Introduction to the R software

Chapter 1. Basics of R

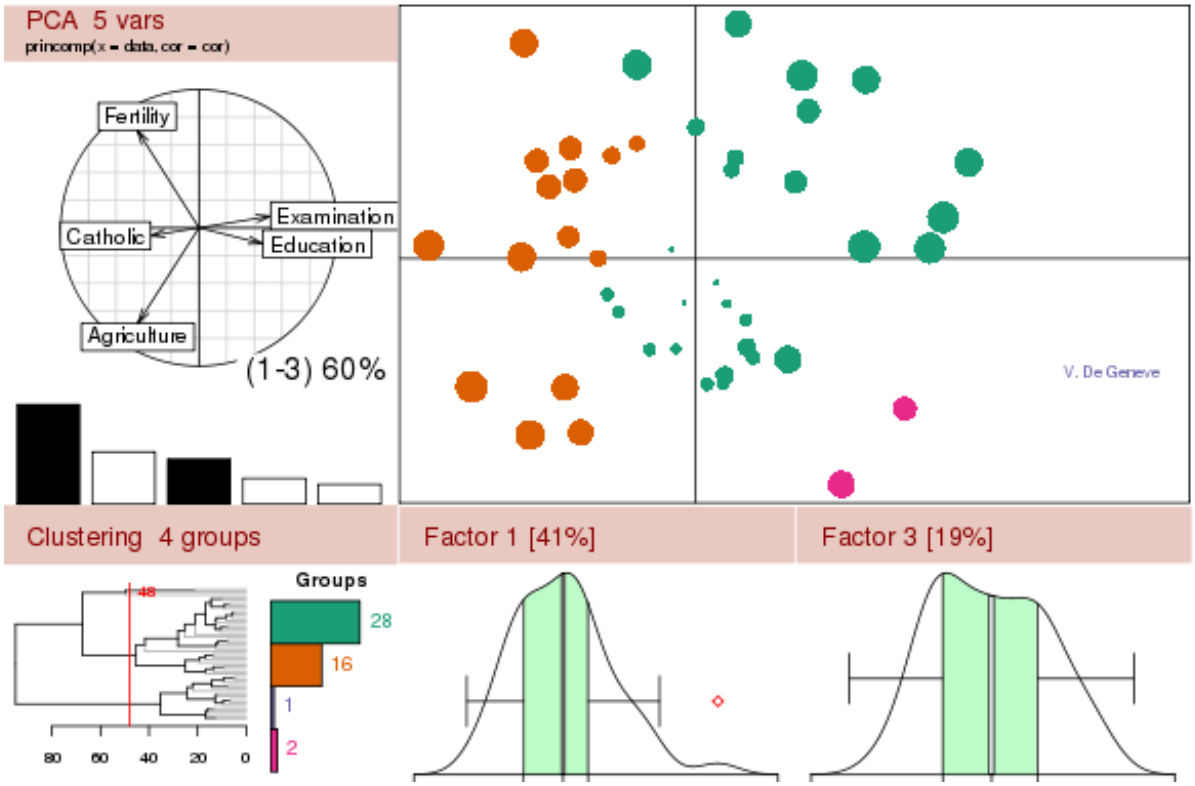
Peng Zhang

School of Mathematical Science
Zhejiang University

Outline

What is R?

- GNU Project Developed by John Chambers @ Bell Lab
- Ross Ihaka and Robert Gentleman
- Free software environment for statistical computing and graphics
- Functional programming language written primarily in C, Fortran



Why R?

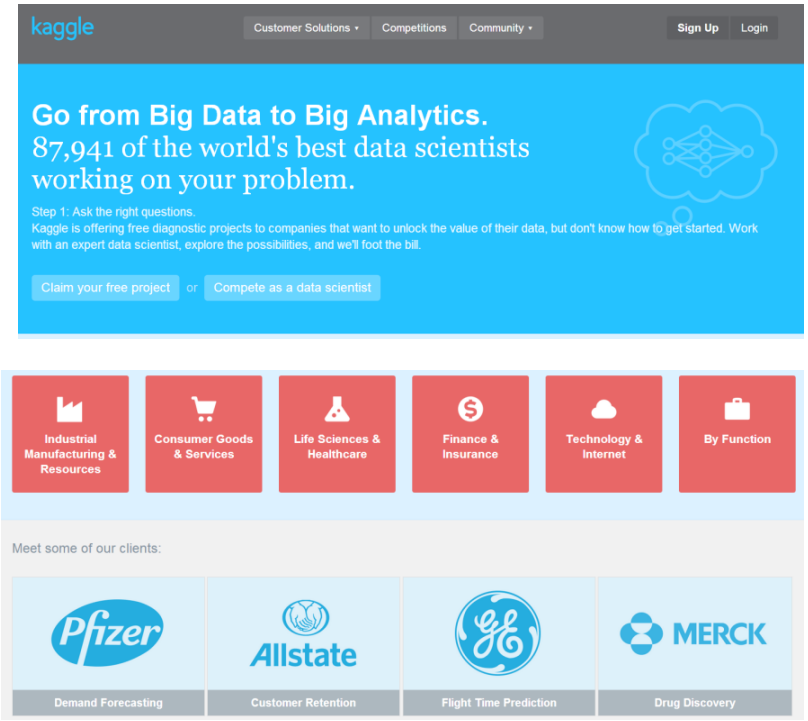
- Vast capabilities, wide range of statistical and graphical techniques
- Very popular in academia, growing popularity in business
- Written primarily by statisticians
- FREE as in free speech: collaborative development
- FREE as in free beer: no cost
- Excellent community support: mailing list, blogs, tutorials
- Easy to extend by writing new functions
- Excellent graphical user interfaces (though you may have to install them)

Why use R?

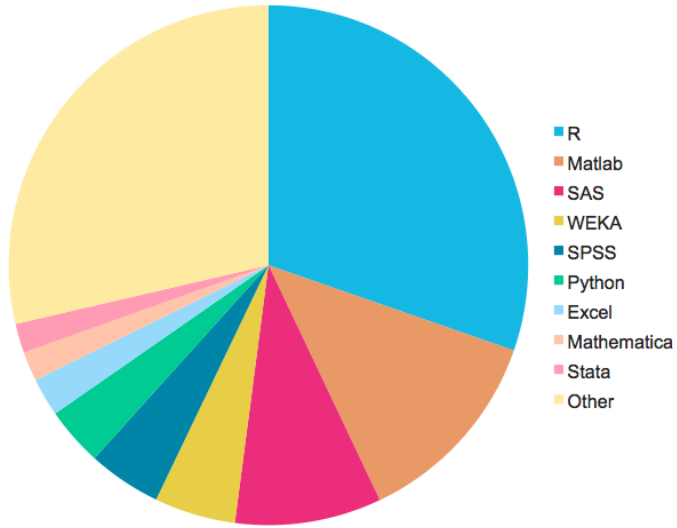
R is especially powerful for data manipulation, calculations and plots. Its features include:

- an integrated and very well conceived documentation system (in English);
- efficient procedures for data treatment and storage;
- a suite of operators for calculations on tables, especially matrices (but also arrays);
- a vast and coherent collection of statistical procedures for data analysis;
- advanced graphical capabilities;
- a "simple" and efficient programming language, including conditioning, loops, recursion and input-output possibilities.

Kaggle



R is the most widely language used by kaggle participants



http://www.kaggle.com/

Google and Apple

Google

What is your programming language of choice, R, Python or something else?

"I use R, and occasionally matlab, for data analysis. There is a large, active and extremely knowledgeable R community at Google."

http://simplystatistics.org/2013/02/15/interview-with-nick-chamandy-statistician-at-google/

"Expert knowledge of SAS (With Enterprise Guide/Miner) required and candidates with strong knowledge of R will be preferred"

http://www.kdnuggets.com/jobs/13/03-29-apple-sr-data-scientist.html?utm_source=twitterfeed&utm_medium=facebook&utm_campaign=tfb&utm_content=FaceBook&utm_term=analytics#.UVXibgXOpfc.facebook

Installing R

Let's start by installing the software!

1. Download the file `R-x-win.exe` (where x is the number of the latest version) at the address: `http://cran.r-project.org/bin/windows/base/`
2. Save this executable file on the Windows Desktop and double-click the file `R-x-win.exe`
3. The software then installs. Follow the instructions displayed on your screen and keep the default options.
4. When the icon is added to the Desktop, installation is complete.

R and Statistics

Many **classical** and **modern** statistical techniques are implemented in **R**. The most common methods for statistical analysis, such as:

- descriptive statistics;
- hypothesis testing;
- analysis of variance;
- linear regression methods (simple and multiple);
- and so on

are directly included at the **core** of the system.

Third part of my book covers the following notions: basic mathematics, descriptive statistics, generation of random values, confidence intervals and hypothesis testing, simple and multiple linear regression, elementary analysis of variance.

Extending R

Most **advanced or recent statistical methods** are available through external **packages**, easy to install from **R**.

They are all grouped and can be browsed on the website of the *Comprehensive R Archive Network* (CRAN):
`http://cran.r-project.org/web/packages/available_packages_by_name.html`

See also the Task Views (on the CRAN) that group packages related to some domains of interest:
`http://cran.r-project.org/web/views/`

Official website of R: The Comprehensive R Archive Network

`http://cran.r-project.org`

R and plots

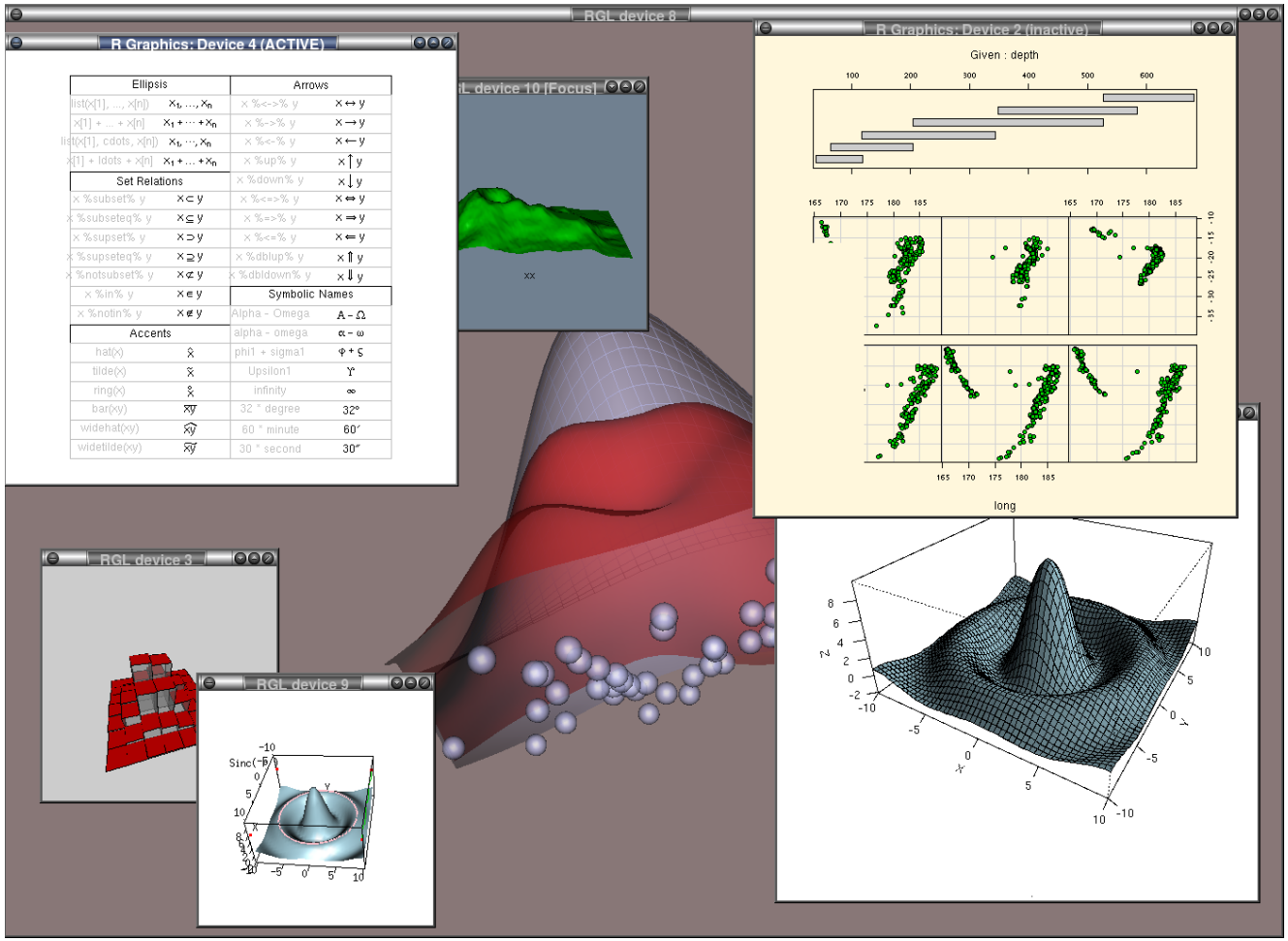


FIGURE : A few of the graphical possibilities offered by R.

R and other software

R can interact with the following software:

- use R from within Excel: `http://rcom.univie.ac.at/download.html#RExcel`
- R from within SAS: `http://support.sas.com/rnd/app/studio/Rinterface2.html`
- R from within SPSS: `http://www.ibm.com/developerworks/library/ba-call-r-spss`
- R from Matlab: `http://www.mathworks.com/matlabcentral/fileexchange/5051-matlab-r-link`
- Matlab from R: install package `R.matlab`

Getting help in R

- Start html help, search/browse using web browser
  - at the R console:
    `help.start()`
  - or use the help menu from you GUI
- Look up the documentation for a function
  `help(topicName)`
  `?topicName`
- Look up documentation for a package
  `help(package="packageName")`
- Search documentation from R
  `help.search("topicName")`

Things to keep in mind

- Case sensitive, like Stata (unlike SAS)
- Comments can be put almost anywhere, starting with a hash mark '#'; everything to the end of the line is a comment
- The command prompt ">" indicates that **R** is ready to receive commands
- If a command is not complete at the end of a line, **R** will give a different prompt, '+' by default
- Parentheses must always match (first thing to check if you get an error)
- **R** Does not care about spaces between commands or arguments
- Names should start with a letter and should not contain spaces
- Can use "." in object names (e.g., "my.data")
- Use / instead of \ in path names

R as a calculator

```
> 2+2
[1] 4
> 2*3*4*5          # * denotes 'multiply'
[1] 120
> sqrt(10)         # the square root of 10
[1] 3.162278
> pi               # R knows about pi
[1] 3.141593
> 2*pi*6378        # Circumference of earth at equator (km)
[1] 40074.16
> 3*4^
+ 2
[1] 48
> 3*4^2; (3*4)^2
[1] 48
[1] 144
```

Displaying results and variable redirecting

R responds to your requests by displaying the result obtained after evaluation. **But, this result is displayed, then lost.**
Nevertheless, we can use the **assignment arrows**: <- or ->

```
x <- 1        # Assignment.
> x           # Display.
[1] 1
> 2 -> x      # Assignment (in the other direction).
> x           # Display.
[1] 2
> (x <- 1)    # Assignment AND display.
[1] 1
```

The = symbol

It is not good practice to use the = symbol for assignment.

Rules for choosing a variable name

- a variable name can only include alphanumerical characters as well as the dot (.);
- variable names are *case sensitive*, which means that **R** distinguishes upper and lower case;
- a variable name may not include white space or start with a digit, unless it is enclosed in quotation marks "".

Work strategy

You should use either a *script window* (**R** *editor*) or Rstudio to type your commands before sending them to **R** for execution.

The key combinations `CTRL+R` (or `CTRL+ENTER`) can be used to execute your commands.

You can also use the `source(file.choose())` command (typed in the **R** console) to read and execute the contents of an external **R** script file.

Note: The function `help()` can be used to access the documentation: `help(source)`.

Using functions

A function in **R** is defined by its **name** and by the list of its **parameters** (or **arguments**). Most functions output a **value**.

Using a function (or **calling** or **executing** it) is done by typing its name followed, in brackets, by the list of (formal) arguments to be used. Arguments are separated by commas. Each argument can be followed by the sign = and the value to be given to the argument.

```
functionname(arg1=value1,arg2=value2,arg3=value3)
```

Note that you do not necessarily need to indicate the names of the arguments, but only the values, as long as you follow their order.
For any **R** function, some arguments must be specified and others are optional (because a default value is already given in the code of the function).

Understanding the use of arguments

The function `log(x,base=exp(1))` can take two arguments: `x` (its value must be specified) and `base` (optional, because a default value is provided as `exp(1)`).

You can call a function by playing with the arguments in several different ways. This is an important feature of **R** which makes it easier to use.

```
log(3)
log(x=3)
log(x=3,base=exp(1))
log(x=3,exp(1))
```

```
log(3,base=exp(1))
log(3,exp(1))
log(base=exp(1),3)
log(base=exp(1),x=3)
```

Question: what is done with this instruction?

```
log(exp(1),3)
```

Using functions

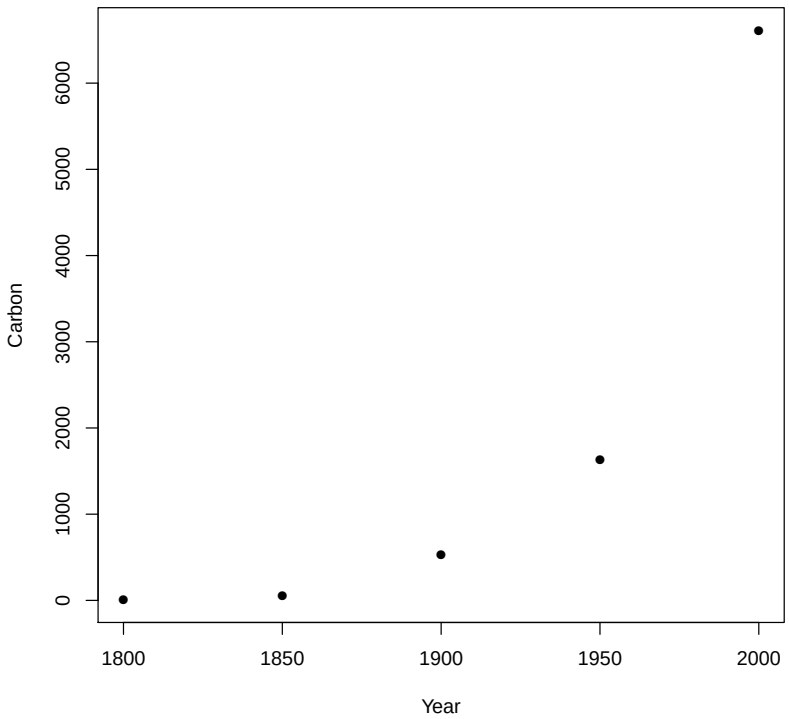
Don't forget the brackets when you call a function

A common mistake for beginners is forgetting the brackets:

```
> factorial # Typing the name gives the code.
function (x)
gamma(x + 1)
<environment: namespace:base>
> factorial(6)
[1] 720
> date
function ()
.Internal(date())
<environment: namespace:base>
> date() # Brackets are also necessary when
>        #  no arguments are required.
[1] "Wed Jan  9 16:04:32 2013"
```

Entry of data at the command line

```
Year <- c(1800, 1850, 1900, 1950, 2000)
Carbon <- c(8, 54, 534, 1630, 6611)
## Now plot Carbon as a function of Year
plot(Carbon ~ Year, pch=16)
```



	year	carbon
1	1800	8
2	1850	54
3	1900	534
4	1950	1630
5	2000	6611

Collection of vectors into a data frame

```
> fossilfuel <- data.frame(year=Year, carbon=Carbon)
> fossilfuel     # Display the contents of the data frame.
   year  carbon
1  1800       8
2  1850      54
3  1900     534
4  1950    1630
5  2000    6611
> plot(carbon ~ year, data=fossilfuel, pch=16)
```

The second column of the data frame

```
> fossilfuel[, 2]
> fossilfuel[, "carbon"]
> fossilfuel$carbon
```

Working directory and workspace

- Type `getwd()` to display the name of the working directory or use menu.
- Type `ls()` to list the workspace contents.
- Type `q()` to quit (exit) from R or use menu.
- Use menu to save image file.
- Use menu or `load()` to load image file, usually `.RData` file.

Packages

- Use package installer to install packages or

```
install.packages("DAAG")
install.packages(c("magic", "schoolmath"), dependencies=T)
```

- Use package manager to load packages or

```
library(DAAG)
```

- Use `sessionInfo()` to see which packages are currently attached.
- The **CRAN Task Views** can be a good place to start when looking for abilities of a particular type.
  http://cran.r-project.org/web/views/

The various data types in R

One of the main strengths of R is its ability to organize data in a structured way. **This will turn out to be very useful for many statistical procedures.**

Data type	Type in R	Display
real number (integer or not)	numeric	`3.27`
complex number	complex	`3+2i`
logical (true/false)	logical	TRUE or FALSE
missing	logical	NA
text (string)	character	`"text"`
binary	raw	`1c`

Dealing with data type

```
> a <- 1 # Similar to: a <- 1.0
> typeof(a)
[1] "double"
> c <- as.integer(a)
> typeof(c)
[1] "integer"
> b <- 3.4
> c(b>a,a==b)
[1] TRUE FALSE
> is.numeric(a)
[1] TRUE
> is.integer(a)
[1] FALSE
> x <- TRUE # Similar to: x <- T
> is.logical(x)
[1] TRUE
```

Character string type

Any information between quotation marks (single ‘ or double ”) corresponds to a character string.

```
> a <- "R is my friend"
> mode(a)
[1] "character"
> is.character(a)
[1] TRUE
```

The various data structures in R

Data structure	Instruction in R	Description
vector	*c*()	Sequence of elements of the **same nature**.
matrix	*matrix*()	Two-dimensional table of elements of the **same nature**.
multidimensional table	*array*()	More general than a matrix; table with several dimensions.
list	*list*()	Sequence of R structures of any (and possibly different) nature.
individual×variable table	*data.frame*()	Two-dimensional table. The columns can be of different natures, but must have the same length.
factor	*factor*(), *ordered*()	Vector of character strings associated with a modality table.
dates	*as.Date*()	Vector of dates.
time series	*ts*()	Values of a variable observed at several time points.

Vectors

- Examples of vectors are

```
> c(2, 3, 5, 2, 7, 1)
[1] 2 3 5 2 7 1
> c(T, F, F, F, T, T, F)
[1] TRUE FALSE FALSE FALSE TRUE TRUE FALSE
> c("Canberra", "Sydney", "Canberra", "Sydney")
[1] "Canberra" "Sydney" "Canberra" "Sydney"
```

- The function `c()` join vectors

```
> x <- c(2, 3, 5, 2, 7, 1)
> y <- c(10, 15, 12)
> z <- c(x, y)
>z
[1] 2 3 5 2 7 1 10 15 12
```

Relational operators

- Relational operators are `<`, `<=`, `>`, `>=`, `==`, and `!=`.
- Check `help(Comparison)`, `help(Logic)`, and `help(Syntax)` for more information on relational operators.

```
> x <- c(3, 11, 8, 15, 12)
> x > 8
[1] FALSE TRUE FALSE TRUE TRUE
> x != 8
[1] TRUE TRUE FALSE TRUE TRUE
```

Extract elements of vectors

There are three ways to extract elements of vectors.

- Specify the indices of the elements that are to be extracted

```
> x <- c(3, 11, 8, 15, 12)
> x[c(2, 4)] # Elements in positions 2 and 4 only
[1] 11 15
```

- Use negative subscripts to omit the elements.

```
> x[-c(2,3)] # Remove the elements in positions 2 and 3.
[1] 3 15 12
```

- Specify a vector of logical values.

```
> x > 10
[1] FALSE TRUE FALSE TRUE TRUE
> x[x > 10]
[1] 11 15 12
```

Name elements of vectors, Patterned data

Elements of vectors can be given names.

```
> heights <- c(Andreas=178, John=185, Jeff=183)
> heights[c("John","Jeff")]
  John Jeff
   185  183
```

```
> 5:15
[1] 5 6 7 8 9101112131415
> seq(from=5, to=22, by=3) # The first value is 5.
[1] 5 8 11 14 17 20         # The final value is <= 22.
> rep(c(2,3,5), 4)
[1] 2 3 5 2 3 5 2 3 5 2 3 5
> c(rep("female", 3), rep("male", 2))
[1] "female" "female" "female" "male" "male"
```

Missing values

The missing value symbol is NA.

```
 > library(DAAG)
 > nbranch <- subset(rainforest, species=="Acacia
mabellae")$branch
 > nbranch # Number of small branches (2cm or less)
 [1] NA 35 41 50 NA NA NA NA NA 4 30 13 10 17 46 92
 > mean(nbranch)
 [1] NA
 > mean(nbranch, na.rm=TRUE)
 [1] 33.8
 > ## Replace all NAs by -999
 > nbranch[is.na(nbranch)] <- -999
 > nbranch
 [1] -999 35 41 50 -999 -999 -999 -999 -999 4 30 13
 [13] 10 17 46 92
```

Missing data

A missing or undefined value is indicated by the instruction *NA* (for *non available*).

```
> x <- c(3,NA,6)
> is.na(x)
[1] FALSE  TRUE FALSE
> mean(x)                    # Let's try to calculate the mean.
[1] NA
> mean(x,na.rm=TRUE)         # The na.rm argument indicates
                             # that NA's should be removed.
[1] 4.5
```

Factors and time series

- A factor is stored internally as a numeric vector with values $1, 2, 3, ..., k$.

```
> ## Create character vector
> gender <- c(rep("female",691), rep("male",692))
> ## From character vector, create factor
> gender <- factor(gender)
> levels(gender)
[1] "female" "male"
```

- The function `ts()` converts numeric vectors into time series objects.

```
numjobs <- c(982,981,984,982,981,983,983,983,983,979,973,
  979,974,981,985,987,986,980,983,983,988,994,990,999)
numjobs <- ts(numjobs, start=1995, frequency = 12)
plot(numjobs)
first15 <- window(numjobs, start=1995.75, end=1996.25)
```

Matrices and arrays

```
> ( X <- matrix(1:12,nrow=4,ncol=3,byrow=TRUE) )
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
[4,]   10   11   12
> class(X)
[1] "matrix"
> X <- array(1:12,dim=c(2,2,3))
```

Lists: the most flexible and richest structure in R

Unlike the previous structures, lists can **group together in one structure data of different types** without altering them.

```
> ( A <- list(TRUE,-1:3,my.matrix=matrix(1:4,nrow=2),
+             c(1+2i,3),"A character string") )
[[1]]
[1] TRUE
[[2]]
[1] -1  0  1  2  3
$my.matrix
     [,1] [,2]
[1,]    1    3
[2,]    2    4
[[4]]
[1] 1+2i 3+0i
[[5]]
[1] "A character string"
```

Note: we have named this element of A.

Data frames: most used structure in Statistics

```
> ( BMI <- data.frame(Gender=c("M","F","M","F","M","F"),
+     Height=c(1.83,1.76,1.82,1.60,1.90,1.66),
+     Weight=c(67,58,66,48,75,55),row.names=c("Jack",
+     "Julia","Henry","Emma","William","Elsa")) )
        Gender Height Weight
Jack         M   1.83     67
Julia        F   1.76     58
Henry        M   1.82     66
Emma         F   1.60     48
William      M   1.90     75
Elsa         F   1.66     55
> str(BMI) # Structure of each column.
'data.frame': 6 obs. of  3 variables:
 $ Gender: Factor w/ 2 levels "F","M": 2 1 2 1 2 1
 $ Height: num  1.83 1.76 1.82 1.6 1.9 1.66
 $ Weight: num  67 58 66 48 75 55
```

Data frame

```
> Cars93.summary
        Min.passengers  Max.passengers  No.of.cars  abbrev
Compact              4               6          16       C
  Large              6               6          11       L
Midsize              4               6          22       M
  Small              4               5          21      Sm
 Sporty              2               4          14      Sp
    Van              7               8           9       V

> head(Cars93.summary, n=3) # Display the first 3 rows
        Min.passengers  Max.passengers  No.of.cars  abbrev
Compact              4               6          16       C
  Large              6               6          11       L
Midsize              4               6          22       M
```

Column and row names, subsets

- The function `rownames()` extracts the names of rows, while `colnames()` extracts column names,

```
rownames(Cars93.summary) # Extract row names
colnames(Cars93.summary) # Extract column names
```

- The functions `names()` (or `colnames()`) and `rownames()` can also be used to assign new names.

```
names(Cars93.summary)[3] <- "numCars"
names(Cars93.summary) <- c("minPass","maxPass","numCars",
"code")
```

```
Cars93.summary[1:3, 2:3] # Rows 1-3 and columns 1-3
Cars93.summary[, 2:3] # Columns 2-3 (all rows)
Cars93.summary[, c("No.of.cars", "abbrev")] # by name
Cars93.summary[, -c(2,3)] # omit columns 2 and 3
```

List

A list is an arbitrary collection of R objects, allowing different classes and lengths.

```
> ## Cities with more than 2.5 million inhabitants
> USACanada <- list(USACities=c("NY", "LA", "Chicago"),
+                   CanadaCities=c("Toronto", "Montreal"),
+                   millionsPop=c(USA=305.9, Canada=31.6))
> USACanada
$USACities
[1] "NY" "LA" "Chicago"

$CanadaCities
[1] "Toronto" "Montreal"

$millionsPop
  USA Canada
305.9   31.6
```

with() and attach()

- c(mean(cfseal$weight), median(cfseal$weight))

```
> ## cfseal (DAAG) has data on Cape Fur seals
> with(cfseal, c(mean(weight), median(weight)))
[1] 54.8 46.2
> with(pair65, # stretch of rubber bands, from DAAG
+       {lenchange <- heated-ambient
+       c(mean(lenchange), median(lenchange))
+ })
[1] 6.33 6.00
```

- An alternative is attach().

```
> attach(fossilfuel) # Attach data frame fossilfuel
> year
[1] 1800 1850 1900 1950 2000
> detach(fossilfuel) # Detach data frame
```

Data frames and matrices

```
fossilfuelmat <- matrix(c(1800, 1850, 1900, 1950, 2000,
                          8, 54, 534, 1630, 6611), nrow=5)
colnames(fossilfuel) <- c("year", "carbon")
fossilfuelmat <- cbind(year=c(1800, 1850, 1900, 1950,
                       2000), carbon=c(8, 54, 534, 1630, 6611))
```

- Matrix elements are stored in column order in one long vector, i.e., columns are stacked one above the other, with the first column first.
- The extraction of submatrices has the same syntax as for data frames.
- The `names()` function returns `NULL` when the argument is a matrix.
- The function `nrow()` returns the number of rows, while `ncol()` returns the number of columns.

Common useful built-in functions

```
all()          # returns TRUE if all values are TRUE
any()          # returns TRUE if any values are TRUE
args()         # information on the arguments to a function
cat()          # prints multiple objects, one after the other
cumprod()      # cumulative product
cumsum()       # cumulative sum
diff()         # form vector of first differences
               # N. B. diff(x) has one less element than x
history()      # displays previous commands used
is.factor()    # returns TRUE if the argument is a factor
is.na()        # returns TRUE if the argument is an NA
               # NB also is.logical(), is.matrix(), etc.
length()       # number of elements in a vector or of a list
ls()           # list names of objects in the workspace
```

Common useful built-in functions

```
mean()        # mean of the elements of a vector
median()      # median of the elements of a vector
order()       # x[order(x)] sorts x (by default NAs are last)
print()       # prints a single R object
range()       # minimum and maximum value elements of vector
sort()        # sort elements into order, by default
              # omitting NAs
rev()         # reverse the order of vector elements
str()         # information on an R object
unique()      # form the vector of distinct values
which()       # locates 'TRUE' indices of logical vectors
which.max()   # locates (first) maximum of a numeric vector
which.min()   # locates (first) minimum of a numeric vector
with()        # do computation using columns of specified
              # data frame
```

table() and sapply()

- The `table()` function can be used to count up the numbers of observations in each group combination to form a contingency table.

```
> library(DAAG) # tinting is from DAAG
> table(Sex=tinting$sex, AgeGroup=tinting$agegp)
        AgeGroup
Sex   younger  older
  f        63     28
  m        28     63
```

- The function `sapply()` applies a function to each column of a data frame, or to each element of a list.

```
> sapply(jobs[, -7], range)
        BC  Alberta  Prairies  Ontario  Quebec  Atlantic
[1,]  1737     1366       973     5212    3167       941
[2,]  1840     1436       999     5360    3257       968
```

Goals of this section

Describing the instructions

- to enter data directly in **R**;
- to import or export data, to and from other software:
  - Excel;
  - SPSS;
  - Minitab;
  - SAS;
  - Matlab.

Importing data from a text file

The three main **R** functions to import data from a text file.

Function name	**Description**
`read.table()`	Best suited for data sets presented as tables, as it is often the case in Statistics.
`read.ftable()`	Reads contingency tables.
`scan()`	Much more flexible and powerful. Use this in all other cases.

Reading data with `read.table()`

To read data present in an ASCII file:

```
my.data <- read.table(file=file.choose(),header=TRUE,
      sep="\t",dec=".",row.names=1)
```

Argument name	Description
`file=path/to/file`	Location and name of the file to be read.
`header=TRUE`	Indicates whether the variable names are given on the first line of the file.
`sep="\t"`	The values on each line are separated by this character ("\t"=TABULATION; ""=whitespace; ","=,; etc.).
`dec="."`	Decimal mark for numbers ("." or ",").
`row.names=1`	1st column of the file gives the individuals' names.

Reading data with `read.table()`

The path can be specified explicitly:

```
my.data <- read.table(file="C:/MyFolder/somedata.txt")
```

or (doubling the backslashes):

```
my.data <- read.table(file="C:\\MyFolder\\somedata.txt")
```

We can also use the function `setwd()` to change the work directory (equivalent to using the menu "File/Change current directory"):

```
setwd("C:/MyFolder")
my.file <- "mydata.txt"
data <- read.table(file=my.file)
head(data)
```

Reading data with `read.ftable()`

To read contingency tables like this one:

```
          "alcohol" "nondrinker" "occasional drinker" "regular drinker"
"GENDER" "tobacco"
"M"      "non-smoker"          6                   19                   7
         "former smoker"       0                    9                   0
         "smoker"              1                    6                   5
"F"      "non-smoker"         12                   26                   2
         "former smoker"       3                    5                   1
         "smoker"              1                    6                   1
```

use the instructions:

```
Intima.table <- read.ftable("Intima_ftable.txt",
        row.var.names=c("GENDER","tobacco"),
        col.vars=list("alcohol"=c("nondrinker",
        "occasional drinker","regular drinker")))
ftable(Intima.table)
```

Reading data with the function scan()

Function scan() should be used when the data are not organized as a rectangular table.

For example, suppose your data file contains the following lines:

```
File description:
-----------------------
The individual data are registered for nine variables
in the following order:
GENDER AGE height weight tobacco packyear SPORT measure alcohol

Data:
--------
1 33 170 70 1 1 0 0,52 1 2 33 177 67 2 20 0 0,42 1
2 53 164 63 1 30 0 0,65 0 2 42 169
76 1 26 1 0,48 1
```

Reading data with the function scan()

Here are the commands we suggest you use to read this file. The argument skip=*n* is used to omit reading the first *n* lines of the file.

```
# Reading variable names:
variable.name <- scan("Intima_Media2.txt",skip=4,
                      nlines=1,what="")
# Reading data:
data <- scan("Intima_Media2.txt",skip=7,dec=",")
mytable <- as.data.frame(matrix(data,ncol=9,byrow=TRUE))
colnames(mytable) <- variable.name
```

Note: you can read data files directly from Internet with read.table() and scan().

```
read.table("http://www.biostatisticien.eu/springeR/
                    temperature.dat")
```

Importing data from Excel or the Open Office spreadsheet

First approach: using copy-paste

Using the mouse, select the range of the data (in the spreadsheet) which you wish to incorporate into **R**. Once the data are selected, copy them to the clipboard (from the Edit menu, or with the keyboard shortcuts `CTRL+C` on Windows or `COMMAND+C` on a Mac).

All you need to do now is type the following instructions in the **R** console to transfer the data from the clipboard.

```
x <- read.table(file("clipboard"),sep="\t",
                header=TRUE,dec=",")
```

Importing data from Excel or the Open Office spreadsheet

Second approach: Using an intermediary ASCII file

Save your file in an ASCII format, then refer to the previous section.

Third approach: Using specialized packages

- Use function `read.xls()` from package `gdata` (needs PERL, can be installed on Windows via the file Rtools29.exe);
- Use function `read.xlsx()` from packages `xlsx` or `openxlsx`.

Importing data from SPSS, Minitab, SAS or Matlab

Table : Packages and **R** importation functions from common software.

Software	Package	R function	File extension	Output format
SPSS	`foreign`	`read.spss()`	`*.sav`	`list`
Minitab	`foreign`	`read.mtp()`	`*.mtp`	`list`
SAS	`foreign`	`read.xport()`	`*.xpt`	`data.frame`
Matlab	`R.matlab`	`readMat()`	`*.mat`	`list`

Large data files

R can handle large data sets quickly and efficiently. For this, you need to specify explicitly the type of each column using the argument `colClasses` (e.g., `=rep("character",3)`) of the function `read.table()`.

```
tm <- Sys.time() # Gets the current time.
dbsnp <- read.table("dbsnp123.dat")
Sys.time() - tm
Time difference of 5.063645 mins

tm <- Sys.time()
dbsnp <- read.table("dbsnp123.dat",
          colClasses=rep("character",3))
Sys.time() - tm
Time difference of 13.75810 secs
```

Exporting data to an ASCII text file:

```
write.table(mydata, file = "myfile.txt", sep = "\t")
```

Exporting data to Excel or OpenOffice Calc:

```
X<-data.frame(Weight=c(80,90,75),Height=c(182,190,160))
write.table(X,file("clipboard"),sep="\t",dec=".",
                                        row.names=FALSE)
```

The data have now been copied to the clipboard. You can now paste them into your spreadsheet, for example by typing `Ctrl+V`.

See also the function `write.xlsx()` from packages `xlsx` or `openxlsx`.

Entering toy data: the c(), seq() and :() functions

```
> c(1,5,8,2.3)
[1] 1.0 5.0 8.0 2.3
> seq(from=4,to=5)
[1] 4 5
> seq(from=4,to=5,by=0.1)
 [1] 4.0 4.1 4.2 4.3 4.4 4.5 4.6 4.7 4.8 4.9 5.0
> seq(from=4,to=5,length=8)
[1] 4.000000 4.142857 4.285714 4.428571 4.571429 4.714286
[7] 4.857143 5.000000
> 1:12
 [1]  1  2  3  4  5  6  7  8  9 10 11 12
```

Entering toy data: the rep() function

```
> rep(1,4)
[1] 1 1 1 1
> rep(1:4, 2)
[1] 1 2 3 4 1 2 3 4
> rep(1:4, each = 2)
[1] 1 1 2 2 3 3 4 4
> rep(1:4, c(2,1,2,3))
[1] 1 1 2 3 3 4 4 4
> rep(1:4, each = 2, len = 4)
[1] 1 1 2 2
> rep(1:4, each = 2, len = 10)
 [1] 1 1 2 2 3 3 4 4 1 1
> rep(1:4, each = 2, times = 3)
 [1] 1 1 2 2 3 3 4 4 1 1 2 2 3 3 4 4 1 1 2 2 3 3 4 4
```

Entering data from a hard copy

- **Creating a vector with the function** `scan()`

In this context, `scan()` is more user-friendly than `c()`. It can be used to easily enter data as you go.

```
> z <- scan() # R is waiting for you to enter data.
1: 4.2
2: 5.6
3: 8.9
4: 1
5: 2.3
6:      # Press ENTER after an empty line
        # to halt the procedure.
Read 5 items
> z
[1] 4.2 5.6 8.9 1.0 2.3
```

Entering data from a hard copy

- **Creating several vectors of different lengths**

```
data.entry("")
```

You can change the names of the variables (columns) and enter data. Columns can contain different numbers of observations. If you leave the mini spreadsheet and type in the instruction `ls()`, you will see the variables you have created.

- **Creating and individual×variables table**

To enter data directly into **R**'s mini spreadsheet (as if using Excel), simply use the function `de()` (for *data entry*), as shown in the following instruction.

```
X <- as.data.frame(de(""))
```

Change the names of the variables and the types of the columns by clicking on the cells on the first row.

Goals of this section

Describing the instructions for

- elementary data manipulation;
- extraction tool (direct and by logical mask);
- dealing with character strings.

Vector arithmetic

R can operate on vectors and matrices:

```
> x <- c(1,2,4,6,3)
> y <- c(4,7,8,1,1)
> x + y
[1]  5  9 12  7  4
```

Returns the vector of sums $(x_1 + y_1, \ldots, x_n + y_n)$.

This is one of the **main strengths** of R. It is called **vectorization**.

```
> M <- matrix(1:9,nrow=3)
> exp(M)
          [,1]      [,2]     [,3]
[1,]  2.718282  54.59815 1096.633
[2,]  7.389056 148.41316 2980.958
[3,] 20.085537 403.42879 8103.084
```

Vectorization is much quicker

```
> x <- rnorm(1000000)
> system.time(z<-0;for(i in 1:1000000) z <- z + x[i])
   user  system elapsed
  1.143   0.009   1.154
> z
[1] 367.689
> system.time(z <- sum(x))
   user  system elapsed
  0.002   0.000   0.003
> z
[1] 367.689
```

Recycling

```
# Vector of length 15:
> x <- c(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15)
# Vector of length 10:
> y <- c(1,2,3,4,5,6,7,8,9,10)
# Vector of length 15:
> x + y
 [1]  2  4  6  8 10 12 14 16 18 20 12 14 16 18 20

> matrix(1:4,ncol=3,nrow=3)
     [,1] [,2] [,3]
[1,]    1    4    3
[2,]    2    1    4
[3,]    3    2    1
```

Basic functions

- `length()`: returns the length of a vector.

```
> length(c(1,3,6,2,7,4,8,1,0))
[1] 9
```

- `sort()`: sorts the elements of a vector, in increasing or decreasing order.

```
> sort(c(1,3,6,2,7,4,8,1,0))
[1] 0 1 1 2 3 4 6 7 8
> sort(c(1,3,6,2,7,4,8,1,0),decreasing=TRUE)
[1] 8 7 6 4 3 2 1 1 0
```

- `rev()`: rearranges the elements of a vector in reverse order.

```
> rev(c(1,3,6,2,7,4,8,1,0))
[1] 0 1 8 4 7 2 6 3 1
```

Basic functions

• `order()`, `rank()` : the first function returns the vector of (increasing or decreasing) ranking indices of the elements. The second function returns the vector of ranks of the elements. In case of a tie, the ordering is always from left to right.

```
> vec <- c(1,3,6,2,7,4,8,1,0) ; names(vec) <- 1:9
> vec
1 2 3 4 5 6 7 8 9
1 3 6 2 7 4 8 1 0
> sort(vec)
9 1 8 4 2 6 3 5 7
0 1 1 2 3 4 6 7 8
> order(vec)
[1] 9 1 8 4 2 6 3 5 7
> rank(vec)
  1   2   3   4   5   6   7   8   9
2.5 5.0 7.0 4.0 8.0 6.0 9.0 2.5 1.0
```

Merging columns of matrices or data.frames

The generic function is `cbind()`.

```
> cbind(1:4,5:8)
     [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    3    7
[4,]    4    8
```

Merging columns of matrices or data.frames

Try to merge these two tables using `cbind()`:

X1=

Id	GENDER	Weight
1	M	75
2	F	68
3	F	48
4	M	72

$\cup$

X2=

Id	GENDER	Height
1	M	182
2	F	165
3	F	160
4	M	178

What is the problem?

Now, try using the `merge()` function.

Merging columns of matrices or data.frames

```
> X
  GENDER Height Weight Income
1      F    165     50     80
2      M    182     65     90
3      M    178     67     60
4      F    160     55     50
```

```
> Y
  GENDER Height Weight Salary
4      F    165     55     70
5      M    182     65     90
6      M    178     67     40
7      F    160     85     40
```

```
> merge(X,Y,by=c("GENDER","Weight"))
  GENDER Weight Height.x Income Height.y Salary
1      F     55      160     50      165     70
2      M     65      182     90      182     90
3      M     67      178     60      178     40
```

```
> merge(X,Y,by=c("GENDER","Weight"),all=TRUE)
  GENDER Weight Height.x Income Height.y Salary
1      F     50      165     80       NA     NA
2      F     55      160     50      165     70
3      F     85       NA     NA      160     40
4      M     65      182     90      182     90
5      M     67      178     60      178     40
```

Merging lines of matrices or data.frames

The generic function is `rbind()`.

```
> rbind(1:4,5:8)
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
```

Merging lines of matrices or data.frames

```
> require(gtools)
> df1 <- data.frame(A=1:5, B=LETTERS[1:5])    # The square
                                              # brackets [] to
                                              # extract
> df2 <- data.frame(A=6:10, E=letters[1:5])   # elements will
                                              # be described
                                              # later.
> smartbind(df1, df2)
     A    B    E
1.1  1    A <NA>
1.2  2    B <NA>
1.3  3    C <NA>
1.4  4    D <NA>
1.5  5    E <NA>
2.1  6 <NA>    a
2.2  7 <NA>    b
2.3  8 <NA>    c
2.4  9 <NA>    d
2.5 10 <NA>    e
```

The function apply()

```
> ( X <- matrix(c(1:4, 1, 6:8), nr = 2) )
     [,1] [,2] [,3] [,4]
[1,]    1    3    1    7
[2,]    2    4    6    8
> apply(X, MARGIN=1, FUN=mean)
[1] 3 5
> apply(X, MARGIN=2, FUN=sum)
[1]  3  7  7 15
```

When the operation is summing or calculating the means of rows or columns, other possible functions are: `rowSums()`, `colSums()`, `rowMeans()`, `colMeans()`.

Perform the "**Do it yourself**" on page 93.

The function transform()

```
> ( X <- data.frame(Weight=c(80,75,60,52),Height=c(180,
+             170,165,150),Cholesterol=c(44,12,23,34),
+           Gender=c("Male","Male","Female","Female")) )
  Weight Height Cholesterol Gender
1     80    180           44   Male
2     75    170           12   Male
3     60    165           23 Female
4     52    150           34 Female
> ( X <- transform(X,Height=Height/100,
>            BMI=Weight/(Height/100)^2) )
  Weight Height Cholesterol Gender      BMI
1     80   1.80           44   Male 24.69136
2     75   1.70           12   Male 25.95156
3     60   1.65           23 Female 22.03857
4     52   1.50           34 Female 23.11111
```

Operations on lists: the function `lapply()`

```
> x <- list(a = 1:10, beta = exp(-3:3),
+       logic = c(TRUE,FALSE,FALSE,TRUE))
> lapply(x,mean) # Mean of each element of the list.
$a
[1] 5.5
$beta
[1] 4.535125
$logic
[1] 0.5
```

Operations on sets

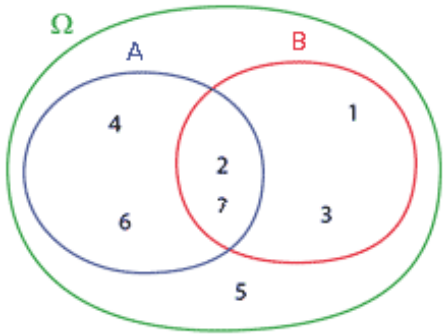


Table : Operations on sets.

Operation	R Instruction	Output
Membership: $a \in A$	`is.element(vec,A)`	`T F T`
Inclusion (subset): $A \subset B$	`all(A %in% B)`	`F`
Superset: $A \supset B$	`all(B %in% A)`	`F`
Intersection: $A \cap B$	`intersect(A,B)`	`2 7`
Union: $A \cup B$	`union(A,B)`	`4 6 2 7 1 3`
Complement: $A \backslash B$	`setdiff(A,B)`	`4 6`
Symmetric difference: $(A \cup B) \backslash (A \cap B)$	`setdiff(union(A,B),intersect(A,B))`	`4 6 1 3`

Create `A`, `B` and `vec <- c(2,3,7)` and play with these functions!

Extracting from vectors

```
> vec <- c(2,4,6,8,3)
> vec[2]
[1] 4
> "["(vec,2)          # Note: "[" is indeed a function.
[1] 4
> vec[-2]             # All elements except the second.
[1] 2 6 8 3
> vec[2:5]
[1] 4 6 8 3
> vec[-c(1,5)]
[1] 4 6 8
> vec[c(T,F,F,T,T)] # Extraction by logical mask.
[1] 2 8 3
> vec>4
[1] FALSE FALSE  TRUE  TRUE FALSE
> vec[vec>4]          # Extraction by logical mask.
[1] 6 8
```

Extracting from vectors

It is important to note here the syntactical simplicity of an instruction such as `x[y>0]`, which extracts from `x` all elements of index i such that $y_i > 0$.

```
> x <- 1:5
> y <- c(-1,2,-3,4,-2)
> x[y>0]
[1] 2 4
```

You need to learn to use as often as possible such constructions, which are called **logical masks**. There are two advantages: the code is easy to read, and execution is very fast.

Note: the functions `which()`, `which.min()` and `which.max()` are often very useful.

Replacement into vectors

```
> z
[1] 0 0 0 2 0
> z[c(1,5)] <- 1
> z
[1] 1 0 0 2 1
> z[which.max(z)] <- 0
> z
[1] 1 0 0 0 1
> z[z==0] <- 8   # The z_i such that
                 # z_i is worth 0 are replaced with
                 # 8.
> z
[1] 1 8 8 8 1
```

Insertion into vectors

```
> vecA <- c(1, 3, 6, 2, 7, 4, 8, 1, 0)
> vecA
[1] 1 3 6 2 7 4 8 1 0
> (vecB <- c(vecA, 4, 1))
 [1] 1 3 6 2 7 4 8 1 0 4 1
> (vecC <- c(vecA[1:4], 8, 5, vecA[5:9]))
 [1] 1 3 6 2 8 5 7 4 8 1 0

> a <- c()
> a <- c(a,2)
> a <- c(a,7)
> a
[1] 2 7
```

Extraction from matrices

Two methods are possible to extract elements from a matrix `X`. Each method has its own syntax.

1. *Extracting by indices:* `X[indr,indc]`, where `indr` is the vector of indices of rows and `indc` is the vector of indices of columns to extract. Omitting `indr` (respectively `indc`) means that all rows are selected (respectively all columns). Note that `indr` and `indc` can be preceded by a minus sign (-) to indicate elements not to extract.
2. *Extracting by logical mask:* `X[mask]`, where `mask` is a matrix of logical values `TRUE/FALSE` of the same size as `X`, indicating which elements to extract.

Extraction from matrices

```
> ( Mat <- matrix(1:12,nrow=4,ncol=3,byrow=TRUE) )
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
[4,]   10   11   12
> Mat[2,3]          # Extracting the element at the
                    # intersection of row 2 and column 3.
[1] 6
> Mat[,1]           # All rows, and only column 1.
[1]  1  4  7 10
> Mat[c(1,4),]      # All columns, and rows 1 and 4.
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]   10   11   12
> Mat[3,-c(1,3)]    # Row 3 and column 2.
[1] 8
```

Extraction from matrices (logical mask)

```
> MatLogical <- matrix(c(TRUE,FALSE),nrow=4,ncol=3)
> MatLogical       # Is of the same size as Mat.
      [,1]  [,2]  [,3]
[1,]  TRUE  TRUE  TRUE
[2,] FALSE FALSE FALSE
[3,]  TRUE  TRUE  TRUE
[4,] FALSE FALSE FALSE
> Mat[MatLogical] # Make sure that you understand this
                  # instruction.
[1] 1 7 2 8 3 9
```

Extraction from matrices

```
> m <- matrix(c(1,2,3,1,2,3,2,1,3),3,3)
> m
     [,1] [,2] [,3]
[1,]    1    1    2
[2,]    2    2    1
[3,]    3    3    3
> which(m == 1)          # m is seen as the concatenation
                         # of its columns.
[1] 1 4 8
> which(m == 1,arr.ind=TRUE) #Outputs indices as couples.
     row col
[1,]   1   1
[2,]   1   2
[3,]   2   3
```

Insertion into matrices

```
> m
     [,1] [,2] [,3]
[1,]    1    1    2
[2,]    2    2    1
[3,]    3    3    3
> m[m!=2] <- 0
> m
     [,1] [,2] [,3]
[1,]    0    0    2
[2,]    2    2    0
[3,]    0    0    0
> Mat <- Mat[-4,] ; Mat
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
> m[Mat>7] <- Mat[Mat>7]
> m
     [,1] [,2] [,3]
[1,]    0    0    2
[2,]    2    2    0
[3,]    0    8    9
```

Extracting from Lists

```
> ( L <- list(12,c(34,67,8),Mat,1:15,list(10,11))  )
[[1]]
[1] 12
[[2]]
[1] 34 67  8
[[3]]
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
[[4]]
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
[[5]]
[[5]][[1]]
[1] 10
[[5]][[2]]
[1] 11
> L[2]
[[1]]
[1] 34 67  8
> class(L[2])
[1] "list"
> L[c(3,4)]
[[1]]
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
[[2]]
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
```

Extracting from Lists

```
> L[[2]]
[1] 34 67  8
> "[["(L,2)
[1] 34 67  8
> class(L[[2]])
[1] "numeric"
> L[[5]][[2]]
[1] 11
> L <- list(cars=c("FORD","PEUGEOT"),climate=
+                          c("Tropical","Temperate"))
> L[["cars"]]
[1] "FORD"    "PEUGEOT"
> L$cars
[1] "FORD"    "PEUGEOT"
> L$climate
[1] "Tropical"  "Temperate"
```

Inserting into Lists

```
> L$climate[2] <- "Continental"
> L
$cars
[1] "FORD"     "PEUGEOT"
$climate
[1] "Tropical"    "Continental"
```

Manipulating character strings

```
> ( string <- c("one","two","three") )
[1] "one"   "two"   "three"
> as.character(1:3)
[1] "1" "2" "3"
> string1 <- c("a","ab","B","bba","one","!@","brute")
> nchar(string1)   # Counts the number of symbols in each string.
[1] 1 2 1 3 3 2 5
> string1[nchar(string1)>2]
[1] "bba"   "one"   "brute"
> string2 <- c("e","D")
> paste(string1,string2) # Concatenates the strings.
[1] "a e"     "ab D"    "B e"     "bba D"   "one e"   "!@ D"
[7] "brute e"
> paste(string1,string2,sep="-")  # A separator can be included
                                  # between the strings.
[1] "a-e"     "ab-D"    "B-e"     "bba-D"   "one-e"   "!@-D"
[7] "brute-e"
> paste(string1,string2,collapse="",sep="")  # collapse is used to
                                             # concatenate the elmts
                                             # into a single string.

[1] "aeabDBebbaDonee!@Dbrutee"
```

Manipulating character strings

```
> substring("abcdef",first=1:3,last=2:4)
[1] "ab" "bc" "cd"
> strsplit(c("05 Jan","06 Feb"), split=" ")
[[1]]
[1] "05"  "Jan"
[[2]]
[1] "06"  "Feb"
> grep("i",c("Pierre","Benoit","Rems"))
[1] 1 2
> gsub("i","L",c("Pierre","Benoit","Rems"))
[1] "PLerre" "BenoLt" "Rems"
> sub("r","L",c("Pierre","Benoit","Rems"))
[1] "PieLre" "Benoit" "Rems"
```

Goals of this section

Describing the instructions for

- control structures;
- creating basic functions.

if statements

```
Carbon <- fossilfuel$carbon
> if (mean(Carbon)>median(Carbon)) print("Mean > Median")
+     else print("Median <= Mean")
[1] "Mean > Median"
> dist <- c(148, 182, 173, 166, 109, 141, 166)
> dist.sort <- if (dist[1] < 150)
+     sort(dist, decreasing=TRUE) else sort(dist)
> dist.sort
[1] 182 173 166 166 148 141 109
```

Instructions if and else

```
> if (TRUE) 1+1
[1] 2
> x <- 2
> y <- -3
> if (x <= y) {
+     z <- y-x
+     print("x smaller than y")
+                 } else {
+     z <- x-y
+     print("x larger than y")
+     z
+                        }
[1] "x larger than y"
[1] 5
```

Selection and matching

- The operator %in% is used for testing set membership.

```
> x <- rep(1:5, rep(3,5))
[1] 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5
> x[x %in% c(2,4)]
[1] 2 2 2 4 4 4
```

- Use `match()` to find matching elements.

```
> match(x, c(2,4), nomatch=0)
[1] 0 0 0 1 1 1 0 0 0 2 2 2 0 0 0
```

Missing values

- Use `is.na()` to identify `NA`s.
- Use `complete.cases()` to return a logical vector whose length is the number of rows and whose TRUE values correspond to rows which do not contain any missing values.

```
> ## Which rows have missing values: data frame science
> science[!complete.cases(science), ]
     State  PrivPub  school  class   sex  like  Class
671    ACT   public      19      1  <NA>     5   19.1
672    ACT   public      19      1  <NA>     5   19.1
```

- The function `na.omit()` omits any rows that contain missing values.

```
> dim(science)
[1] 1385 7
> Science <- na.omit(science)
> dim(Science)
[1] 1383 7
```

Loops

- `for` loop

```
> for (i in 1:3) print(i)
[1] 1
[1] 2
[1] 3
```

- Other loops

```
repeat <expression> # Place break somewhere inside
while (x > 0) <expression> # Or (x < 0), or etc.
```

Here `<expression>` is an R statement, or a sequence of statements that are enclosed within braces.

Instruction for

```
> for (i in 1:3) print(i)
[1] 1
[1] 2
[1] 3
> x <- c(1,3,7,2)
> for (var in x) print(2*var)
[1] 2
[1] 6
[1] 14
[1] 4
```

Instruction while

```
> x <- 2
> y <- 1
> while(x+y<7) x <- x+y
> x
[1] 6
```

An example

```
> ## Relative population increase in Australian states:
> ## 1917-1997, data frame austpop (DAAG)
> relGrowth <- numeric(8) # numeric(8) creates a numeric
>          # vector with 8 elements, all set equal to 0
> for (j in seq(from=2, to=9)) {
+       relGrowth[j-1] <- (austpop[9, j]-austpop[1, j])
+       austpop[1, j]}
> names(relGrowth) <- names(austpop[c(-1,-10)])
> # We have used names() to name the elements of relGrowth

> relGrowth # Output is with options(digits=3)
 NSW   Vic   Qld    SA    WA   Tas    NT   ACT
2.30  2.27  3.98  2.36  4.88  1.46 36.40 102.33
```

User-written functions

```
mean.and.sd <- function(x){
    av <- mean(x)
    sdev <- sd(x)
    c(mean=av, SD=sdev)
    }
> distance <- c(148,182,173,166,109,141,166)
> mean.and.sd(distance)
  mean     SD
155.00  24.68
mean.and.sd <- function(x = rnorm(10)){
    av <- mean(x)
    sdev <- sd(x)
    c(mean=av, SD=sdev)
    }
> mean.and.sd()
      mean          SD
0.6576272  0.8595572
```

The structure of functions

function name — `mean.and.sd` — `<- function(` — *argument(s)* — `x=rnorm(10))`

```
{
av <- mean(x)
sdev <- sd(x)
c(av = av, sd = sdev)
}
```

function body

return value

The formula $BMI = \frac{Weight}{Height^2}$ is easily programmed in **R** as follows:

```
> BMI <- function(weight,height) {
+  bmi <- weight/height^2
+  names(bmi) <- "BMI"
+ return(bmi)
+ }
```

We can now execute the function `BMI()` we just created:

```
> BMI(70,1.82)
     BMI
21.13271
> BMI(1.82,70) # Note that it is not possible to swap the
               # arguments of a function,
         BMI
0.0003714286
> BMI(height=1.82,weight=70) # unless they are preceded by their
                             # names.
     BMI
21.13271
```

This function only outputs a single value.

The code below outputs a list of several variables.

```
> BMI <- function(weight,height) {
+  bmi <- weight/height^2
+  res <- list(weight,height,bmi)
+ return(res)
+ }
```

`BMI()` returns a list of unnamed elements:

```
> BMI(70,1.82)
[[1]]
[1] 70
[[2]]
[1] 1.82
[[3]]
[1] 21.13271
```

To name the elements of the list, you can use the following code

```
> BMI <- function(weight,height) {
+  bmi <- weight/height^2
+  res <- list(Weight=weight,Height=height,BMI=bmi)
+ return(res)
+ }
```

which gives the following result:

```
> BMI(70,1.82)
$Weight
[1] 70
$Height
[1] 1.82
$BMI
[1] 21.13271
```