = exp(-3:3),
+           logic = c(TRUE,FALSE,FALSE,TRUE))
> # list with mean of each list element
> lapply(x,mean)
$a
[1] 5.5

$beta
[1] 4.5351

$logic
[1] 0.5
```

List - Apply sapply()

- sapply = [S]implified lapply
  The result is unlist()ed into a vector, named and possibly reshaped into a matrix[11].

```
> sapply(x,mean)# a named numeric vector
     a   beta  logic
5.5000 4.5351 0.5000
```

[11] or higher array, with argument simplify = "array"

- Median and quartiles for each list element

```
> lapply(x, quantile, probs = 1:3/4)
$a
 25%  50%  75%
3.25 5.50 7.75

$beta
    25%     50%     75%
0.25161 1.00000 5.05367

$logic
25% 50% 75%
0.0 0.5 1.0
```

```
> sapply(x, quantile)
         a      beta logic
0%    1.00  0.049787   0.0
25%   3.25  0.251607   0.0
50%   5.50  1.000000   0.5
75%   7.75  5.053669   1.0
100% 10.00 20.085537   1.0
```


- Example with linear regressions (“Anscombe” data)

```
> data(anscombe)                      # Load the data
> #small data set with 4 target variables and covariate
> ans.reg <- vector(4, mode = "list")# empty list
> # Store 4 regressions (y_i vs x_i) in list:
> for(i in 1:4){
+   form <- as.formula(paste("y",i," ~ x",i, sep=""))
+   ans.reg[[i]] <- lm(form, data = anscombe)
+ }
> lapply(ans.reg, coef)# a list, of length-2 vectors
> sapply(ans.reg, coef)# simplified into 2 x 4 matrix
                [,1]    [,2]     [,3]     [,4]
(Intercept) 3.00009 3.0009 3.00245 3.00173
x1          0.50009 0.5000 0.49973 0.49991
```


Digression: Random Numbers

- The *apply() functions are particularly useful for large data sets and with simulation results, often generated using random numbers
- “Random” numbers are generated by a deterministic function. Examples are runif(), rnorm()
- Nevertheless, two identical calls give different results.

```
> runif(4)
[1] 0.864278 0.421576 0.399071 0.081312
> runif(4)
[1] 0.121530 0.993147 0.080135 0.793373
```

- For reproducibility, e.g. in simulation studies, use ...

```
> set.seed(27); runif(1)
[1] 0.97175
> set.seed(27); runif(1)
[1] 0.97175
```

Functions in sapply, lapply

Can use “anonymous” functions directly inside apply - functions.

Example: Retrieve i-th col/row of all matrices that are elements of a list

```
> set.seed(1234)# define list of matrices
> sl <- list(A= matrix(rnorm(25,10,1),ncol=5),
+             B= matrix(runif(20),ncol=5))
> #retrieve 3rd column from both matrices
> sapply(sl,function(x){x[,3]})
$A
[1]  9.5228  9.0016  9.2237 10.0645 10.9595

$B
[1] 0.174650 0.848392 0.864834 0.041857
```

Note: `sapply` creates different types of objects depending on output. Try out

```
> class(sapply(sl, function(x) x[2,]) ) # a matrix
> class(sapply(sl, function(x) x[,3]) ) # a list, because
>             # matrices in sl do not have same no of rows
```


tapply – a “ragged” array

Summaries over groups of data:

```
> n <- 17
> fac <- factor(rep(1:3, length = n), levels = 1:4)
> fac # last level not present:
 [1] 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2
Levels: 1 2 3 4
> table(fac)
fac
1 2 3 4
6 6 5 0
> tapply(1:n, fac, sum)
 1  2  3  4
51 57 45 NA
```


`tapply()` simplifies the result by default, *when* possible,

```
> tapply(1:n, fac, sum, simplify = FALSE)  # simplify=FALSE
$`1`
[1] 51

$`2`
[1] 57

$`3`
[1] 45

$`4`
NULL
> tapply(1:n, fac, quantile)  # simplification not possible
$`1`
   0%   25%   50%   75%  100%
 1.00  4.75  8.50 12.25 16.00

$`2`
   0%   25%   50%   75%  100%
 2.00  5.75  9.50 13.25 17.00

$`3`
   0%   25%   50%   75% 100%
```


tapply — by()

`by(data,index,fun,...)`

Summaries by groups of data, uses `tapply()` internally!

```
> # help(warpbrakes)
> # split by tension-levels
> by(warpbreaks[, 1:2], warpbreaks[,"tension"], summary)
> # split by tension-and-wool levels
> by(warpbreaks[, 1],   warpbreaks[, -1],       summary)
warpbreaks[, "tension"]: L
     breaks      wool
 Min.   :14.0   A:9
 1st Qu.:26.0   B:9
 Median :29.5
 Mean   :36.4
 3rd Qu.:49.2
 Max.   :70.0
----------------------------------------
warpbreaks[, "tension"]: M
     breaks      wool
 Min.   :12.0   A:9
 1st Qu.:18.2   B:9
 Median :27.0
 Mean   :26.4
 3rd Qu.:33.8
 Max.   :42.0
```

tapply — aggregate()

Summaries over groups of data:

```
> # help(sleep)
> aggregate(sleep[,"extra"],
+           list(sleep[,"group"]), median)
  Group.1    x
1       1 0.35
2       2 1.75
```

Result is a `data.frame`.
Many groups ⟶ Analyze summaries using new `data.frame`!

- Conceptually similar to `by()` (and hence `tapply()`).
Compare output of `by()` above to

```
> aggregate(warpbreaks[,1:2],
+           list(Tension=warpbreaks[,"tension"]),
+           summary)
```


Using R for Data Analysis and Graphics

10. More R: Objects, Methods,...

In this chapter you will learn ...

... more on objects, their classes, attributes and (S3) methods

... more on functions

... using `options()` (and `par()`)

10.1 R Objects - this slide repeated from above

Slide from **??**: The basic building blocks of R are called "objects". – They come in "classes":

- numeric, character, factor ... one-dim. sequence of numbers, strings, ... called *atomic*[12] vectors
- matrix two dimensional array of numbers, strings, ...
- array (1–, 2–, 3–, ...)dimensional; 2-dim. array =: matrix.
- data.frame two dimensional, (numbers, "strings", factors, ...)
- formula specifying (regression, plot, ...) "model"
- function also an object!
- list very *general* collection of objects, → see below
- call, ... and more

Determine class with `class()`.

[12] see help page `?is.atomic` for more

Example

```
> class(d.sport)
[1] "data.frame"
```

This information and more, namely the "inner" structure of an object, is available by `str()`

```
> str(d.sport)
'data.frame': 15 obs. of  7 variables:
 $ weit  : num  7.57 8.07 7.6 7.77 7.48 7.88 7.64 7.61 7.2..
 $ kugel : num  15.7 13.6 15.8 15.3 16.3 ...
 $ hoch  : int  207 204 198 204 198 201 195 213 207 204 ...
 $ disc  : num  48.8 45 46.3 49.8 49.6 ...
 $ stab  : int  500 480 470 510 500 540 540 520 470 470 ...
 $ speer : num  66.9 66.9 70.2 65.7 57.7 ...
 $ punkte: int  8824 8706 8664 8644 8613 8543 8422 8318 83..
```

10.2 Object Oriented Programming

- Many functions do rather different things in dependence of the `class` of their first argument.
- Most prominently: `print()` or `plot()` are such "generic" functions.
- Generic functions examine the class of their first argument and then call a "method" (function) accordingly.
- Example:

```
> plot(speer~kugel, data=d.sport)
```

calls the `formula` method of the `plot` generic function, as `class(speer~kugel)` is of class `"formula"`

Generic Functions

- The most basic generic function is print().
- Example:

```
> class( r.t <- wilcox.test(extra~group, data=sleep) )
[1] "htest"
> r.t
Wilcoxon rank sum test with continuity correction

data:  extra by group
W = 25.5, p-value = 0.06933
alternative hypothesis: true location shift is not equal to 0
```

`> r.t` (or `print(r.t)`) calls the `htest` method of the `print` generic function, as `class(r.t)` is of class `"htest"`

- **Note**: The `print()` function is called *whenever* no explicit function is called: $\Rightarrow$ R is "auto – printing".

Generic Functions — Finding all methods

- Use `methods(`*gnrc*`)` to find all available methods for a generic function *gnrc*
- Example: Find all available methods for the generic function `print()` [13]

```
> methods(print)
 [1] "print.acf"       "print.anova"     "print.aov"
 [4] "print.aovlist"   "print.ar"        "print.Arima"
 [7] "print.arima0"    "print.AsIs"      "print.aspell"
......
> length(methods(print)) # ** MANY **
[1] 181
```

[13] strictly, the *"S3 methods"* only. S3 is the first "informal" object system in S and R; the *"formal"* object system, *"S4"*, defines classes and methods formally, via `setClass()`, `setMethod()` etc; and lists methods via `showMethods()` instead of `methods()`

- Find all available methods for generic function `plot()` :

```
> methods(plot)
 [1] plot.acf*             plot.data.frame*
 [3] plot.decomposed.ts*   plot.default
 [5] plot.dendrogram*      plot.density
......
> length(methods(plot)) # ** MANY **
[1] 28
```

- From these, we have already used *implicitly*
  - `plot.default`, the default method,
  - `plot.formula`, in `plot(y~x, ...)`,
  - `plot.factor`, (which gave boxplots),
  - `plot.data.frame`, giving a scatter plot *matrix*, as with `pairs()`,

etc

Generic Functions and Methods— Finding information

- Suppose, we want to learn what arguments the function `mean()` accepts;
- from the above we use

```
> str(mean)
function (x, ...)
> mean
function (x, ...)
UseMethod("mean")
<bytecode: 0x33bd978>
<environment: namespace:base>
```

  which seems not very helpful!
- Let's see if `mean()` is a generic function

```
> methods(mean)
[1] mean.Date     mean.default  mean.difftime mean.POSIXct
[5] mean.POSIXlt
```


- ok, `mean.default()` seems to be what we are looking for, now let's explore it

```
> str(mean.default)
function (x, trim = 0, na.rm = FALSE, ...)
> mean.default
function (x, trim = 0, na.rm = FALSE, ...)
{
    if (!is.numeric(x) && !is.complex(x) && !is.logical(x)) {
 .......
<bytecode: 0x306c8b8>
<environment: namespace:base>
```

- Of course, we could have looked at help page: `?mean`
- help pages of generic functions list in section Usage first the argument(s) of the generic and then under `## Default S3 method:` those of the default method

Generic Functions — Finding all methods for a class

- Use `methods(class="`*cls*`")` to find all the available methods for a given class *cls*
- Example: Find all available methods for the class `matrix`

```
> methods(class="matrix")
 [1] plot.acf*            plot.data.frame*
 [3] plot.decomposed.ts*  plot.default
 [5] plot.dendrogram*     plot.density
 .......
   Non-visible functions are asterisked
> length(methods(class="matrix")) # ** MANY **
[1] 14
```


- Apart from basic classes like `matrix`, `formula`, `list`, etc, many functions, notably those fitting a statistical model, return their result as a specific `class`.
- Example: Linear regression (⟶ function `lm()`)

```
> r.lm <- lm(speer~kugel, data=d.sport)
> class(r.lm)
[1] "lm"
```

- These classes come with "methods" for `print()`, `plot()`, `summary()`, etc

```
> summary(r.lm)
> plot(r.lm) ##  explained in another lecture
Call:
lm(formula = speer ~ kugel, data = d.sport)

 .......
```

- `methods(class="lm")` lists the methods for class `"lm"`.

```
> methods(class = "lm")
 [1] add1.lm*           alias.lm*
 [3] anova.lm           case.names.lm*
 [5] confint.lm         cooks.distance.lm*
 [7] deviance.lm*       dfbeta.lm*
 [9] dfbetas.lm*        drop1.lm*
[11] dummy.coef.lm      effects.lm*
[13] extractAIC.lm*     family.lm*
[15] formula.lm*        hatvalues.lm
[17] influence.lm*      kappa.lm
[19] labels.lm*         logLik.lm*
[21] model.frame.lm     model.matrix.lm
[23] nobs.lm*           plot.lm
[25] predict.lm         print.lm
[27] proj.lm*           qr.lm*
[29] residuals.lm       rstandard.lm
[31] rstudent.lm        simulate.lm*
[33] summary.lm         variable.names.lm*
[35] vcov.lm*

   Non-visible functions are asterisked
```


Summary: Many functions in R are *generic* functions, which "dispatch" to calling a "method" depending on the class of the first argument:

Generic Functions—Class—Method:

⟨*generic-func*⟩(⟨*obj*⟩, ...)

dispatches to calling

⟨*generic-func*⟩.⟨*class*⟩(⟨*obj*⟩, ...)

where ⟨*class*⟩ is the class of ⟨*obj*⟩, or it calls

⟨*generic-func*⟩.default(⟨*obj*⟩, ...)

if there is no ⟨*generic-func*⟩.⟨*class*⟩ method.

e.g., after `x <- seq(-4, 4, by = 0.05)`,

- `x` calls `print(x)` which really calls `print.default(x)`
- `summary(d.sport)` really calls `summary.data.frame(d.sport)`
- `plot(y ~ x, ....)` really calls `plot.formula(y ~ x, ....)`
- `plot(x, sin(x))` really calls `plot.default(x, sin(x))` (as there is no `plot.numeric()`)

10.3 Options

- Options taylor some aspects[14] of R's behavior to your desires:

```
> (x <- pi * c(1, 10, 100, 0.1))
[1]   3.1415927  31.4159265 314.1592654   0.3141593
> options(digits = 3)
> print(x[1:3], digits= 15) # (alternative)
[1]   3.14159265358979  31.41592653589793 314.15926535897933
> ## revert to default : 7 digits printing:
> options(digits = 7)
```

[14] mostly only how R *outputs*, i.e., `print()`s or `format()`s things

- Enquire `options()` (or also `par()`)

```
> options("digits")
$digits
[1] 7
> ## or, often more conveniently:
> getOption("digits")
[1] 7
> str(par("mar", "col", "cex", "pch"))# a list
List of 4
 $ mar: num [1:4] 5.1 4.1 4.1 2.1
 $ col: chr "black"
 $ cex: num 1
 $ pch: int 1
```

- Good R programming practice: reset options at end to previous values, either for `options()`:

```
> op <- options(digits=13)
> pi * 100^(0:2)
[1]     3.14159265359   314.15926535898 31415.92653589793
> options(op) ## reset to previous value
> str(op)
List of 1
 $ digits: int 7
```

or also for `par()`:

```
> old.par <- par(mfrow = c(2,2), mgp = c(2,1,0))
> for(i in 1:4) curve(sin(i * pi* x),
+                         main = paste("sin(",i,"pi x)"))
> par(old.par)
> par("mfrow")# areback to  (1, 1)
[1] 1 1
```


- The setting of `options` (and `par`) is ”lost” at the end of the R session.
- In order to always set options and other initial action, use the startup mechanism, see `?Startup`; e.g., on Linux or Mac: can provide a file ‘ /.Rprofile’; e.g., at the Seminar für Statistik ETH, we have (among other things)

```
> options(repos= c(CRAN= "http://cran.ch.r-project.org"),
+         pdfviewer = "evince",
+         browser = "firefox")
```

as default for everyone, in a group-wide `.Rprofile` file.

Using R for Data Analysis and Graphics

11. R packages, CRAN, etc: the R Ecosystem

In this chapter you will learn more on ...

... exploring and installing R packages

... CRAN, etc: a glimpse of “The R World”

... how to get help regarding R

11.1 Packages

- R already comes with $14 + 15 = 29$ packages pre-installed, namely the ”standard” (or ”base”) packages

```
base, compiler, datasets, graphics, grDevices, grid,
methods, parallel, splines, stats, stats4, tcltk, tools,
utils
```

and the ”recommended” packages

```
boot, class, cluster, codetools, foreign, KernSmooth,
lattice, MASS, Matrix, mgcv, nlme, nnet, rpart, spatial,
survival
```

- In R, by default you "see" only a basic set of functions, e.g., `c`, `read.table`, `mean`, `plot`, ..., ...
- They are found in your search path of packages

```
> search() # the first is your workspace
[1] ".GlobalEnv"         "package:graphics"  "package:grDevices
[4] "package:datasets"   "package:stats"     "package:utils"
[7] "package:methods"    "Autoloads"         "package:base"
> ls(pos=1) # == ls()  ~= "your workspace" - cf "introd
[1] "baseP"    "ip.H"     "Mlibrary" "pkg"      "printPs"  "re
[7] "tpkgs"
> str(ls(pos=2)) # content of the  2nd  search() entry
 chr [1:87] "abline" "arrows" "assocplot" "axis" "Axis" ...
> str(ls(pos=9)) # content of the  9th  search() entry
 chr [1:1168] "-" "-.Date" "-.POSIXt" ":" "::" ":::" "!" ...
```


- The default list of R objects (functions, some data sets) is actually not so small: Let's call `ls()` on each `search()` entry:

```
> ls.srch <- sapply(grep("package:", search(),
+                        value=TRUE), # "package:<name>"
+                   ls, all.names = TRUE)
> fn.srch <- sapply(ls.srch, function(nm) {
+     nm[ sapply(lapply(nm, get), is.function) ] })
> rbind(cbind(ls   = (N1 <- sapply(ls.srch, length)),
+             funs = (N2 <- sapply(fn.srch, length))),
+       TOTAL = c(sum(N1), sum(N2)))
                    ls funs
package:graphics    88   88
package:grDevices  107  104
package:datasets   103    0
package:stats      498  497
package:utils      199  197
package:methods    375  224
package:base      1268 1226
TOTAL             2638 2336
```

i.e., 2336 functions in R version 3.0.2

- Till now, we have used functions from packages `"base"`, `"stats"`, `"utils"`, `"graphics"`, and `"grDevices"` without a need to be aware of that.
- `find("⟨name⟩")` can be used:

```
> c(find("print"), find("find"))
[1] "package:base"  "package:utils"
> ## sophisticated version of
> ## rbind(find("mean"), find("quantile"), ....):
> cbind(sapply(c("mean", "quantile", "read.csv", "plot"
+               find))
         [,1]
mean     "package:base"
quantile "package:stats"
read.csv "package:utils"
plot     "package:graphics"
```


- Additional functions (and datasets) are obtained by (possibly first *installing* and then) loading additional packages.
- `> library(MASS)` or `require(MASS)`
- How to find a command and the corresponding package?
  `> help.search("...")` [15], (see Intro)
- On the internet: CRAN (http://cran.r-project.org, see Resources on the internet (slide 15) is a huge repository[16] of R packages, written by many experts.
- CRAN Task Views help find packages by application area
- What does a package do?
  `> help(package = class)` or (⟷)
  `> library(help = class)` .
  Example (of small recommended) package:

```
> help(package = class)
```

[15] can take l..o..n..g.. (only the first time it's called in an R session !)
[16] actually a distributed Network with a server and many mirrors,

```
> help(package = class)
Information on package 'class'

Description:

Package:            class
Priority:           recommended
Version:            7.3-9
Date:               2013-08-21
Depends:            R (>= 3.0.0), stats, utils
Imports:            MASS
Authors@R:          c(person("Brian", "Ripley", role = c("aut",
                    "cre", "cph"), email =
                    "ripley@stats.ox.ac.uk"), person("William",
                    "Venables", role = "cph"))
Description:        Various functions for classification.
Title:              Functions for Classification
ByteCompile:        yes
License:            GPL-2 | GPL-3
URL:                http://www.stats.ox.ac.uk/pub/MASS4/
Packaged:           2013-08-21 12:04:50 UTC; ripley
Author:             Brian Ripley [aut, cre, cph], William Venables
                    [cph]
```


Second part of

```
> help(package = class)
NeedsCompilation:    yes
Repository:          CRAN
Date/Publication:    2013-08-21 14:10:11
Built:               R 3.0.1; x86_64-unknown-linux-gnu; 2013-08-22
                     00:16:39 UTC; unix

Index:

SOM                     Self-Organizing Maps: Online Algorithm
batchSOM                Self-Organizing Maps: Batch Algorithm
condense                Condense training set for k-NN classifier
knn                     k-Nearest Neighbour Classification
knn.cv                  k-Nearest Neighbour Cross-Validatory
                        Classification
knn1                    1-nearest neighbour classification
lvq1                    Learning Vector Quantization 1
lvq2                    Learning Vector Quantization 2.1
lvq3                    Learning Vector Quantization 3
lvqinit                 Initialize a LVQ Codebook
lvqtest                 Classify Test Set from LVQ Codebook
multiedit               Multiedit for k-NN Classifier
olvq1                   Optimized Learning Vector Quantization 1
```


Installing packages from CRAN

- Via the "Packages" menu (in RStudio or other GUIs for R)
- Directly via `install.packages()`[17].
  Syntax:

  `install.packages(`*pkgs*, *lib*, *repos* = *getOption*("*repos*")`, ...)`

  pkgs: character vector names of packages whose current versions should be downloaded from the repositories.
  lib: character vector giving the library directories where to install the packages. If missing, defaults to the first element of `.libPaths()`.
  repos: character with base URL(s) of the repositories to use, typically from a CRAN mirror. You can choose it interactively via `chooseCRANmirror()` or explicitly by `options(repos= c(CRAN="http://..."))`.
  `...`: many more (*optional*) arguments.

[17] which is called anyway from the menus mentioned above

Installing packages – Examples

- Install once, then use it via `require()` or `library()`:

```
> chooseCRANmirror()
> install.packages("sfsmisc")
> ## For use:
> require(sfsmisc) # to "load and attach" it
```

- ```
  > install.packages("sp", # using default "lib"
  +         repos = "http://cran.CH.r-project.org")
  ```
- or into a non-default *library* of packages:

```
> install.packages("sp", lib = "my_R_folder/library",
+         repos = "http://cran.CH.r-project.org")
> ## and now load it from that library (location):
> library(sp, lib = "my_R_folder/library")
```

Note that you need *"write permission"* in the corresponding "library", i.e., folder of packages (by default: `.libPaths()[1]`).
```

Maintaining your package installations

Packages are frequently updated or improved. When new R versions are released, some packages need changing too. Therefore it is necessary to maintain your package installations. An easy way to do this is also via command line:

```
> update.packages()
```

This will invoke a dialogue where you can select which packages you would like to update. It will list the current version of the package and the version installed on your computer.