ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ
ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

Introduction to R

Panagiotis Besbeas
Associate Professor in Applied Statistics
Department of Statistics
Athens University of Economics & Business
Email: besbeas@aueb.gr

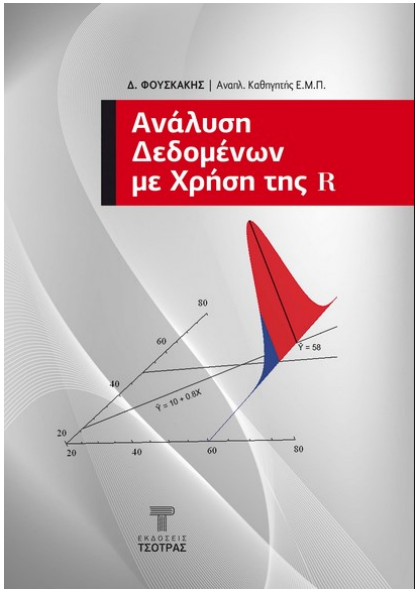
Acknowledgements

These notes are heavily based on previous slides by Prof I Ntzoufras @ AUEB

I am grateful to Prof Ntzoufras for making his notes available to me.

Acknowledgments cont'd

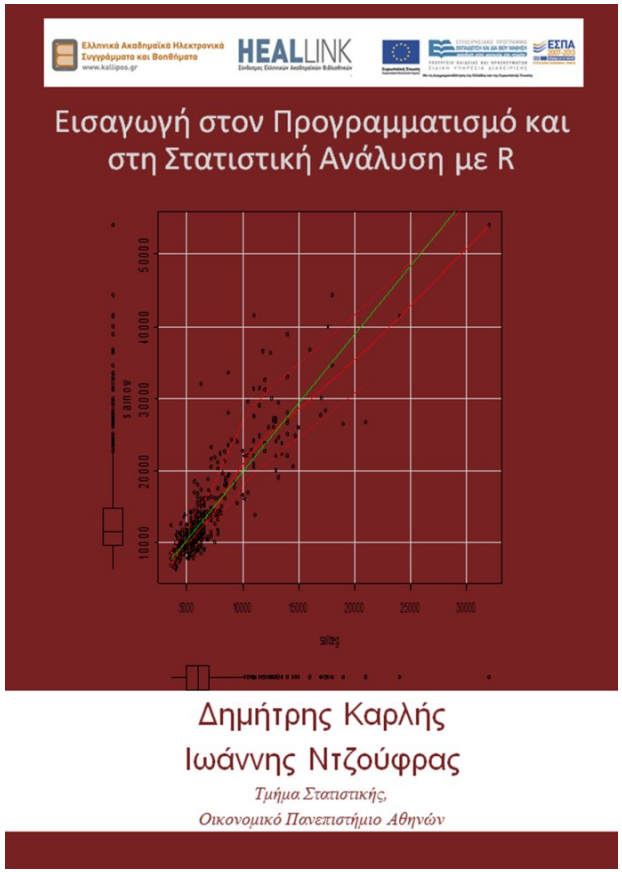
In turn, that material is based on previous slides by D. Fouskakis, Associate professor @ NTUA



Reference:
Δ. Φουσκάκης (2013). *Ανάλυση Δεδομένων με Χρήση της R*. Εκδόσεις Τσότρας. Αθήνα.

Also updated by the recent e-book

Ντζούφρας , Ι & Καρλής Δ. (2015).
Εισαγωγή στον Προγραμματισμό και στη Στατιστική Ανάλυση με R.
[e-book] Αθήνα: Σύνδεσμος Ελληνικών Ακαδημαϊκών Βιβλιοθηκών.
Διαθέσιμο στο: http://hdl.handle.net/11419/2601



What is R?

- R is an implementation of the S programming language.

What is **S**?

Quote from Venables and Ripley (2002, p1):

An **S** environment is an integrated suite of software facilities for data analysis and graphical display. Among other things it offers

- an extensive and coherent collection of tools for statistics and data analysis,
- a language for expressing statistical models and tools for using linear and non-linear models,
- graphical facilities for data analysis and display either at a workstation or as hardcopy,
- an effective object-oriented programming language that can easily be extended by the user community.

There are two major implementations of **S**

- **S-Plus**: Developed at Bell Laboratories (Lucent Technologies, formerly AT&T). Currently exclusively licensed to Insightful Corporation, which distributes an enhanced version.
  - Available for Unix/Linux and Windows. There is no Macintosh version available.
  - Allows many analyses to be run from a menu based system, but the full programming language is always available and is needed for some analyses.
- **R**: An open source (GNU) project originally started by Ross Ihaka (University of Auckland) and Robert Gentleman (Harvard Biostat).
  - Available for Unix/Linux, Windows and Macintosh.
  - No menu-based system for performing analysis. But see §Alternative Interfaces.

Much of the course discussing the **S** environment will be presented in **R** for the following reasons:

- Most things discussed will work in either environment.
- Tends to be faster and has better memory management.
- Faster development of add on libraries (and possibly more development).
- Runs on just about anything, as opposed to **S-Plus**.
- Easier to get up to date versions.

Both packages are extendable, with the built-in object oriented programming language and the abilities to incorporate C, C++, and Fortran routines.

R timeline

1991: Created in New Zealand by Ross Ihaka and Robert Gentleman.

Their experience developing R is documented in a 1996 JCGS paper.

1993: First announcement of R to the public.

1995: Martin Machler convinces Ross and Robert to use the GNU General Public License to make R free software.

1996: A public mailing list is created (R-help and R-devel).

1997: The R Core Group is formed (containing some people associated with S-PLUS). The core group controls the source code for R.

2000: R version 1.0.0 is released.

2026: Current R version is <*submit answer on Chat*>

The original R developers plotting world domination at the *Black Crow Cafe* on Kitchener Street.

Features of R

- Syntax syntax to S, making it easy for S-PLUS users to switch over.
- Runs on almost any standard computing platform/OS (even on the PlayStation 3)
- Frequent releases (annual + bugx releases); active development.
- Quite lean, as far as software goes; functionality is divided into modular packages.
- Graphics capabilities very sophisticated and better than most stat packages.
- Useful for interactive work, but contains a powerful programming language for developing new tools (user programmer)
- Very active and vibrant user community; R-help and R-devel mailing lists and Stack Overow.
- It’s free!!!!!

Drawbacks of R

- Essentially based on 40 year old technology.
- Little built in support for dynamic or 3-D graphics (but things have improved greatly since the "old days").
- Functionality is based on consumer demand and user contributions. If no one feels like implementing your favorite method, then it's your job! (Or you need to pay someone to do it).
- Objects must generally be stored in physical memory; but there have been advancements to deal with this too.

AVERAGE SALARY FOR High Paying Skills and Experience

SKILL	2013	YR/YR CHANGE
R	$ 115,531	n/a
NoSQL	$ 114,796	1.6%
MapReduce	$ 114,396	n/a
PMBok	$ 112,382	1.3%
Cassandra	$ 112,382	n/a
Omnigraffle	$ 111,039	0.3%
Pig	$ 109,561	n/a
SOA (Service Oriented Architecture)	$ 108,997	-0.5%
Hadoop	$ 108,669	-5.6%
Mongo DB	$ 107,825	-0.4%
SOX (Sarbanes-Oxley)	$ 107,697	4.8%
Jetty	$ 107,406	0.4%
UML (Unified Modeling Language)	$ 107,387	4.7%
OpenStack	$ 106,957	n/a
Big Data	$ 106,542	-6.3%
CMMI (Capability Maturity Model Integration)	$ 106,508	2.1%
3Par	$ 106,432	7.4%
FCoE (Fibre Channel Over Ethernet)	$ 106,062	2.7%
Cloudera	$ 105,677	n/a
Lean	$ 105,448	1.8%
Hbase	$ 105,295	n/a
Solaris	$ 104,710	3.7%
Jenkins	$ 104,461	n/a
Puppet	$ 103,925	n/a
ETL (Extract Transform and Load)	$ 103,830	-0.9%
Kanban	$ 103,381	0.7%
Waterfall	$ 103,160	-0.1%
Postgres	$ 103,146	6.4%
Nginx	$ 103,062	1.8%
Scrum	$ 102,955	-0.3%
Change Management	$ 102,889	1.8%
Load Balancers	$ 102,861	2.3%
Hive	$ 102,812	n/a
JDBC (Java Database Connectivity)	$ 102,803	-1.0%
Azure	$ 102,787	1.5%
ERP (Enterprise Resource Planning)	$ 102,757	2.4%
Balsamiq	$ 102,747	n/a
Objective C	$ 102,652	-2.2%
Amazon AWS	$ 102,573	3.0%
SDLC (System Development Life Cycle)	$ 102,361	1.8%
Korn Shell	$ 102,182	-1.5%
SOAP (Simple Object Access Protocol)	$ 102,131	3.0%
HP-UX	$ 102,066	-0.2%
EMC	$ 101,673	2.6%
JBoss	$ 101,646	1.1%
Weblogic	$ 101,643	-0.7%
Fortran	$ 101,553	-2.1%
Agile	$ 101,450	1.6%
Angular	$ 101,208	n/a
SaaS (Software as a Service)	$ 101,127	0.2%
TCL (Tool Command Language)	$ 101,102	2.0%

SKILL	2013	YR/YR CHANGE
SDN (Software-Defined Networking)	$ 101,026	n/a
Sybase	$ 100,868	6.0%
ITIL (Information Technology Infrastructure Library)	$ 100,746	0.8%
Perl	$ 100,656	3.3%
Knockout	$ 100,566	n/a
TOAD (Tool for Application Development)	$ 100,486	2.2%
C	$ 100,134	3.2%
ABAP (Advanced Business Application Development)	$ 100,024	3.4%
AIX	$ 99,995	-0.5%
iRise	$ 99,934	n/a
Bash	$ 99,783	6.7%
HL7 (Health Level 7)	$ 99,642	n/a
Tomcat	$ 99,435	1.7%
Netezza	$ 99,401	n/a
Oracle DB	$ 99,158	2.6%
Wan Opt	$ 99,111	2.5%
JIRA	$ 98,971	0.1%
Oracle eBusiness Suite	$ 98,967	2.7%
Microsoft Project	$ 98,950	1.7%
DOORS (Dynamic Object Oriented Requirements Management System)	$ 98,940	-0.4%
Business Intelligence	$ 98,691	-3.1%
Fibre Channel	$ 98,687	3.3%
JSP (Java Server Pages)	$ 98,657	0.4%
Data Warehouse	$ 98,536	-2.5%
NetApp	$ 98,209	3.2%
Rackspace	$ 98,145	n/a
Cloud Computing	$ 98,032	2.0%
PCI (Peripheral Component Interconnect)	$ 98,013	5.2%
MPLS (Multi Protocol Label Switching)	$ 97,989	3.6%
Shell	$ 97,883	3.7%
Six Sigma	$ 97,833	1.8%
Unix	$ 97,806	3.0%
Lighttpd	$ 97,792	1.3%
HP Eva	$ 97,767	n/a
CPOE (Computerized Physician Order Entry)	$ 97,757	7.7%
Disaster Recovery	$ 97,732	2.0%
Telepresence	$ 97,543	2.7%
Hitachi	$ 97,468	5.6%
Websphere	$ 97,453	-1.9%
Zookeeper	$ 97,405	n/a
Teradata	$ 97,359	-2.0%
EDI (Electronic Data Exchange)	$ 97,248	-2.7%
VSAM (Virtual Storage Access Method)	$ 97,232	5.1%
Labview	$ 97,118	n/a
Java/J2EE	$ 96,955	3.1%
SAP	$ 96,438	2.6%
Matlab	$ 96,248	5.0%
Metro Ethernet	$ 96,191	5.3%
Visio	$ 96,172	1.8%

NOTE: Several new tech skills were added to the 2013 survey and therefore yr/yr change is not available.

2014-2013

Dice Tech Salary Survey

Released January 29, 2014

Dice

Salaries and Confidence Rise for U.S. Tech Professionals

AVERAGE SALARY FOR High Paying Skills and Experien

SKILL	2013
R	$ 115,531
NoSQL	$ 114,796
MapReduce	$ 114,396
PMBok	$ 112,382
Cassandra	$ 112,382
Omnigraffle	$ 111,039
Pig	$ 109,561
SOA (Service Oriented Architecture)	$ 108,997
Hadoop	$ 108,669
Mongo DB	$ 107,825
SOX (Sarbanes-Oxley)	$ 107,697

SKILL
R

2013
$ 115,531

Design of the R System

- The R system is divided into 2 conceptual parts:
  - The "base" R system that you download from CRAN.
  - Everything else.

- CRAN is the "Comprehensive R Archive Network".

  It is a collection of sites which carry identical material, consisting of the R distribution(s), the contributed extensions, documentation for R, and binaries.

Design of the R System

- R functionality is divided into a number of packages.
- The "base" R system contains, among other things, the base package which is required to run R and contains the most fundamental functions.
- The other packages contained in the "base" system include utils, stats, datasets, graphics, grDevices, grid, methods, tools, parallel, compiler, splines, tcltk, stats4.

Design of the R System

- And there are many other packages available.
- There are about 11000 packages (!!!) on CRAN that have been developed by users and programmers around the world.
- People often make packages available on their personal websites; there is no reliable way to keep track of how many packages are available in this fashion.

Some R Resources

Available from CRAN

http://cran.r-project.org

- An Introduction to R
- Writing R Extensions
- R Data Import/Export
- R Installation and Administration (mostly for building R from sources)
- R Internals (not for the faint of heart)

Books

Standard Texts:

- Adler (2010). R in a Nutshell, O'Reilly.
- Albert (2007). Bayesian Computation with R, Springer.
- Albert & Rizzo (2011). R by Example, Springer.
- Chambers (2008). Software for Data Analysis: Programming with R, Springer.
- Crawley (2007). The R book, Wiley.
- Dalgaard (2002). Introductory Statistics with R, Springer – Verlag.
- Everitt & Hothorn (2006). A Handbook of Statistical Analyses using R, Chapman & Hall/CRC Press.
- Venables & Ripley (2002). Modern Applied Statistics with S, Springer.
- Murrell (2005). R Graphics, Chapman & Hall/CRC Press.

Books

Textbooks and Notes in Greek

- Φουσκάκης (2013). *Ανάλυση Δεδομένων με Χρήση της R*. Εκδόσεις Τσότρας.
- Καρλής & Ντζούφρας (2006). *Εισαγωγή στον Προγραμματισμό με R/Splus*. Πανεπιστημιακές σημειώσεις. Τμήμα Στατιστικής. Οικονομικό Πανεπιστήμιο Αθηνών.
- Φωκιανός & Χαραλάμπους (2010). *Εισαγωγή στην R: Πρόχειρες Σημειώσεις*. Τμήμα Μαθηματικών & Στατιστικής, Πανεπιστήμιο Κύπρου. Διαθέσιμο στην ιστοσελίδα: http://cran.r-project.org/doc/contrib/mainfokianoscharalambous.pdf .

Other resources:

- Springer has a series of books called Use R!.
- A longer list of books is at http://www.r-project.org/doc/bib/R-books.html.

Installing R

1. Visit the CRAN downloads page:

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

2. See “Installing R and Rstudio” files in STAT122/Misc on e-class.

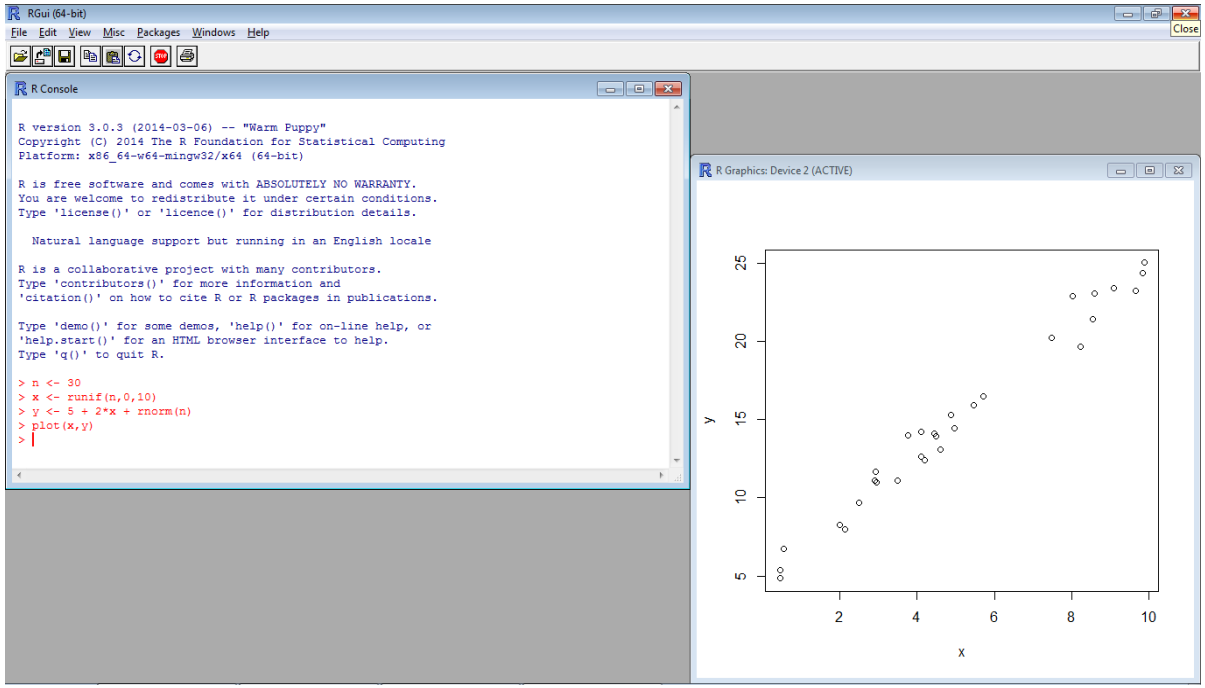
Starting R

Windows

- Click on Start → All Programs → R
- or double-click on the R icon on your desktop
- or double-click an .RData file in desired directory.

Linux or Unix

1. Change to the desired directory.
2. Type R <return>



Exiting/Interrupting R

- The Esc key or the stop-sign icon will terminate the current operation (Ctrl-C on Linux)
- `q()` will exit from R
- Exit via the File menu
- Close the window

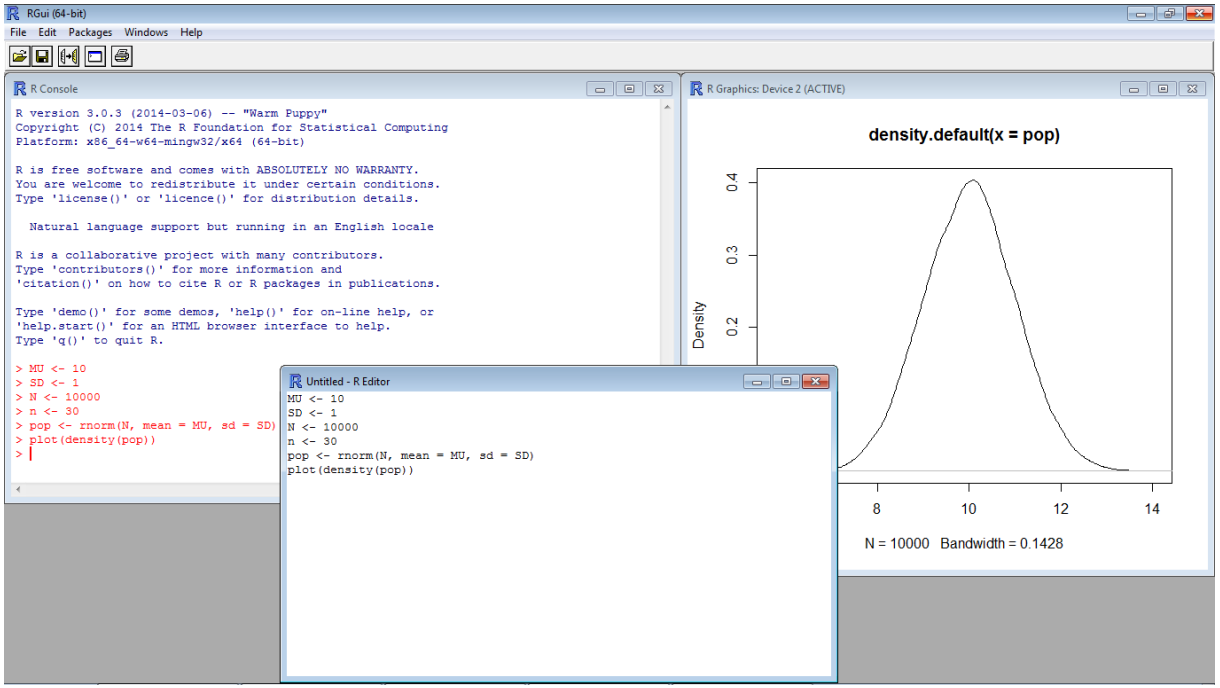
The R GUI

The default GUI includes the following windows:

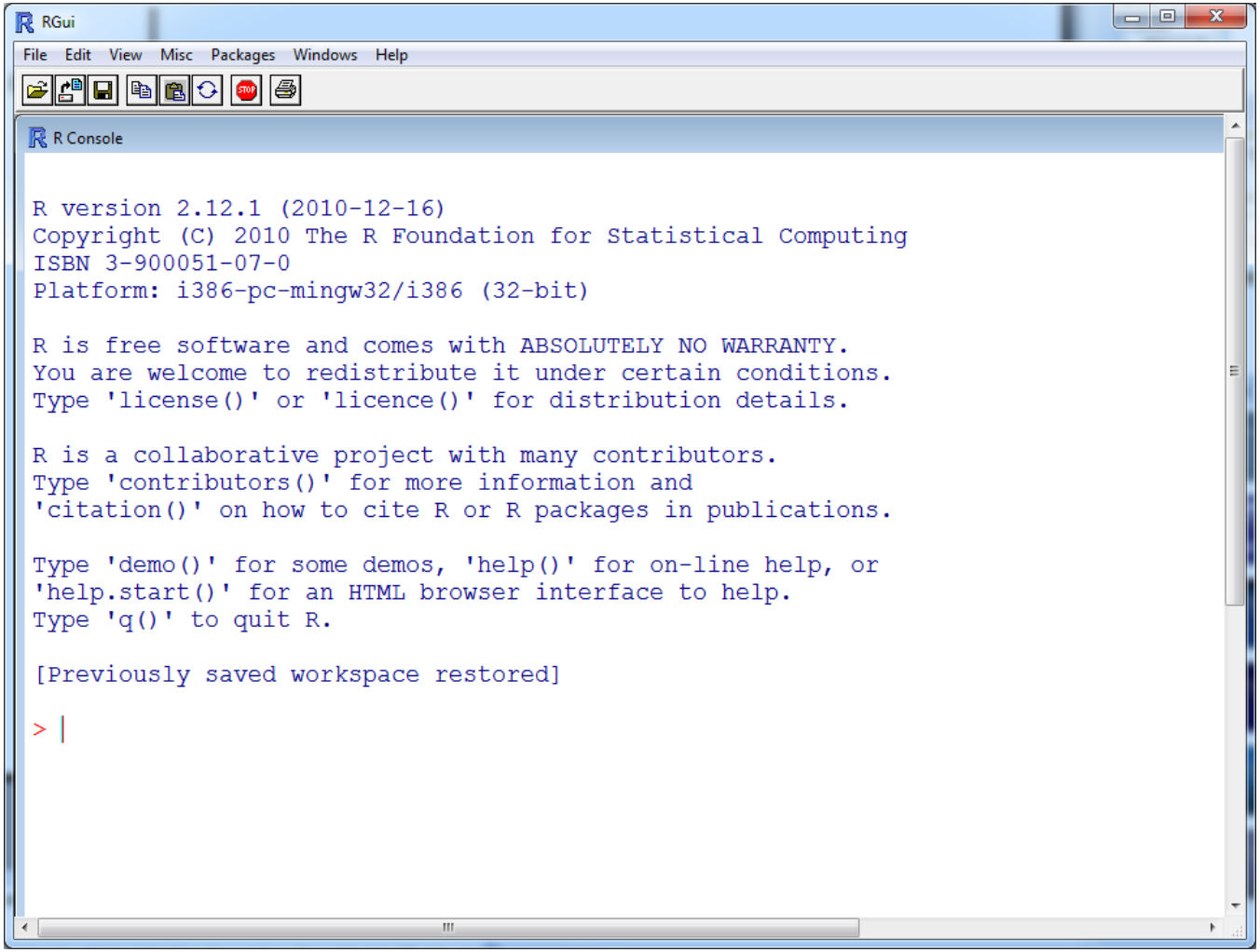
(i) R Console

(ii) R Graphics

(iii) R Editor



The R Console



R Console Basics

- The first line gives the R version number and name. We don't understand where the names come from!

- The important detail above is the symbol **>** which is the R default command prompt.

- R is an interactive system. The user types some input, and R returns an answer.

- User input consists of either *expressions* or *assignments*:

  - Expressions are evaluated, printed and the value is discarded.

  - An assignment also evaluates an expression but passes the value to variable or, in R terminology, an *object*; the value is not printed. There are several ways to enter an assignment, including = and the operator <− which is recommended.

A Brief Demo

```
> 2+3
[1] 5
> mean(rivers)
[1] 591.1844
> m <- mean(rivers); v <- var(rivers)
> sqrt(v)/m                                  # Coefficient of variation
[1] 0.8353922
```

Notes

1. The labeling `[1]` indicates that the answer is starting at the first element of a vector

2. `R` is *case-sensitive*, so that `v` and `V` are different variables. Practically any combination of letters, numbers, and symbols can be used to name a variable apart from some choices (e.g. for, if, `TRUE`, `FALSE`) which conflict with reserved function names.

3. We can see what objects are stored in our workplace by typing `objects()` or `ls()`.

4. Commands are separated either by a semi-colon, ***;***, or by a newline.

5. A command history is available using the arrow keys or typing `history()`.

6. The hash symbol, **#**, indicates a comment; everything on a line after this symbol is ignored.

7. If a command is not complete at the end of a line, **S** will give a different prompt, namely

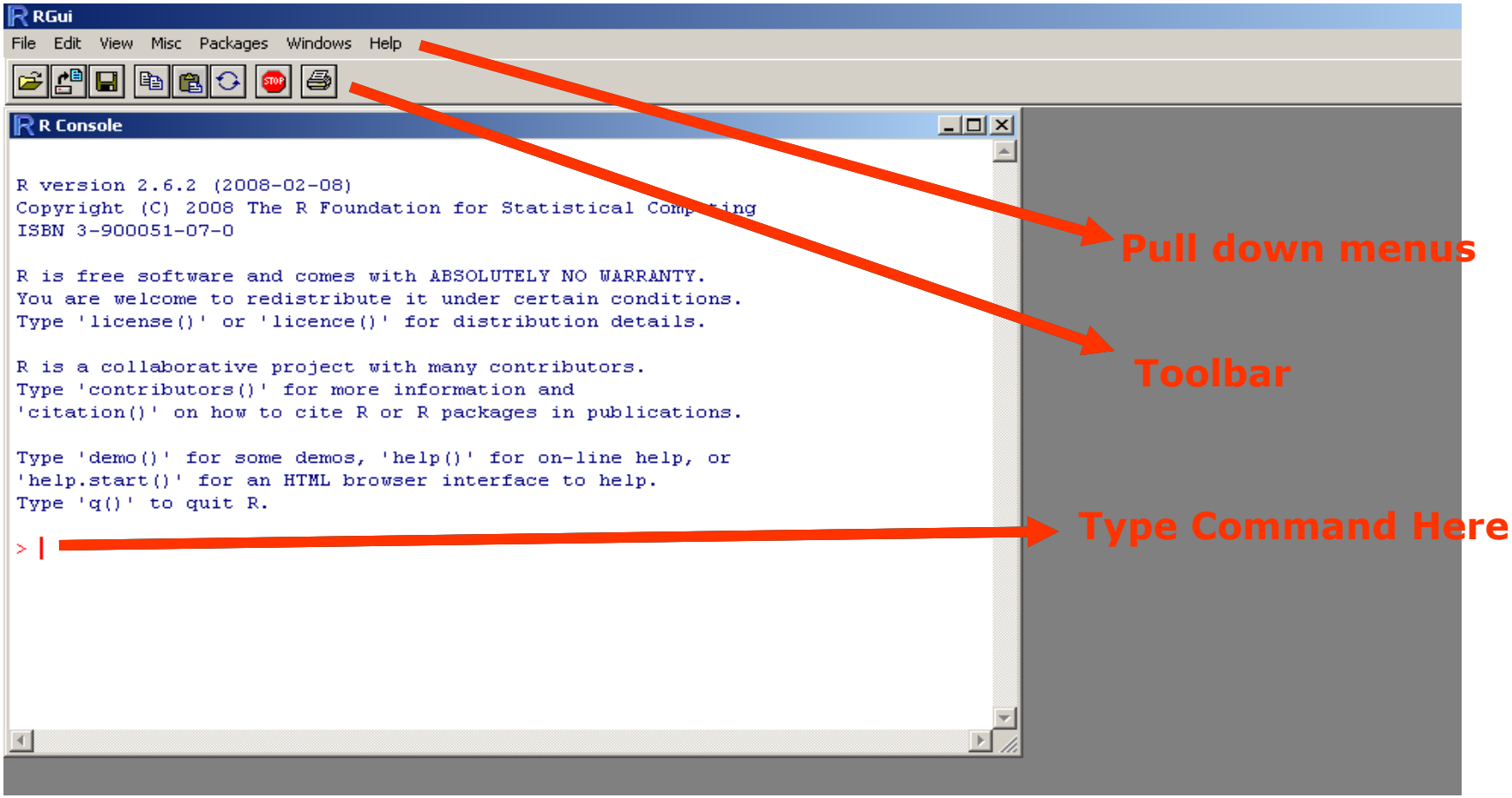
   +

   on subsequent lines, and continue to read input until the command is complete.

R Console

- We write R commands here.
- This window also displays all the commands R has run, the results, and any errors.
- This window appears when you launch R.
- We can type and run commands in this window line by line, or via the R Editor

The R console



The R default environment

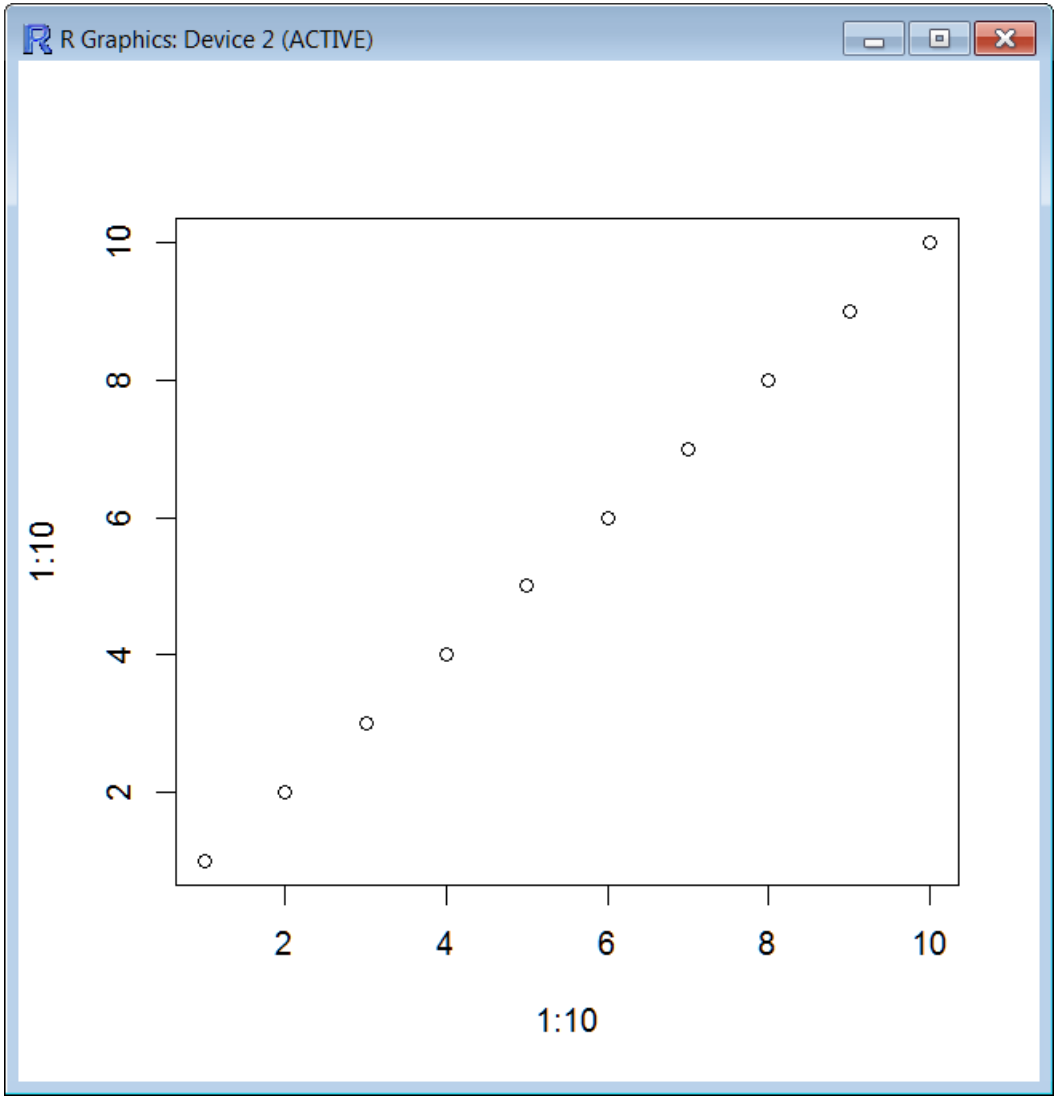
- The R environment is run with scripts
- In the long run it is more advantageous and flexible
- You can easily bring back up as you perform multiple calculations
- Here we will use the default graphical interface

The R Graphics window

- The R Graphics window is activated automatically when you create a graph.
- Graphs can be saved in various formats (e.g. jpg, png, bmp, ps, pdf).
- You can open multiple Graphics windows, using `dev.new()`, but only one window can be active at a time.

R graphics windows

```
> # New plot
> plot(1:10,1:10)
># New graphics window
> dev.new()
># close all graphics devices
> dev.off()
>
```



The R Editor

We have already seen the R Console and R Graphics windows. The R Editor facilitates working with R.

To open an R Editor window

From main menu, select File → New script.

To execute one line of the editor window

Position the cursor on the line and then press Ctrl-R to execute it.

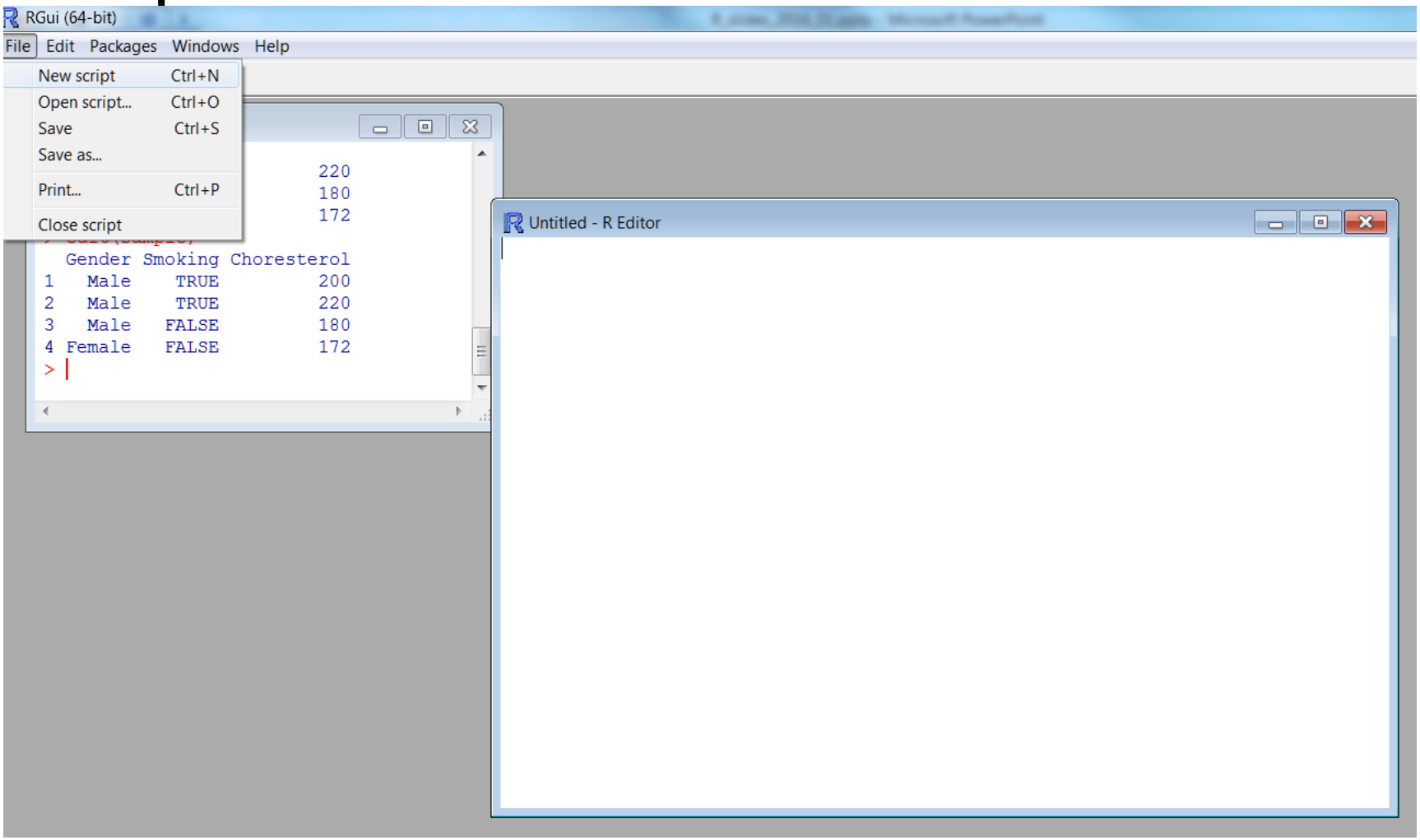
To execute several lines of the editor window

Highlight the lines using your mouse; then press Ctrl-R to execute them.

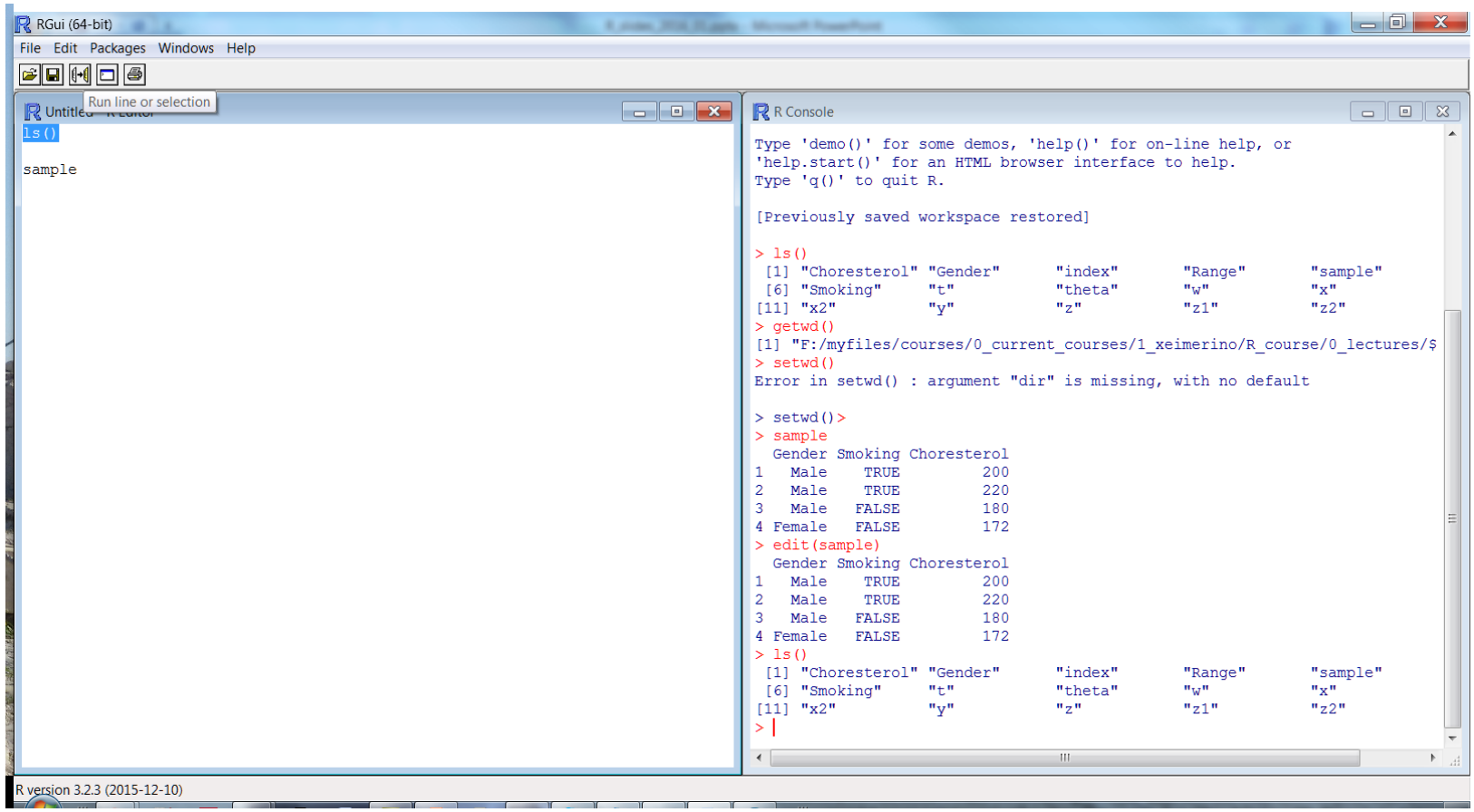
To execute the entire contents of the editor window

Press Ctrl-A to select the entire window contents and then Ctrl-R to execute them.

The R Editor


RGui (64-bit)
File Edit Packages Windows Help
New script Ctrl+N
Open script... Ctrl+O
Save Ctrl+S
Save as...
Print... Ctrl+P
Close script
220
180
172
Gender Smoking Choresterol
1 Male TRUE 200
2 Male TRUE 220
3 Male FALSE 180
4 Female FALSE 172
>
Untitled - R Editor

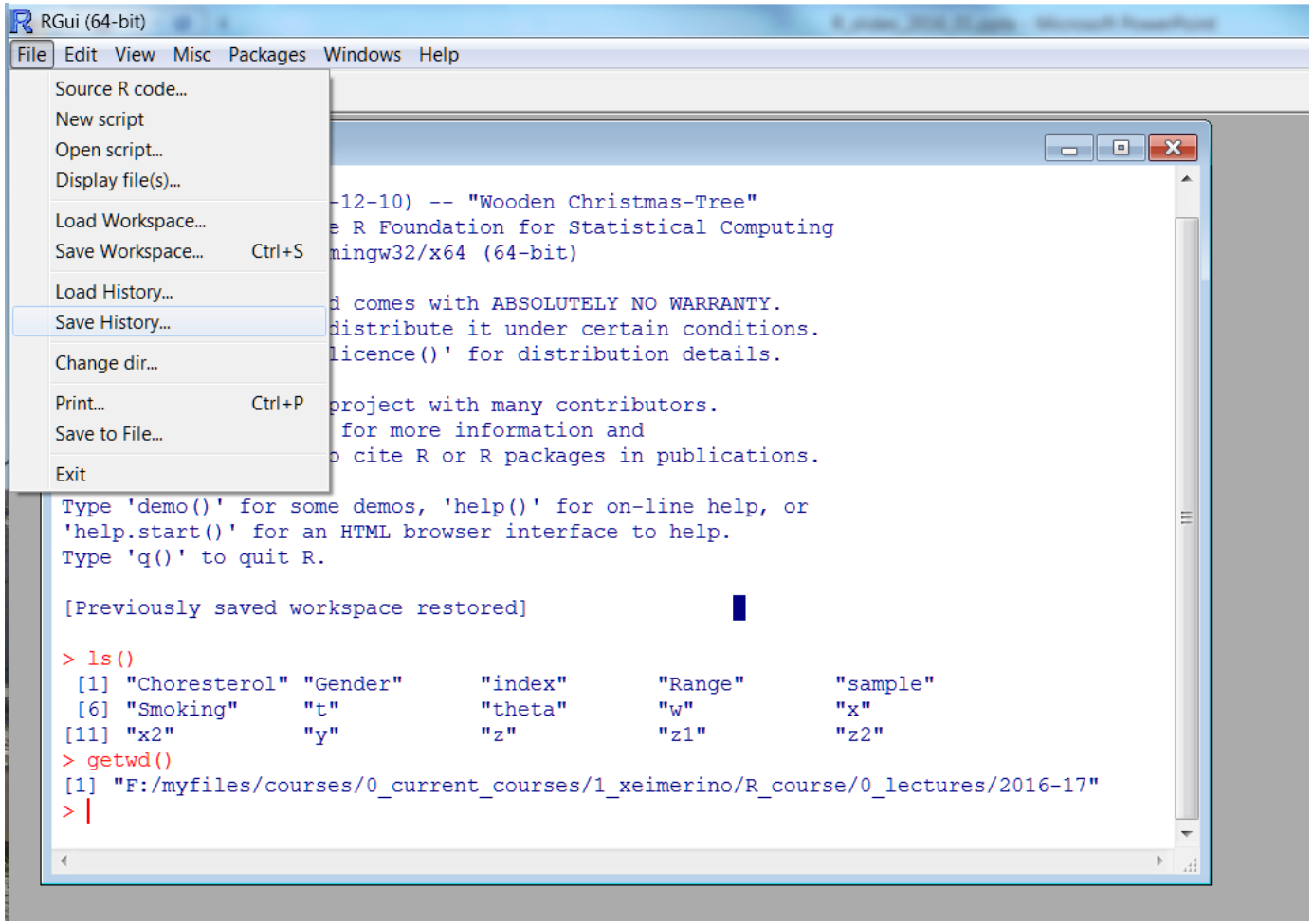
The R Editor


RGui (64-bit)
File Edit Packages Windows Help
Run line or selection
ls()
sample
R Console
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.
[Previously saved workspace restored]
> ls()
[1] "Choresterol" "Gender" "index" "Range" "sample"
[6] "Smoking" "t" "theta" "w" "x"
[11] "x2" "y" "z" "z1" "z2"
> getwd()
[1] "F:/myfiles/courses/0_current_courses/1_xeimerino/R_course/0_lectures/$
> setwd()
Error in setwd() : argument "dir" is missing, with no default
> setwd()>
> sample
Gender Smoking Choresterol
1 Male TRUE 200
2 Male TRUE 220
3 Male FALSE 180
4 Female FALSE 172
> edit(sample)
Gender Smoking Choresterol
1 Male TRUE 200
2 Male TRUE 220
3 Male FALSE 180
4 Female FALSE 172
> ls()
[1] "Choresterol" "Gender" "index" "Range" "sample"
[6] "Smoking" "t" "theta" "w" "x"
[11] "x2" "y" "z" "z1" "z2"
R version 3.2.3 (2015-12-10)

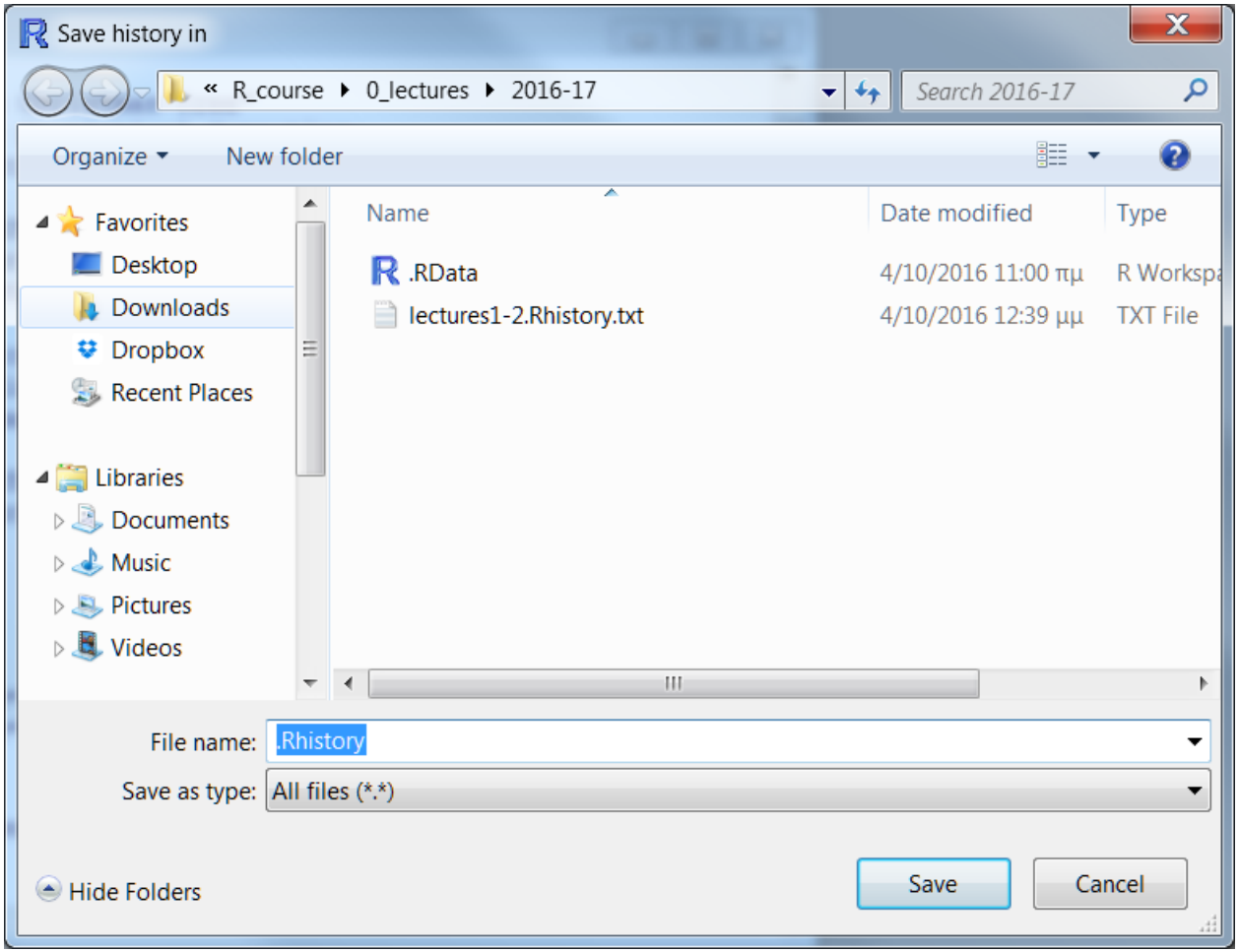
The R working directory

```
> getwd()
[1] "C:/Users/Administrator/Documents"
```

The R working directory



The R working directory



The R work space

We can change the working directory (and the workspace) with the function setwd()

We recommend to

1. Save one workspace (.Rdata file) for each project/assignment/analysis in a separate directory
2. Empty the workspace from all elements before you save it for first time
3. Open R using double click on the desired directory and workspace

Toolbar & Menu Bar

Like most window-based programs, R has a toolbar and a menu bar with pull-down menus that you can use to access some features of the program.

Toolbar & Menu Bar

Menu	*Functions*
File	Open source R code, create, open and save script, load and save workspace, load and save history, display files, change working directory, print files, and exit R.
Edit	Copy, paste, select all, clear console, data editor, R configuration window editor.
Misc	Stop current computation, buffer output, list objects in the memory, remove all objects, and list search path.
Packages / Packages & Data	Load, install, update packages, set CRAN mirror, select repositories, install packages, install packages from local zip files.
Window(s)	Cascade and tile R console windows. Arrange icons and switch among windows.
Help	Get help on R procedures, commands, and connect to the R website for more help information.

Getting Help

R provides help files for all its functions. To access a help file, use the Help pull-down menu,

Or type ? followed by a function name in the R console.

For example, to get help on the scan function, type:

```
> ?scan
```

Getting Help

```
> ?mean
> help(mean)
```

mean {base} R Documentation

Arithmetic Mean

Description

Generic function for the (trimmed) arithmetic mean.

Usage

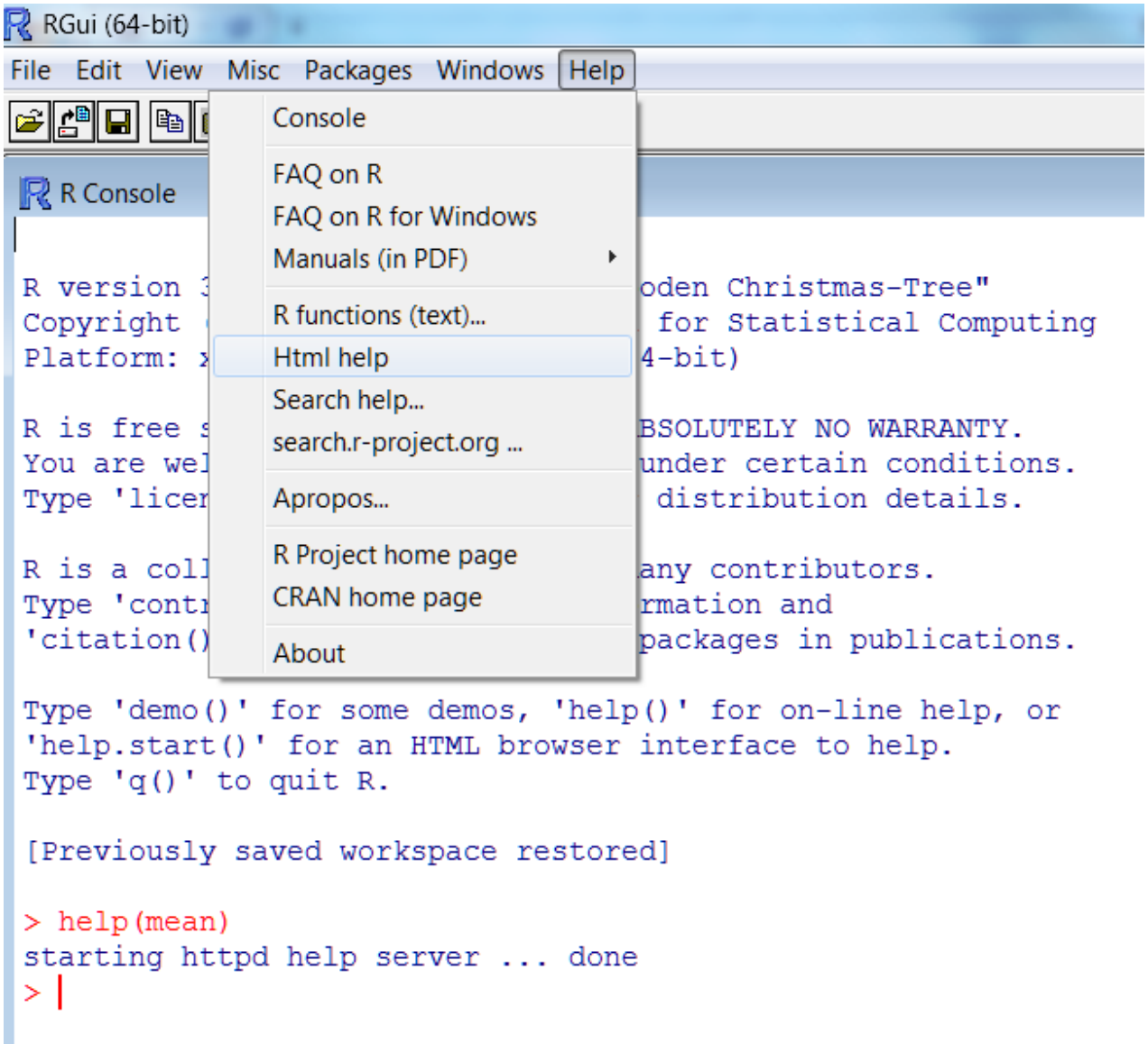
```
mean(x, ...)

## Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)
```

Arguments

`x`
: An R object. Currently there are methods for numeric/logical vectors and date, date-time and time interval objects. Complex vectors are allowed for `trim = 0`, only.

`trim`
: the fraction (0 to 0.5) of observations to be trimmed from each end

Getting Help



Getting Help

Statistical Data Analysis

Manuals

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

Reference

Packages
Search Engine & Keywords

Miscellaneous Material

About R
License
NEWS
Authors
Frequently Asked Questions
User Manuals
Resources
Thanks
Technical papers

Material specific to the Windows port

CHANGES up to R 2.15.0
Windows FAQ

Getting Help

Search Engine

Search

You can search for keywords, function and data names, concepts and within help page titles. The first search of a session will be slower.

Usage: Enter a string in the text field below and click the Search button or hit `RETURN`.

mean

Fields: ☑ Topics ☑ Titles ☑ Concepts ☐ Keywords

Options: ☑ Ignore case ☐ Fuzzy match

Types: ☑ Help pages ☑ Vignettes ☑ Demos

Search Reset

Concepts

Browse concepts available for searching the help system

Keywords

Getting Help

Search Results

The search string was **"mean"**

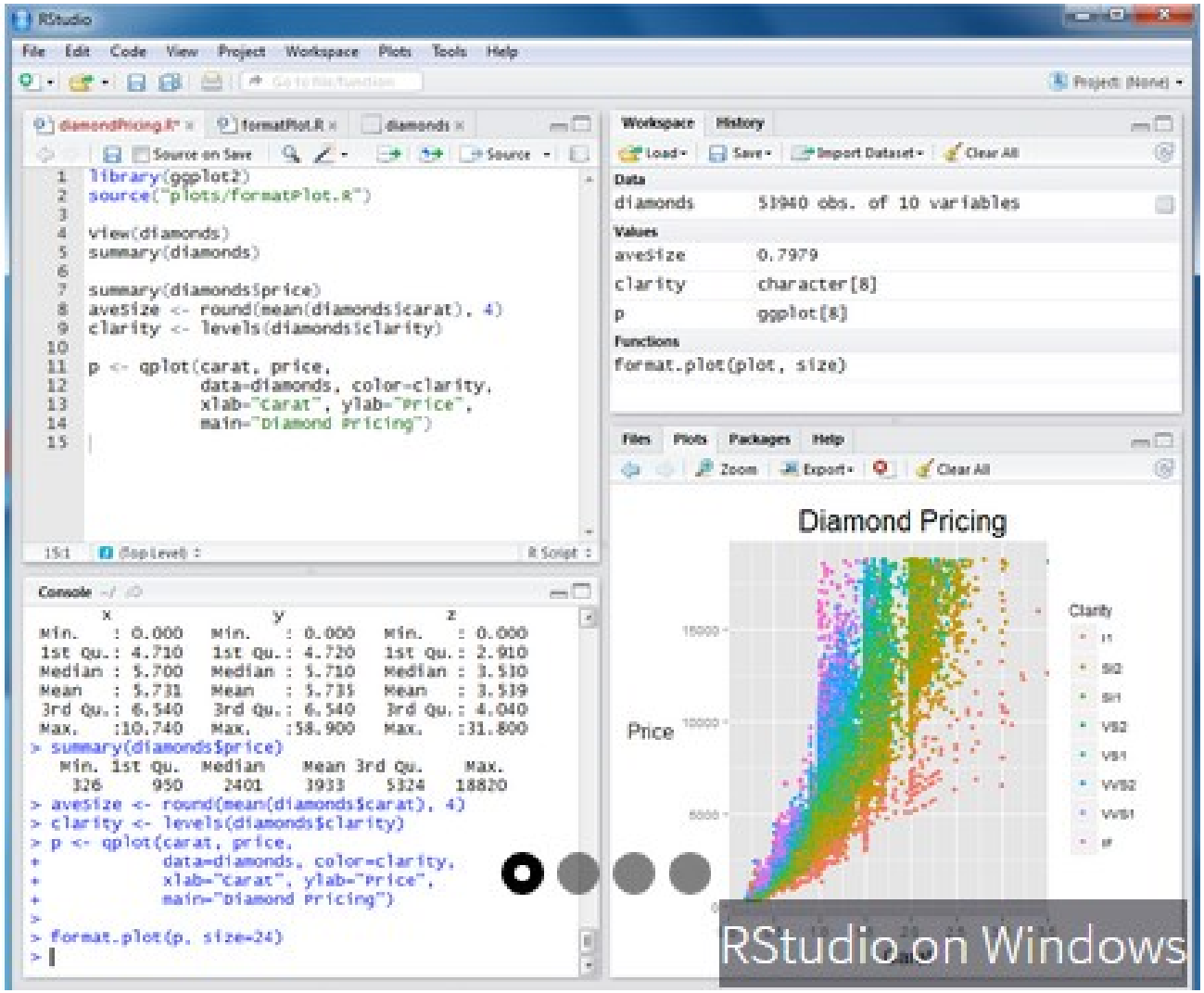
Help pages:

actuar::aggregateDist	Aggregate Claim Amount Distribution
actuar::mean.grouped.data	Arithmetic Mean
agricolae::bar.err	Plotting the standard error or standard deviance of a multiple comparison of means
agricolae::bar.group	Plotting the multiple comparison of means
agricolae::diffograph	Plotting the multiple comparison of means
base::colSums	Form Row and Column Sums and Means
base::Date	Date Class
base::DateTimeClasses	Date-Time Classes
base::difftime	Time Intervals
base::mean	Arithmetic Mean
bayesm::condMom	Computes Conditional Mean/Var of One Element of MVN given All Others
BiasedUrn::BiasedUrn-Univariate	Biased urn models: Univariate distributions
BiasedUrn::BiasedUrn-Multivariate	Biased urn models: Multivariate distributions
boa::boa.batchMeans	Batch Means
boot::sunspot	Annual Mean Sunspot Numbers

An alternative environment: R-studio

- The advanced R environment
- Monitor all windows simultaneously
- Many advantages

An alternative environment: R-studio



Notation & Common R Operators

> Indicates the prompt at the start of each new line in the R console.

The comment operator.

<- The assignment operator. Also by "="

== Boolean equality operator. It is used in logical statements, assessing whether two quantities are equal.

Examples

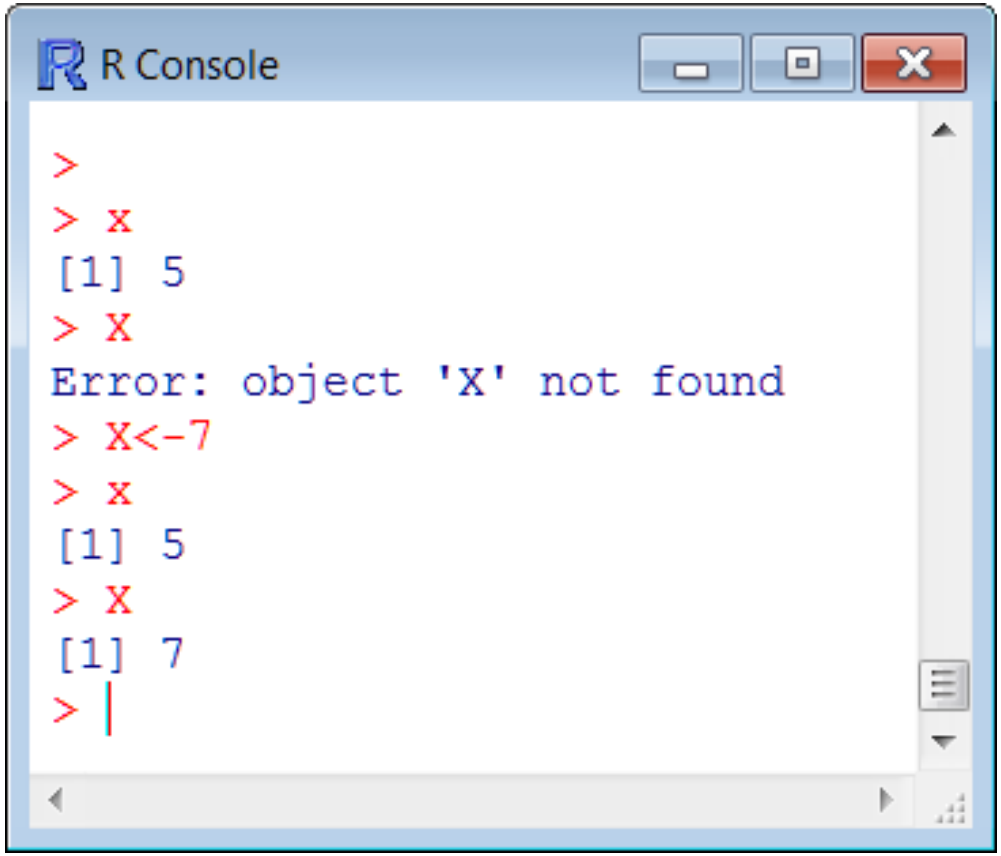
```
x <- 5   # assigns the value 5 to x
x == 5   # returns TRUE
# assigns the value 6 to x, overwriting the previous statement x <- 5.
x = 6
x == 5 # returns FALSE
```

Some important characteristics

- R is case sensitive
- In Console:
  - In every line we write a single command/syntax and we execute them by pressing enter
  - Multiple commands in a single line should be separated by “;”
- The menus and the icon bar are changing according the window you work

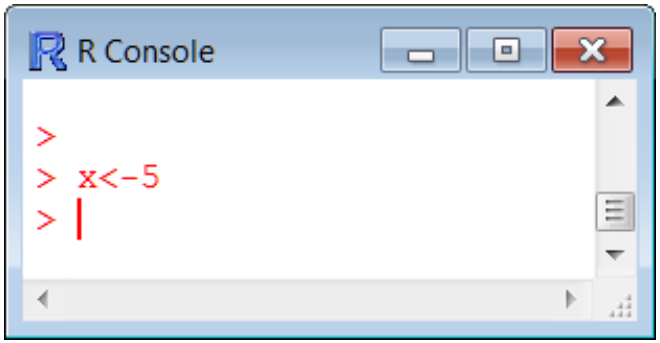
R is case sensitive

- R is case sensitive, i.e. x and X are different objects.

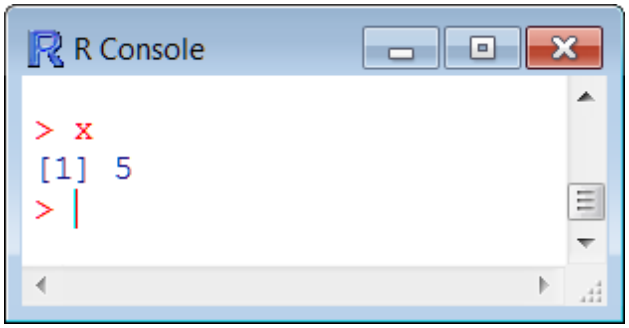


Small details

- When you use the assignment operator the result is not visible on screen.



- To see the value that your object took just type its name.



- Everything is assigned on objects (vectors, matrices, data frames, or lists) (more later).

Assignment, list and removal

```
> ls()
character(0)
> 5+6
[1] 11
> ls()
character(0)
> y
Error: object 'y' not found
```

```
> y<-5+6
> y
[1] 11
> ls()
[1] "y"
> rm(y)
> y
Error: object 'y' not found
> ls()
character(0)
>
```

Arithmetic Operators in R

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
^	Exponentiation
% / %	Integer Division
% %	Modulus

Examples

```
> 3+3  # this is my first command
[1] 6
> 9-2
[1] 7
> 17/3
[1] 5.666667
> 4*2
[1] 8
> 2^3
[1] 8
> 17%/%3
[1] 5
> 17%%3
[1] 2
> 7/0
[1] Inf
```

Notes:

1. Basic operations return a number. Logical operations (see later) return `TRUE` or `FALSE`

2. **S** uses the standard order of operation rules (BODMAS)

   - Parentheses can be used to intervene.

3. Logical expressions may be used in ordinary arithmetic, `FALSE` evaluating to `0` and `TRUE` evaluating to `1`.

Homework

1. Can you solve this?

$$9 - 3 \div \frac{1}{3} + 1$$

Use of parentheses

```
# Calculations and parentheses
> 4+3*6
[1] 22
> (4+3)*6
[1] 42
> (4+3)
[1] 7
> 4+3*6^2
[1] 112
> (4+3*6)^2
[1] 484
> 6^2
[1] 36
> 6^2*3
[1] 108
```

Positive and Negative Numbers

```
> +4
[1] 4
> -4
[1] -4
> 4+
+ 6
[1] 10
> 4+
+ 4+
+ 4+*5
Error: unexpected '*' in:
"4+
4+*"
> 4+*5
Error: unexpected '*' in "4+*"
```

```
> +4*5
[1] 20
> -4*5
[1] -20
> -4*-5
[1] 20
> (-4)*(-5)
[1] 20
> -4*+5
[1] -20
> sign(5)
[1] 1
> sign(-3)
[1] -1
```

Continuation prompt

If we send only part of a command to R, it will recognise that the command is incomplete by displaying the continuation prompt (the symbol "+" instead of ">").

- This prompt means R is waiting for the rest of the command.
- Otherwise, hit the ESC key to exit.

```
> 4+5/
+ 3
[1] 5.666667
```

Using objects in computations

We can save values to objects and use then in mathematical expressions or computations.

```
> r<-10
> 4+5/r
[1] 4.5
```

Using objects in computations

```
> # Illustration of %% and %/% using
objects a and b.
> a<-10
> b<-3
> a%%b
[1] 1
> a-b*(a%/%b)
[1] 1
> a%%b/b
[1] 0.3333333
> a<-140534
> b<-323
> a%%b
[1] 29
```

```
> a-b*(a%/%b)
[1] 29
> a%%b/b
[1] 0.08978328
```

Special Numeric values

R supports 4 special numeric values:

Inf: Infinity

-Inf: Minus infinity

NA: Not available

NaN: Not a Number

NULL: Null object in R (with no elements)

There are special functions available to check for these special values (see examples in the next slides).

Special Numbers Example

```
> exp(709)
[1] 8.218407e+307
> exp(710)
[1] Inf
> -exp(-709)
[1] -1.216781e-308
> -exp(710)
[1] -Inf
> 7/0
[1] Inf
> -1/0
[1] -Inf
> -10^1000
[1] -Inf
```

```
> exp(-Inf)
[1] 0
> log(-2)
[1] NaN
> 0/0
[1] NaN
> x<-1
> names(x)
[1] NULL
> 7/0-7/0
[1] NaN
> 7/0-1
[1] Inf
```

Checking for special values

Special logical functions that check for the existence of values:

is.infinite:	Infinity
is.na:	Not available
is.nan:	Not a Number
is.null:	Null object in R (with no elements)

Checking for special values

```
> is.null(x)
[1] FALSE
> is.null(names(x))
[1] TRUE

> is.infinite(exp(700))
[1] FALSE
> is.infinite(exp(710))
[1] TRUE
> is.na(exp(710))
[1] FALSE
> is.nan(exp(710))
[1] FALSE
```

```
> is.na(7/0)
[1] FALSE
> x<-0/0
> x
[1] NaN
> is.na(x)
[1] TRUE
> is.nan(x)
[1] TRUE
> x<-NA
> is.na(x)
[1] TRUE
> is.nan(x)
[1] FALSE
>
```

Checking for special values

```
> x<-c(0,Inf,-Inf, NaN, NA)
> is.finite(x)
[1] TRUE FALSE FALSE FALSE FALSE
> is.infinite(x)
[1] FALSE TRUE TRUE FALSE FALSE
> is.nan(x)
[1] FALSE FALSE FALSE TRUE FALSE
> is.na(x)
[1] FALSE FALSE FALSE TRUE TRUE
```

Basic Mathematical Functions in R

Function	Description	Function	Description
sqrt()	Square root	asin()	Inverse sine
abs()	Absolute value	atan()	Inverse tangent
log()	Natural logarithm (ln)	gamma()	Gamma function
log2()	Logarithm base 2	lgamma()	Natural logarithm of gamma function
log10()	Logarithm base 10	beta()	Beta function
exp()	Exponential function	floor()	Previous integer
cos()	Cosine	ceiling()	Next integer
sin()	Sine	factorial()	Factorial
tan()	Tangent	choose()	Combinations
acos()	Inverse cosine	lchoose()	Natural logarithm of combinations

Examples

```
> sqrt(16)
[1] 4
> abs(-2)
[1] 2
> log(10)
[1] 2.302585
> log2(10)
[1] 3.321928
> log10(10)
[1] 1
> exp(3)
[1] 20.08554
> cos(pi)
[1] -1
> sin(2*pi)
[1] -2.449213e-16
> tan(pi/2)
[1] 1.633178e+16
```

```
> sin(pi/2)
[1] 1
> tan(0)
[1] 0
> acos(0.2)
[1] 1.369438
> atan(2)
[1] 1.107149
> asin(0)
[1] 0
> gamma(2)
[1] 1
> beta(2,3)
[1] 0.08333333
```

```
> lgamma(4)
[1] 1.791759
> floor(4.9)
[1] 4
> ceiling(4.1)
[1] 5
> factorial(5)
[1] 120
> choose(5,2)
[1] 10
> lchoose(5,2)
[1] 2.302585
```

Logarithm

log {base} R Documentation

Logarithms and Exponentials

Description

`log` computes logarithms, by default natural logarithms, `log10` computes common (i.e., base 10) logarithms, and `log2` computes binary (i.e., base 2) logarithms. The general form `log(x, base)` computes logarithms with base `base`.

`log1p(x)` computes *log(1+x)* accurately also for $|x| << 1$.

`exp` computes the exponential function.

`expm1(x)` computes *exp(x) - 1* accurately also for $|x| << 1$.

Usage

```
log(x, base = exp(1))
logb(x, base = exp(1))
log10(x)
log2(x)

log1p(x)

exp(x)
```

Examples for log

```
> log(10)
[1] 2.302585
> log(10,2)
[1] 3.321928
> log(10,10)
[1] 1
> exp(2.302585)
[1] 9.999999
> exp(log(10))
[1] 10
> log2(10)
[1] 3.321928
> log10(10)
[1] 1
```

Examples for factorial

```
> factorial(10)
[1] 3628800
> factorial(1000)
[1] Inf
> factorial(1000)/factorial(999)
[1] NaN
> sum( log( 1:1000 ))
[1] 5912.128
> lfactorial(999)
[1] 5905.22
> lfactorial(1000)-lfactorial(999)
[1] 6.907755
> exp(lfactorial(1000)-lfactorial(999))
[1] 1000
>
```

Examples for choose

```
> choose(5,2)
[1] 10
> lchoose(5,2)
[1] 2.302585
> exp(lchoose(5,2))
[1] 10
>
> factorial(5)/(factorial(2)*factorial(5-2))
[1] 10
> lfactorial(5)-lfactorial(2)-lfactorial(5-2)
[1] 2.302585
> exp(lfactorial(5)-lfactorial(2)-lfactorial(5-2))
[1] 10
```

Examples for choose (2)

```
> choose(1000,10)
[1] 2.634096e+23
> choose(10000,10)
[1] 2.743355e+33
> choose(100000,10)
[1] 2.754492e+43
> choose(100000,100)
[1] Inf
> lchoose(100000,100)
[1] 787.5037
```

Using objects in computations

- Suppose we want to compute the quantity

$$-\frac{\log(1-\left(1-e^{-\theta}\right)\mathrm{e}^{-t})}{\theta}$$

  for θ=3 and t=5.

- Which of the following commands is correct?

```
> theta <- 3; t <- 5
> -log( 1-(1-exp(theta)*exp(-t)) )/theta          #try1
[1] 0.6666667
> -log( 1-(1-exp(-theta)*exp(-t)) )/theta         #try2
[1] 2.666667
> -log( 1-(1-exp(-theta))*exp(-t) )/theta         #try3
[1] 0.002141023
```

Using objects in computations

- Suppose we want to compute the quantity

$$-\frac{\log(1-\left(1-e^{-\theta}\right)\mathrm{e}^{-t})}{\theta}$$

  for θ=3 and t=5.

- Which of the following commands is correct?

```
> theta <- 3; t <- 5
> -log( 1-(1-exp(theta)*exp(-t)) )/theta      #try1
[1] 0.6666667
> -log( 1-(1-exp(-theta)*exp(-t)) )/theta     #try2
[1] 2.666667
> -log( 1-(1-exp(-theta))*exp(-t) )/theta     #CORRECT
[1] 0.002141023
```

How assignment works - updating an object

```
> x <- 5*3                    #assignment
> x
[1] 15
> x*10
[1] 150
> x+ x                        #expression
[1] 30
> x <- x + x                  #assignment (updating value)
> x
[1] 30
> x <- x + x                  #assignment (updating value)
> x
[1] 60
> x <- x + x                  #assignment (updating value)
> x
[1] 120
> x <- 10                     #assignment
> x
[1] 10
```

Assigning values to default objects

```
> pi                                  #built-in constant
[1] 3.141593
> pi<- x                              #assignment
> pi
[1] 10
> ls()
[1] "pi"                "x"
> rm(pi)                              #remove object pi
> ls()
[1] "x"
> pi
[1] 3.141593
> # default objects cannot be removed
> rm(pi)
Warning message:
In rm(pi) : object 'pi' not found
```

Assigning values to default objects

```
> sum <- 50
> sum
[1] 50
> sum(1:10)
[1] 55
> rm(sum)
> sum
function (..., na.rm = FALSE)  .Primitive("sum")
> rm(sum)
Warning message:
In rm(sum) : object 'sum' not found
```

Logical Operators

Operator	Description
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
==	Exactly equal to
!=	Not equal to
!x	Not x
x|y	x OR y
x&y	x AND y

Examples

```
> x<-56
> x
[1] 56
> 5*2->y
> y
[1] 10
> x>y
[1] TRUE
> y<4
[1] FALSE
```

```
> x>=56
[1] TRUE
> y<=9
[1] FALSE
> x==56
[1] TRUE
> y!=1
[1] TRUE
> !(5>3)
[1] FALSE
```

More on Logical Operators

We usually use the symbols & (AND), | (OR) to combine two or more logical operators.

```
> (5>3) & (8>10)
[1] FALSE
> (5>3) | (8>10)
[1] TRUE
```

Logical operators are frequently used in loops, e.g. if, for, etc. (more later).

Examples

```
> x<-5
> y<-10
> x>y
[1] FALSE
> x<y
[1] TRUE
> !(x<y)
[1] FALSE
> x<y
[1] TRUE
> (x<y) & (5!=4)
[1] TRUE
> (x<y) & (5==4)
[1] FALSE
> (x<y) | (5==4)
[1] TRUE
```

```
> x
[1] 5
> (x>8)
[1] FALSE
> (x<5)
[1] FALSE
> (x>8) | (x<5)
[1] FALSE
```

General Functions in R

- **builtins()**: built-in functions in R.
- **cat()** or **print():** Print on screen.
- **ls()**: list all objects
- **rm()**: remove objects.
- **getwd()**: returns working directory.
- **setwd()**: changes working directory.
- **list.files()**: returns a list with all files in working directory.
- **date()** or **Sys.time()**: Returns current day & time.

Data Types in R

There are four atomic data types in R:

- Numeric

```
> value <- 605
[1] 605
```

- Character

```
> string <- "Female"
[1] "Female"
```

- Logical

```
> logi <- 2 < 4
[1] TRUE
```

- Complex number

```
> cn <- 2+3i
[1] 2+3i
```

Type of object elements in R

We can use the function mode() check the type of the elements of an object.

Type of object elements in R

```
x1 <- 3
mode(x1)
[1]  "numeric"

x2 <-  x1>4
x2
[1] FALSE
mode(x2)
[1]  "logical"
```

```
x3 <- 'Ioannis'
x3
[1] "Ioannis"
mode(x3)
[1] "character"

x1 <- 4+3i
x1
[1] 4+3i
mode(x1)
[1] "complex"
```

Data Objects in R

The basic data objects in R are:

(i) vector

(ii) matrix

(iii) array

(iv) list

(v) data frame

Some objects contain only one type of data (e.g. vector, matrix, array), others can support a mix of types (data frame, list).

We shall provide details about the different types next, starting with vectors.

Type of data objects in R

We use class() to obtain the type of data object in R.

Class vs. mode

Function mode will give

- Numeric (real number)
- Complex
- Logical
- Character

If the object has elements of the same type (i.e. for vectors, matrices and arrays).

For lists/data.frames the mode is list

Class vs. mode

Function class is more detailed and it gives

- The same as mode for vectors (except for factors)
- The names of the object type for matrices, arrays, lists and data.frames

Class vs. mode

```
> x<-1:10
> mode(x); class(x)
[1] "numeric"
[1] "integer"
>
> x<- 1:10<5
> mode(x); class(x)
[1] "logical"
[1] "logical"
>
```

```
> x<- c('Yiannis','Grigoris')
> mode(x); class(x)
[1] "character"
[1] "character"
>
> x<- 1+5i
> mode(x); class(x)
[1] "complex"
[1] "complex"
>
```

Class vs. mode

```
> x <- matrix(1:10,2)
> mode(x); class(x)
[1] "numeric"
[1] "matrix"
>
> x <- array(1:8, c(2,2,2))
> mode(x); class(x)
[1] "numeric"
[1] "array"
```

```
> x <- list(x=1:10)
> mode(x); class(x)
[1] "list"
[1] "list"

> x <- data.frame(x=1:10)
> mode(x); class(x)
[1] "list"
[1] "data.frame"
```

Vectors

Vectors are a central component of R. A vector is a collection of items of the same type or, in R terminology, the same *mode*. Thus a vector can contain either numbers (mode=numeric), strings (character), or logical values (logical) but not a mixture. The main function for creating a vector is the function **c**.

```
> c(0,1,1,2,3,5,8,13,21)
[1] 0 1 1 2 3 5 8 13 21
> c("Female","Male","Female","Female")
[1] "Female" "Male" "Female" "Female"
> c(TRUE,TRUE,FALSE,TRUE)
[1] TRUE TRUE FALSE TRUE
```

Vectors

The same function can be used to concatenate already defined vectors:

```
> x <- c(0,1,1,2,3,5,8,13,21)
[1] 0 1 1 2 3 5 8 13 21
> y <- c(34, 55, 89, 144, 233, 377)
[1] 34  55  89 144 233 377
> z <- c(x, y)
[1] 0   1   1   2   3   5   8  13  21  34  55  89 144 233 377
```

Vectors cannot mix different data types. If we do, R tries to accommodate this by converting one of them:

```
> c(1, "a")
[1] "1" "a"
> c(1, TRUE, FALSE)
[1] 1 1 0
```

Vectors (and lists, as we shall see) can have *names*—each element (or list component) has a name. The following two commands are equivalent, only operating at creation and post creation respectively:

```
> whitehouse <- c(Kennedy=1961, Johnson=1963, Carter=1977, Clinton=1993)
> whitehouse
Kennedy Johnson  Carter Clinton
   1961    1963    1977    1993

> whitehouse2 <- c(1961, 1963, 1977, 1993)
> names(whitehouse2) <- c("Kennedy", "Johnson", "Carter", "Clinton")
> whitehouse2
Kennedy Johnson  Carter Clinton
   1961    1963    1977    1993
```

Functions for vectors

- length(x): length of the vector (number of elements of the vector)
- names(x) : extract or give a name to each element of the vector
- mode(x) : type of vector (i.e. numeric, logical, etc.)
- Vector functions (or statistical): Input is a vector. The function uses all values of the vector and the result is (usually) one value

mean, median, sd, min, max, sum, prod

Functions for vectors

- length(x): length of the vector

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

Functions for vectors

- names(x) : extract or give a name to each element of the vector

```
> height<-c(1.75,1.84,1.81,1.63)
> names(height)
NULL
> names(height)<-c("Jim","George","John","Mary")
> names(height)
[1] "Jim" "George" "John" "Mary"
> height
Jim George John Mary
1.75   1.84   1.81   1.63
> names(height) <- NULL # removes the names
```

Performing Vector Arithmetic

Vector operations are one of R's great strengths. The usual arithmetic/logical operators can be applied to vectors, operating in an element-wise manner. Most functions operate element-wise on vectors too, and return a vector result.

```
> fib <- c(0, 1, 1, 2, 3, 5, 8, 13, 21)
> fib + 2
[1]  2  3  3  4  5  7 10 15 23
> fib > 4
[1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE
> 2^fib
[1]       1       2       2       4       8      32     256    8192
[9] 2097152
> log(fib)
[1]      -Inf 0.0000000 0.0000000 0.6931472 1.0986123 1.6094379 2.0794415
[8] 2.5649494 3.0445224
```

This vectorization has two great advantages. The first and most obvious is convenience. The second is speed. Most vectorized operations are implemented directly in C code, and so are substantially faster than writing an explicit loop in `R`.

Some useful functions for vectors:

`min`, `max`, `range`, `length`, `sum`, `prod`, `sort` and **`order`.**

Examples of vector operations

```
> x<-1:10
> y<-c(2,2,2,2,2,3,3,3,3,3)
> x
[1] 1 2 3 4 5 6 7 8 9 10
> y
[1] 2 2 2 2 2 3 3 3 3 3
> x+y
[1] 3 4 5 6 7 9 10 11 12 13
> x-y
[1] -1 0 1 2 3 3 4 5 6 7
> x^y
[1] 1 4 9 16 25 216 343 512 729 1000
```

Examples of vector operations

```
> x*y
[1] 2 4 6 8 10 18 21 24 27 30
> x/y
[1] 0.500000 1.000000 1.500000 2.000000 2.500000
[6] 2.000000 2.333333 2.666667 3.000000 3.333333
> (x^2)/(y-5)
[1] -0.3333333 -1.3333333 -3.0000000 -5.3333333
[5] -8.3333333 -18.0000000 -24.5000000 -32.0000000
[9] -40.5000000 -50.0000000
> x*y-2
[1] 0 2 4 6 8 16 19 22 25 28
> x*(y-2)
[1] 0 0 0 0 0 6 7 8 9 10
```

Functions for numerical Vectors

- **Min & Max**

```
>x<-1:5
> min(x)
[1] 1
> max(x)
[1] 5
> range(x)
[1] 1 5
```

- **Sum & Product**

```
> sum(x)
[1] 15
> prod(x)
[1] 120
```

- **Other examples**

```
> mean(x)
[1] 3
> median(x)
[1] 3
>var(x)
[1] 2.5
> sd(x)
[1] 1.581139

> quantile(x)
  0%  25%  50%  75% 100%
   1    2    3    4    5
```

Functions for numerical Vectors – Quantiles

```
> x<-rnorm(100)
> quantile(x)
        0%        25%        50%        75%       100%
-2.19241520 -0.81158193 -0.09602603  0.43857679  2.16810345
> quantile(x, probs=0.9)
     90%
0.8846058
> quantile(x, probs=c(0.7,0.9))
     70%       90%
0.3872975 0.8846058
> quantile(x, probs=seq(0,1,0.1))
        0%        10%        20%        30%        40%        50%        60%        70%        80%
-2.19241520 -1.36411796 -0.92007561 -0.57215473 -0.37186166 -0.09602603  0.09954098
     0.38729749  0.60784569
       90%       100%
 0.88460576  2.16810345
```

Understanding the Recycling Rule

Vector arithmetic between vectors works predictably when both vectors have the same length. When the vectors have unequal lengths, R invokes a Recycling Rule. In this rule, the value of the expression is a vector with the same length as the longest vector in the expression. Shorter vectors are *recycled* as often as need be until they match the length of the longest vector. R gives a warning only if the length of the longer vector is not multiple of one of the shorter ones.

Example

```
> x <- c(1,2,3,4,5,6) ; y <- c(1,2,3) ; z <- c(1,2,3,4)
> x + y
[1] 2 4 6 5 7 9
> x + z                    # Oops! Length of x not a multiple of 4
[1] 2 4 6 8 6 8
Warning message:
In x + z : longer object length is not a multiple of shorter object
length
```

Selecting Vector Elements

We select elements from a vector using square brackets, e.g. `x[5]`. The *index vector* in `[]` specifying the elements can be any of four distinct types:

1. **A vector of positive integers**, of any length, in any desired order, indicating elements to select. For example,

```
> fib[c(2, 5)]                # Select elements 2 and 5
[1] 1 3
```

2. **A vector of negative integers**, indicating elements to *exclude*. For example,

```
> fib[-c(2, 5)]               # Exclude elements 2 and 5
[1]  0  1  2  5  8 13 21
```

We can't mix positive and negative subscripts; a "0" subscript returns nothing.

3. **A logical vector.** This must be of the same length as the original vector otherwise R will recycle. The elements corresponding to TRUE in the index vector will be selected. Thus,

```
> fib %% 2 == 0
[1]  TRUE FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE
> fib[fib %% 2 == 0]                # Select the even elements
[1] 0 2 8
```

Here's a similar question: "what are the indices of the even elements of fib?"

```
> (1:length(fib))[fib %% 2 == 0]
[1] 1 4 7
```

4. **A vector of character strings.** This option only applies if the original vector has names. The names may be used in the same way as the positive integers in 1.

```
> whitehouse[c("Kennedy", "Clinton")]
Kennedy Clinton
1961    1993
```

Extracting elements in vectors

```
> x<- c(1, 3, 5, 7, 9)
> x
[1] 1 3 5 7 9
> x[2]
[1] 3

> x[2:4]
[1] 3 5 7

> x[c(1,3)]
[1] 1 5
```

```
> x[-c(1,3)]
[1] 3 7 9

> y<-c(F,T,T,F,T)
> x[y]
[1] 3 5 9
```

Missing Values

```
x3<-c(1,2,3,NA,9)
> x3
[1]  1  2  3 NA  9
> is.na(x3)
[1] FALSE FALSE FALSE  TRUE FALSE
```

Missing Values

```
> x3<-c(x3,NA, NA,x3)
> x3
 [1]  1  2  3 NA  9 NA NA  1  2  3 NA  9
> is.na(x3)
 [1] FALSE FALSE FALSE  TRUE FALSE  TRUE  TRUE FALSE
     FALSE FALSE  TRUE FALSE
> x3[is.na(x3)]
[1] NA NA NA NA
> x3[!is.na(x3)]
[1] 1 2 3 9 1 2 3 9
```

Missing Values

```
> which(is.na(x3))
[1]  4  6  7 11
> index<-1:length(x3)
> index
 [1]  1  2  3  4  5  6  7  8  9 10 11 12
> index[is.na(x3)]
[1]  4  6  7 11
> is.na(x3)
 [1] FALSE FALSE FALSE  TRUE FALSE  TRUE  TRUE FALSE
     FALSE FALSE  TRUE FALSE
```

More Vector Input Commands

S has a number of facilities for generating vectors.

(i) The colon (`:`) is useful for generating sequences:

```
> 0:9
[1] 0 1 2 3 4 5 6 7 8 9
> 16:10
[1] 16 15 14 13 12 11 10
```

The colon has highest priority within an expression so, for example, `1:n+1` is the sequence `2,3,4, …,n,n+1`.

Creating sequences of numbers

The colon Operator

In a:b, if (b-a) is not an integer, then R will stop before b

```
> x<-3.3:6.9
> x
[1] 3.3 4.3 5.3 6.3
```

(ii) The **seq** function is a more general facility for generating sequences. It has four main arguments,

```
seq(from = value, to = value, by = value, length.out = value)
```

but it is an error to specify all 4 in any one call. The function uses default values when only 1 or 2 arguments are given :

```
> seq(from = 0, to = 20)
[1]  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
> seq(from = 0, to = 20, by = 4)
[1]  0  4  8 12 16 20
> seq(from = 0, to = 20, length.out = 4)
[1]  0.000000  6.666667 13.333333 20.000000
```

Creating sequences of numbers

The sequence command – seq()

From all the arguments from, to, by, length.out, along.with you can specify up to 3.

if the user specifies 2 then by default R values for the rest of the arguments is one (1).

The seq() last element is always lower or equal to the value of the argument "to".

e.g.

```
> seq(from=1,to=10,by=2)
[1] 1 3 5 7 9
```

(iii) The **rep** function can be used for replicating a structure in various ways. The function has arguments control the number of times, the length of the output and/or the repeating pattern:

```
rep(x = vector, times = vector, each = value, length.out = value)
```

- x: a vector to replicate
- times:
  1. Single value=> how many times to replicate x
  2. Vector of the same length as x: how many times to repeat each element of x
- each: (non-negative integer) each element of x is repeated each times. Default value = 1

```
> rep(1:3, times = 5)
[1] 1 2 3 1 2 3 1 2 3 1 2 3 1 2

> rep(1:3, each=5)
[1] 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3
```

(iv) The **`paste`** function returns a character vector which is the result of pasting corresponding elements of two or more vectors together. The shorter vector is recycled. The argument **`sep="`***`string`***`"`** can be used to change the default spacing (a single space) to *`string`*.

```
> paste ("no", 1:5)
[1] "no 1" "no 2" "no 3" "no 4" "no 5"
> paste("no", 1:5, sep = "")                    # no spacing
[1] "no1" "no2" "no3" "no4" "no5"
```

Sorting numerical Vectors

Sort

```
> x<-c(3,5,2,1,6)
> sort(x)
[1] 1 2 3 5 6
> sort(x,decreasing=T)
[1] 6 5 3 2 1
```

Rank

```
> x<-c(3,5,2,1,6)
> rank(x)
[1] 3 4 2 1 5
```

Order

```
> x<-c(3,5,2,1,6)
> order(x)
[1] 4 3 1 2 5
> x[order(x)]
[1] 1 2 3 5 6
```

the smallest value is on the 4th position, the next one on the 3rd, etc

Handling ties in rank

Average: assigns each tied element the "average" rank

```
> x2<-c(3,5,2,1,6,3)
> rank(x2, ties.method="average")
[1] 3.5 5.0 2.0 1.0 6.0 3.5
```

First: lets the "earlier" entry "win", so the ranks are in numerical order

```
> rank(x2, ties.method="first")
[1] 3 5 2 1 6 4
```

Random: breaks ties randomly

```
> rank(x, ties.method="random")
[1] 4 5 2 1 6 3
> rank(x, ties.method="random")
[1] 3 5 2 1 6 4
```

Min/Max: assigns every tied element to the lowest/highest rank

```
> rank(x, ties.method="min")
[1] 3 5 2 1 6 3
> rank(x, ties.method="max")
[1] 4 5 2 1 6 4
```

Character Vectors

Every vector with character elements is included in quotes ("*")

```
> x<-c("Statistics", "Mathematics")
> x
[1] "Statistics" "Mathematics"
> x<-c( 'Yiannis', 'Yiorgos', 'Grigoris' )
> x
 [1] "Yiannis"  "Yiorgos"  "Grigoris"
> x[2]
[1] "Yiorgos"
```

Arithmetic operations and characters

Generally, these do not make sense:

```
> x1 <- 1
> x2 <- "1"
> # adding a character and a numeric vector
> x1+x2
Error in x1 + x2 : non-numeric argument to binary operator
```

Arithmetic operators and characters

```
> # using as.numeric() in character vectors
> # converting a character to a number and adding it to another
    number
> as.numeric('1')+5
[1] 6
> as.numeric('16g')+5
[1] NA
Warning message:
NAs introduced by coercion
> x1<- "5"
> x1^2
Error in x1^2 : non-numeric argument to binary operator
> as.numeric(x1)^2
[1] 25
```

Logical operations and characters

Generally, these also do not make much sense, other than equalities perhaps:

```
> x1 <- "5"
> x1 == "5"
[1] TRUE
> x1 == "5 "    #  spaces are important
[1] FALSE

> x2 <- "g"
> x2 == "G"    # R is case sensitive
[1] FALSE
> x2 == "g"
[1] TRUE
```

Equalities with characters

```
> letters
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s" "t" "u"
     "v" "w" "x" "y" "z"
> # checking if each element of character vector 'letters'  is equal to 'b'
>  letters=='b'
 [1] FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
     FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[19] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
> x<-letters=='b'
> index<-1:length(letters)
> index
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
> index[x]
[1] 2
> which(x)
[1] 2
```

Equalities with characters

```
> letters==c('b','c')
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
     FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[19] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
> letters==c('g','b','c')
 [1] FALSE  TRUE  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
     FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[19] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
Warning message:
In letters == c("g", "b", "c") :
  longer object length is not a multiple of shorter object length
> as.numeric(letters==c('g', 'b','c'))
 [1] 0 1 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Warning message:
In letters == c("g", "b", "c") :
  longer object length is not a multiple of shorter object length
```

Functions for character vectors

- **character(length = 0)** : creates a character vector with length 0
- **as.character(x, ...)** : turns a vector to a character vector
- **is.character(x)** : checks if a vector is a character vector
- **print**: prints all elements with quotes
- **noquote** : prints all elements without quotes
- **nchar** : number of element's characters

Functions for character vectors

- **character(length = 10)**: creates a character vector with length 10

```
> character(10)
 [1] "" "" "" "" "" "" "" "" "" ""
```

- **as.character(x, ...)** : turns a vector to a character vector

```
> as.character(1:10)
 [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"
> as.character(c(T,F))
[1] "TRUE"  "FALSE"
```

- **is.character(x)** : checks if a vector is a character vector

```
> x<-as.character(c(T,F))
> is.character(x)
[1] TRUE
> is.character(1:10)
[1] FALSE
```

Functions for character vectors

- **print**: prints all elements with quotes
- **noquote** : prints all elements without quotes

```
> x<-c('Statistics', 'Mathematics')
> x
[1] "Statistics"  "Mathematics"
> print(x)
[1] "Statistics"  "Mathematics"
> noquote(x)
[1] Statistics  Mathematics
> z<-noquote(x)
> z[1]
[1] Statistics
> z[2]
[1] Mathematics
```

Functions for character vectors

- **nchar** : number of element's characters

```
> xsurnames<-c('Papadopoulos','Kitsos','Ioannou', 'Tsapara', 'Grigoriadou')
> nchar(xsurnames)
 [1] 12  6  7  7 11
```

More examples

is.character()

```
> y<-as.character(x)
 [1] "1"  "2"  "3"  "4"  "5"  "6"
     "7"  "8"  "9"  "10"
> is.character(y)
[1] TRUE
> x<-c("-0.1","2.7","B")
> x
[1] "-0.1" "2.7"  "B"
> is.character(x)
[1] TRUE
> x<-as.numeric(x)
Warning message:
NAs introduced by coercion
> x
[1] -0.1  2.7   NA
> is.character(x)
[1] FALSE
```

noquote()

```
> y<-as.character(x)
 [1] "1"  "2"  "3"  "4"  "5"
     "6"  "7"  "8"  "9"  "10"
> noquote(y)
[1] 1  2  3  4  5  6  7  8  9
    10
```

nchar()

```
> y<-as.character(x)
 [1] "1"  "2"  "3"  "4"  "5"
     "6"  "7"  "8"  "9"  "10"
> nchar(y)
 [1] 1 1 1 1 1 1 1 1 1 2
```

Pasting character vectors

paste(vector1, ..., vectorN, sep = " ", collapse = NULL)

Combines different character vectors

sep is the separator between the vectors to be combined

collapse checks whether the new vector is going to be a sequence or not

Pasting character vectors Examples

```
> paste('Yiannis','Nicolaou')
[1] "Yiannis Nicolaou"

> x1<-'Yiannis'
> x2<-'Nicolaou'
> paste(x1,x2)
[1] "Yiannis Nicolaou"

> x1<-5
> x2<-13
> paste(x1,x2, x1+x2, sep=' + ')
[1] "5 + 13 + 18"
> paste(paste(x1,x2, sep=' + '), x1+x2, sep=' = ')
[1] "5 + 13 = 18"
```

Pasting character vectors Examples

```
> xnames<-c('Yiannis','Yiorgos','Barbara','Aleka', 'Eugenia')
> xsurnames<-c('Papadopoulos','Kitsos','Ioannou', 'Tsapara',‘
  Grigoriadou')
> paste(xnames, xsurnames)
[1] "Yiannis Papadopoulos" "Yiorgos Kitsos "  "Barbara Ioannou"
  "Aleka Tsapara" "Eugenia Grigoriadou"

> paste(xsurnames,xnames, sep=', ')
[1] "Papadopoulos, Yiannis" "Kitsos, Yiorgos" "Ioannou, Barbara"
  "Tsapara, Aleka"  "Grigoriadou, Eugenia"
```

Pasting character vectors Examples

```
> paste(1:10, collapse=' ')
[1] "1 2 3 4 5 6 7 8 9 10"
> paste(1:10, collapse='')
[1] "12345678910"
> paste(1:10, collapse='+')
[1] "1+2+3+4+5+6+7+8+9+10 "
> paste(xnames, collapse=', ')
[1] "Yiannis, Yiorgos, Barbara, Aleka, Eugenia "
> paste(xnames, xsurnames, collapse=',')
[1]  "Yiannis Papadopoulos,Yiorgos Kitsos,Barbara Ioannou,Aleka
     Tsapara,Eugenia Grigoriadou"
```

Pasting vectors - summary

paste()

```
> paste('Mathematical','Statistics')
[1] "Mathematical Statistics"
```

Argument "sep"

```
> paste('3','5','8', sep="+")
[1] "3+5+8"
> paste('Chapter',2, sep=" ")
[1] "Chapter 2"
> paste("Today is", date())
[1] "Today is Mon Jun 03 20:42:41 2013"
```

Nested pasted

```
> paste(paste(3,5, sep=' + '), 8, sep=' = ')
[1] "3 + 5 = 8"
```

Pasting elements of two vectors

```
> a<-c('Kwstas', 'Maria')
> b<-c('Papadopoulos', 'Kyriakou')
> paste(a,b)
[1] "Kwstas Papadopoulos" "Maria Kyriakou"
```

Pasting vectors - summary

Pasting a character string with a vector

```
> paste("Chapter", 1:2, sep=" ")
[1] "Chapter 1" "Chapter 2"
```

Pasting two vectors of different length

```
> a<-c('Kwstas', 'Maria')
> b<-c('Papadopoulos', 'Kyriakou', 'Anagnostou')
> paste(a,b)
[1] "Kwstas Papadopoulos" "Maria Kyriakou"  "Kwstas Anagnostou"
```

The collapse argument

```
> a<-c('Kwstas', 'Maria')
> paste(a, collapse=",")
[1] "Kwstas,Maria"
> paste(1:10, collapse='+')
[1] "1+2+3+4+5+6+7+8+9+10"
> b<-c('Papadopoulos', 'Kyriakou', 'Anagnostou')
> paste(a, b, collapse=", ")
[1] "Kwstas Papadopoulos, Maria Kyriakou, Kwstas
Anagnostou"
```

Changing to capital or lower case letters

Changing to upper and lower case letters using the functions: toupper ( ) & tolower( )

```
> x
[1] "Statistics"  "Mathematics"
> tolower(x)
[1] "statistics"  "mathematics"
> toupper(x)
[1] "STATISTICS"  "MATHEMATICS"
```

Splitting characters

strsplit(x, split, extended = TRUE, fixed = FALSE, perl = FALSE)

splits characters of a vector

x the vector to be split

split: a vector with the characters to which the separation will take place

```
> strsplit('Yiannis','i')
[[1]]
[1] "Y"   "ann" "s "
>  strsplit('Yiannis',NULL)
 [[1]]
 [1] "Y" "i" "a" "n" "n" "i" "s "
```

Splitting characters

```
x<-paste( xnames, xsurnames, sep=', ')
> x
[1] "Yiannis, Papadopoulos" "Yiorgos, Kitsos " " Barbara, Ioannou"
[4] "Aleka, Tsapara“  "Eugenia, Grigoriadou"
> strsplit(x,split=',')
[[1]]
[1] "Yiannis"       " Papadopoulos“

[[2]]
[1] "Yiorgos" " Kitsos“

[[3]]
[1] "Barbara"  " Ioannou“

[[4]]
[1] "Aleka"    " Tsapara“

[[5]]
[1] "Eugenia"      " Grigoriadou"
```

Splitting characters

Splitting characters function strsplit()

```
> x<-c("Statistics", "Mathematics")
> strsplit(x,split="a")
[[1]]
[1] "St"      "tistics"
[[2]]
[1] "M"    "them" "tics"
> strsplit(x, split=")
[[1]]
 [1] "S" "t" "a" "t" "i" "s" "t" "i" "c" "s"
[[2]]
 [1] "M" "a" "t" "h" "e" "m" "a" "t" "i" "c" "s"
> strsplit(x, split="th")
[[1]]
[1] "Statistics"
[[2]]
[1] "Ma"      "ematics"
```

Substrings

Extract substrings from a character using the function substr()

```
> substr("abcdef",2,4)
[1] "bcd"
> x<-c("Statistics", "Mathematics")
> substr(x,2,4)
[1] "tat" "ath"
```

Identifying substrings in characters using grep()

```
> countries<-c("Greece", "United States", "United Kingdom", "Italy",
    "France", "United Arab Emirates")
> grep("United", countries)
[1] 2 3 6
> grep("United", countries, value=TRUE)
[1] "United States"      "United Kingdom"      "United Arab Emirates"
```

Substitutions

Character substitution using sub() [only the first time]

```
> values<-c("1,700", "2,300")
> as.numeric(values)
[1] NA NA
Warning message:
NAs introduced by coercion
> as.numeric(gsub(",","",values))
sub("," , "*", "1,000,000")
[1] "1*000,000"
> gsub("," , "*", "1,000,000")
[1] "1*000*000"
```

Character substitution using gsub() [all times]

```
> values<-c("1,000,000", "2,000,000")
> sub(",","",values)
[1] "1000,000" "2000,000"
> gsub(",","",values)
[1] "1000000" "2000000"
```

Logical Vectors

```
> logical(3)
[1] FALSE FALSE FALSE

> as.logical(c(0:10))
[1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
    TRUE  TRUE  TRUE  TRUE
```

Logical Vectors

```
> x<-logical(10)
> x
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
     FALSE
> as.logical( 0:4 )
[1] FALSE  TRUE  TRUE  TRUE  TRUE
> as.logical( 0.000001:4 )
[1] TRUE TRUE TRUE TRUE
> as.logical( c('TRUE', 'FALSE') )
[1]  TRUE FALSE
> as.logical( c('TRUE', 'FALS') )
[1] TRUE   NA
> as.logical( c('TRUE', 'F') )
[1]  TRUE FALSE
> as.logical( c('TRUE', '0') )
[1] TRUE   NA
```

Logical Vectors

```
> 1:6<= 20
[1] TRUE TRUE TRUE TRUE TRUE TRUE
> 1:6 <= 3
[1]  TRUE  TRUE  TRUE FALSE FALSE FALSE
> 1:6 <= 1:6
[1] TRUE TRUE TRUE TRUE TRUE TRUE
> 1:6 <= c(4,2,1,6,3,2)
[1]  TRUE  TRUE FALSE  TRUE FALSE FALSE
> 1:6 <= c(4,2,1,6,3)
[1]  TRUE  TRUE FALSE  TRUE FALSE FALSE
Warning message:
In 1:6 <= c(4, 2, 1, 6, 3) :
  longer object length is not a multiple of shorter object length
> 3*(1:6 <= c(4,2,1,6,3,2))
[1] 3 3 0 3 0 0
> sum(19:25)<= 20
[1] FALSE
```

Logical Vectors

```
> x<-  c(3, 9, 7, 1, 1, 5, 7, 2, 9, 1)
>y<-c(7, 9, 4, 1, 10, 4, 8, 10, 4, 8)
> x==y
 [1] FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
> x!=y
 [1]  TRUE FALSE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
> !(x=y)
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
     FALSE
> !(x==y)
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
     FALSE
> (x==y)
 [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
> !(x==y)
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
     FALSE
```

Logical Vectors

```
> as.numeric(x ==1)
 [1] 0 0 0 1 1 0 0 0 0 1
> sum(as.numeric(x ==1))
[1] 3
> sum(as.numeric(x>3))
[1] 5
> sum(x>3)
[1] 5
> sum(!(x>3))
[1] 5
```

Logical Vectors

```
> (x<4)*2
 [1] 2 0 0 2 2 0 0 2 0 2
> x
 [1] 3 9 7 1 1 5 7 2 9 1
> x<4*2
 [1]  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE
> (x<4)*2
 [1] 2 0 0 2 2 0 0 2 0 2
> x<(4*2)
 [1]  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE
>
```

Logical Vectors

```
> y<- 10*(x>=5) + (-10)*(x<5)
> y
 [1] -10  10  10 -10 -10  10  10 -10  10 -10
> y<- 10*(x<3) + (-10)*(x>7) + 25*(x>=3)*(x<=7)
> y
 [1]  25 -10  25  10  10  25  25  10 -10  10
> (x>=3)*(x<=7)
 [1] 1 0 1 0 0 1 1 0 0 0
> (x>=3)&(x<=7)
 [1]  TRUE FALSE  TRUE FALSE FALSE  TRUE  TRUE FALSE
    FALSE FALSE
> (x>=3)&(x<=7) ==  (x>=3)&(x<=7)
 [1]  TRUE FALSE  TRUE FALSE FALSE  TRUE  TRUE FALSE
    FALSE FALSE
```

Logical Vectors

```
> (x>=3)*(x<=7)
 [1] 1 0 1 0 0 1 1 0 0 0
> (x>=3)&(x<=7)
 [1]  TRUE FALSE  TRUE FALSE FALSE  TRUE  TRUE FALSE FALSE
    FALSE
# Be careful
> (x>=3)&(x<=7) ==  (x>=3)&(x<=7)
 [1]  TRUE FALSE  TRUE FALSE FALSE  TRUE  TRUE FALSE FALSE
    FALSE
> (x>=3)&((x<=7) ==  (x>=3))&(x<=7)
 [1] FALSE FALSE  TRUE  TRUE  TRUE
 [6]  TRUE  TRUE FALSE FALSE FALSE
> ((x>=3)&(x<=7)) ==  ((x>=3)&(x<=7))
 [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE
 [8] TRUE TRUE TRUE
```

Logical Vectors

```
> x1<-(x>=3)*(x<=7)
> x2<-(x>=3)&(x<=7)
> x1 == as.numeric(x2)
 [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
> sum(x1 == as.numeric(x2))
[1] 10
> sum(x1 == as.numeric(x2))==length(x1)
[1] TRUE
> all(x1 == as.numeric(x2))
[1] TRUE
> any(x1 == as.numeric(x2))
[1] TRUE
> sum(x1 == as.numeric(x2))>=1
[1] TRUE
```

Functions all & any

- all(x) => TRUE if all the elements of the vector x are TRUE
  Equivalent to sum(x)==length(x)

- any(x) => TRUE if at least one element of the vector x are TRUE
  Equivalent to sum(x)>0

```
> all( c(T,T,T,T,T) )
[1] TRUE
> all( c(T,T,T,T,F) )
[1] FALSE
> any( c(T,T,T,T,F) )
[1] TRUE
```

```
> any( c(T,F,F,F,F) )
[1] TRUE
> any( c(F,F,F,F,F) )
[1] FALSE
> all( c(F,F,F,F,F) )
[1] FALSE
```

Functions all & any

```
> test<-c(T,F,T,T,F,T,F)
> test
[1]  TRUE FALSE  TRUE  TRUE FALSE  TRUE FALSE
> any(test)      # = TRUE - έστω και ένα να είναι TRUE τότε το
    αποτέλεσμα TRUE
[1] TRUE
> any( c(F,F,F))  # = FALSE- αφού όλα είναι false
[1] FALSE
> any( c(F,F,T))  # = TRUE - αφού ένα είναι T
[1] TRUE
> all( c(F,F,F))  # = FALSE - για να είναι T πρέπει όλα να είναι T
[1] FALSE
> all( c(F,F,T))  # = FALSE
[1] FALSE
```

Functions all & any

```
> all( c(F,T,T))  # = FALSE
[1] FALSE
> all( c(T,T,T))  # = TRUE εφόσον όλα είναι true
[1] TRUE
> sum( c(T,T,T))  # αθροισμα από true
[1] 3
> sum( c(T,T,T))==length(c(T,T,T))
# είναι το άθροισμα των true ίσο με το μήκος του διανύσματος δλδ είναι
    όλα true;
[1] TRUE
```

Functions all & any

```
> #
> # γενικότερη προσέγγιση μέσω διανύσματος
> #
> w2<-c(T,T,T)
> sum( w2)==length(w2)# ισοδύναμό του all
[1] TRUE
> all(w2)
[1] TRUE
> any(w2)
[1] TRUE
> sum( w2)>0 # ισοδύναμό του any
[1] TRUE
> !(sum( w2)==0) # ισοδύναμό του any
[1] TRUE
> !any(w2) # ολά false
[1] FALSE
```

Functions all & any

```
> math <- c(6, 8, 7, 6, 9, 3, 3, 6, 7, 4, 1, 2, 9, 9, 10, 6, 6, 4, 1, 7, 5, 7, 10, 4, 1, 5,
    10, 2, 1, 6, 9, 9, 10, 8, 10, 9, 8, 7, 6, 6, 8, 6, 9, 1, 4, 8, 3, 1, 5, 8, 10, 1, 4, 3, 4,
    6, 5, 1, 1, 2, 1, 9, 2, 5, 1, 1, 1, 5, 3, 7, 6, 10, 9, 2, 2, 10, 8, 5, 9, 9, 10, 4, 2, 10,
    5, 6, 3, 6, 8, 4, 1, 1, 6, 5, 8, 5, 9, 8, 5, 4)
> any(math<5) # υπάρχει έστω και ένας φοιτητής κάτω από τη βάση;
[1] TRUE
> any(math>8) # υπάρχει έστω και ένας φοιτητής πάνω από 8;
[1] TRUE
> all(math>=5) # περάσαν όλοι οι φοιτητές το μάθημα;
[1] FALSE
> table(math)  # δίνει τον πίνακα συχνοτήτων με τις βαθμολογίες
1  2  3  4  5  6  7  8  9 10
15  7  6  9 11 14  6 10 12 10
```

Functions all & any

```
> yiannis <- TRUE
> #
> # πράξεις με all και λογικά διανύσματα
> all(math>=4) & yiannis
[1] FALSE
> all(math>=5) & yiannis
[1] FALSE
> all(math>=5) * yiannis
[1] 0
> all(math>=4) * yiannis
[1] 0
> all(math>=4) & !yiannis
[1] FALSE
>
```

Logical operators: & and |

```
> x <- round( rnorm(10, 70,10) ); x
 [1] 66 87 65 60 87 67 62 52 67 79
> t1<- (x<70);  t1
 [1]  TRUE FALSE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE
    FALSE
> !t1
 [1] FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
    TRUE
> t2<- (x>70);  t2
 [1] FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
    TRUE
> t1&t2
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
    FALSE
> t1|t2
 [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

Combine logicals with other vectors

```
> c((x>=30),(x<=70))
 [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
    TRUE FALSE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE
[19]  TRUE FALSE
> c((x>=30),(x<=70), '1')
 [1] "TRUE"  "TRUE"  "TRUE"  "TRUE"  "TRUE"  "TRUE"  "TRUE"  "TRUE"
    "TRUE"  "TRUE"  "TRUE"  "FALSE" "TRUE"
[14] "TRUE"  "FALSE" "TRUE"  "TRUE"  "TRUE"  "TRUE"  "FALSE" "1"
> c((x>=30),(x<=70), 1)
 [1] 1 1 1 1 1 1 1 1 1 1 1 0 1 1 0 1 1 1 1 0 1
> c((x>=30),(x<=70), 10)
 [1]  1  1  1  1  1  1  1  1  1  1  1  0  1  1  0  1  1  1  1  0 10
> c((x>=30),(x<=70), 10, '5fsd')
 [1] "TRUE"  "TRUE"  "TRUE"  "TRUE"  "TRUE"  "TRUE"  "TRUE"  "TRUE"
    "TRUE"  "TRUE"  "TRUE"  "FALSE" "TRUE"
[14] "TRUE"  "FALSE" "TRUE"  "TRUE"  "TRUE"  "TRUE"  "FALSE" "10"
    "5fsd"
```

The logical operator %in%

The logical operator %in% checks if an element is also an element of another object (e.g. vector)

```
> 3 %in% 1:10
[1] TRUE
> 15 %in% 1:10
[1] FALSE
> c(3,15)%in% 1:10
[1]  TRUE FALSE
> # is 'a' an ellement of vector 'letters'?
> 'a' %in% letters
[1] TRUE
>
```

The logical operator %in%

```
> z<-c("Grigoris", "Antonis", "Yiannis")
> z
[1] "Grigoris" "Antonis"  "Yiannis"
> NA %in% z
[1] FALSE
> z<-c(z,NA)
> NA %in% z
[1] TRUE
>
```

The logical operator %in%

```
> z<-c(10, 5, 9, 11, 11, 16, 7)
> z[3:7] %in% z
[1] TRUE TRUE TRUE TRUE TRUE
> all(z[3:7] %in% z) # checks if all of z[3], …, z[7] belong to z
[1] TRUE
>
> 1:4 %in% z
[1] FALSE FALSE FALSE FALSE
> all(1:4 %in% z) # checks if all of 1, 2, 3, 4 belong to z
[1] FALSE
> any(1:4 %in% z) # checks if any of 1, 2, 3, 4 belong to z
[1] FALSE
>
```

Factors

Categorical Variables:

```
> gender<-c('Male', 'Female', 'Male', 'Male', 'Female')
> gender
[1] "Male"   "Female" "Male"   "Male"   "Female"
> factor(gender)
[1] Male   Female Male   Male   Female
Levels: Female Male
> levels(factor(gender))
[1] "Female" "Male"
```

Ordinal Variables:

```
> opinion<-c('Low', 'Low', 'High', 'High', 'High',
'Medium')
> ordered(opinion, levels=c('Low', 'Medium', 'High'))
[1] Low    Low    High   High   High   Medium
Levels: Low < Medium < High
```

Factors

```
> gender<-c('Male', 'Female', 'Male', 'Male', 'Female'); gender
[1] "Male"   "Female" "Male"   "Male"   "Female"
> is.character(gender)
[1] TRUE
> x<-factor(gender); x
[1] Male   Female Male   Male   Female
Levels: Female Male
> is.character(x)
[1] FALSE
> is.numeric(x)
[1] FALSE
> is.factor(x)
[1] TRUE
```

```
> mode(x)
[1] "numeric"
> class(x)
[1] "factor"
```

Factors

```
> levels(x) <- c('Γυναίκα', 'Ανδρας'); x
[1] Ανδρας  Γυναίκα Ανδρας  Ανδρας  Γυναίκα
Levels: Γυναίκα Ανδρας
> levels(x) <- NULL
Error in 'levels<-.factor'('*tmp*', value = NULL) :
  number of levels differs
> levels(x) <- c(1,2)
> x
[1] 2 1 2 2 1
Levels: 1 2
```

Factors

```
> z<-c(1,2,3,4,1,1,1,2,3,4,1); factor(z)
 [1] 1 2 3 4 1 1 1 2 3 4 1
Levels: 1 2 3 4
> z0<-factor(z)
> levels(z0)
[1] "1" "2" "3" "4"
> levels(z0)<-paste('level',1:4,sep='')
> levels(z0)
[1] "level1" "level2" "level3" "level4“
> z0
 [1] level1 level2 level3 level4 level1 level1 level1 level2 level3 level4 level1
Levels: level1 level2 level3 level4
> factor(z, labels=c('l1', 'L2', 'l3', 'L4'))
 [1] l1 L2 l3 L4 l1 l1 l1 L2 l3 L4 l1
Levels: l1 L2 l3 L4
```

Factors

```
> opinion<-c('Low', 'Low', 'High', 'High', 'High', 'Medium')
> ordered(opinion, levels=c('Low', 'Medium', 'High'))
[1] Low    Low    High   High   High   Medium
Levels: Low < Medium < High
> ordered(rep(1:5,2), levels=c('Low', 'Medium', 'High'))
 [1] <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA>
Levels: Low < Medium < High
> rep(1:5,2)
 [1] 1 2 3 4 5 1 2 3 4 5
> ordered(rep(1:5,2), levels=c('VL','L','M','H', 'VH'))
 [1] <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA>
Levels: VL < L < M < H < VH
> ordered(rep(1:5,2), levels=1:5, labels=c('VL','L','M','H', 'VH'))
 [1] VL L  M  H  VH VL L  M  H  VH
Levels: VL < L < M < H < VH
```

Factors

```
> ordered(rep(1:5,2), levels=1:5, labels=c('L','VL','M','H', 'VH'))
 [1] L  VL M  H  VH L  VL M  H  VH
Levels: L < VL < M < H < VH
> ordered(rep(1:5,2), levels=c(2,1,3:5), labels=c('VL','L','M','H', 'VH'))
 [1] L  VL M  H  VH L  VL M  H  VH
Levels: VL < L < M < H < VH
> ordered(rep(1:5,2), levels=c('VL','L','M','H', 'VH'))
 [1] <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA> <NA>
Levels: VL < L < M < H < VH
> ordered(rep(1:5,2), levels=c(1:5), labels=c('VL','L','M','H', 'VH'))
 [1] VL L  M  H  VH VL L  M  H  VH
Levels: VL < L < M < H < VH
```

Matrices

In R, a matrix is just a vector that has dimensions. Thus, all the elements of a matrix must be of the same mode (e.g. all numeric, logical or character).

The main function for creating a matrix is the **matrix** command:

```
matrix(data = vector, nrow = value, ncol = value, byrow = FALSE)
```

where data is a vector containing the elements, nrow is the number of rows, and ncol is the number of columns.

```
> matrix(c(1.1,1.2,2.1,2.2,3.1,3.2), nrow = 2, ncol = 3)
     [,1] [,2] [,3]
[1,]  1.1  2.1  3.1
[2,]  1.2  2.2  3.2
```

Enries go down columns. If the data do not fill the matrix, R will apply the Recycling Rule.

```
> matrix(0, 2, 3)                    # Create an all-zero matrix
     [,1] [,2] [,3]
[1,]    0    0    0
[2,]    0    0    0
```

Let's save this as our matrix to work with.

```
> ( m <- matrix (c(1.1, 1.2, 2.1, 2.2, 3.1, 3.2), 2, 3) )
     [,1] [,2] [,3]
[1,]  1.1  2.1  3.1
[2,]  1.2  2.2  3.2

> dim(m)                             # What are the dimensions of m?
[1] 2 3                              # Notice this is a vector of length 2.
```

Examples on matrices

```
> matrix0 <- matrix(nrow=3, ncol=4)
> matrix0
     [,1] [,2] [,3] [,4]
[1,]   NA   NA   NA   NA
[2,]   NA   NA   NA   NA
[3,]   NA   NA   NA   NA
> x<-c(1:3,11:13,21:23,31:33)
> x
[1] 1 2 3 11 12 13 21 22 23 31 32 33
> y<-matrix(x,ncol=3)
> y
     [,1] [,2] [,3]
[1,]    1   12   23
[2,]    2   13   31
[3,]    3   21   32
[4,]   11   22   33
```

```
> y<-matrix(x,nrow=4)
> y
     [,1] [,2] [,3]
[1,]    1   12   23
[2,]    2   13   31
[3,]    3   21   32
[4,]   11   22   33
```

Entering the elements of a Matrix by rows

- In matrices, elements are introduced by columns i.e. first the first column, then the second etc.
- The **byrow** argument specifies how the matrix is to be filled.
- The default value for **byrow** is **FALSE** which means that by default the matrix will be filled column by column.
- If we want to fill the matrix row by row, we have to write in the function matrix() the argument byrow=T.

Entering the elements of a matrix by row

Entries go down columns unless we specify `byrow=TRUE`:

```
> matrix(c(1.1,1.2,2.1,2.2,3.1,3.2), nrow = 2, ncol = 3)
     [,1] [,2] [,3]
[1,]  1.1  2.1  3.1
[2,]  1.2  2.2  3.2

> matrix(c(1.1,1.2,2.1,2.2,3.1,3.2), 2, 3, byrow = TRUE)
     [,1] [,2] [,3]
[1,]  1.1  1.2  2.1
[2,]  2.2  3.1  3.2
```

Returning back to vectors

```
> x<-matrix(1:12, 3)
> x
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> as.vector(x)
 [1]  1  2  3  4  5  6  7  8  9 10 11 12
```

```
> x<-matrix(1:12, 3, byrow=TRUE)
> x
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
> as.vector(x)
 [1]  1  5  9  2  6 10  3  7 11  4  8 12
```

Single column or row matrices

- To enter a $p \times 1$ column vector, simply enter a $p \times 1$ matrix.

```
> y<- matrix(c (1 ,2 ,3 ,4) ,4 ,1)
> y
     [,1]
[1,]    1
[2,]    2
[3,]    3
[4,]    4
```

- Row vectors are, likewise, entered as $1 \times q$ matrices.

```
> x<- matrix(c (1 ,2 ,3 ,4) ,1 ,4)
> x
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
```

Main characteristics of matrices

```
> x <- c(1:3,11:13,21:23,31:33)
> y <- matrix(x, 3, 4)

     [,1] [,2] [,3] [,4]
[1,]    1   11   21   31
[2,]    2   12   22   32
[3,]    3   13   23   33
```

Length

```
> length(y)
[1] 12
```

Dimensions

```
dim(y)
nrow(y)
ncol(y)
```

Names

```
names(y)
dimnames(y)
colnames(y)
rownames(y)
```

The dimension of a matrix

- The dim() function provides the dimensions of a matrix. The outcome is a vector with two elements.

- We can also use dim() to create matrices from vectors. So, if we give dimensions to a vector, then the elements of the vector will create a matrix with those dimensions. The matrix is filled by column only.

The dimension of a matrix

```
> x<-1:6
> dim(x)<-c(3,2);  x
     [,1]  [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> dim(x)<-c(2,3); x
    [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> dim(x)<-c(1,6);  x
    [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    2    3    4    5    6
```

The dimension of a matrix

```
> matrix0 <- matrix(1:20, 4, 5)
> dim(matrix0)     # vector with two elements - number of rows and
                   # number of columns
[1] 4 5
> dim(matrix0)[1]  # number of rows
[1] 4
> dim(matrix0)[2]  # number of columns
[1] 5
> nrow(matrix0)    # alternative command: number of rows
[1] 4
> ncol(matrix0)    # alternative command: number of columns
[1] 5
```

Dimension names of a matrix

- We can assign names to the rows and columns of the matrix with the utilization of dimnames() command.
- By this way we can access the elements by names.
- It is needed to specify the names for the rows and columns in that order in a list.
- The dimnames() of a matrix can be a list of the same length as the dimension of the matrix or NULL (no name). Its components of the list are either a character vector with positive length of the appropriate dimension of the matrix or NULL. The list can have names.

Dimension names of a matrix

- If we want to delete a name from a specific row or column, we set it as NULL.
- The command dimnames can be included as a parameter in the matrix() command,

  matrix(..., dimnames = list(…), ...).

- If we want to name only the rows or the columns of a matrix we can use the rownames() or colnames() commands, respectively. The dimnames() command can operate on both rows and columns at the same time.

Dimension names of a matrix

```
> x<-matrix(c(2.3,1.4,0,-12,5.27,6), nrow=2, ncol=3)
> dimnames(x) <- list( c("Row 1","Row 2"),c("Col 1", "Col 2","Col 3")); x
        Col 1  Col 2  Col 3
Row 1   2.3     0      5.27
Row 2   1.4   -12     6.00
> dimnames(x)<-list(NULL,c("Col 1","Col 2","Col 3")); x
   Col 1  Col 2 Col 3
[1,]   2.3    0    5.27
[2,]   1.4  -12   6.00
> dimnames(x)<-list( c("Row 1","Row 2"), NULL) ; x
     [,1] [,2] [,3]
Row 1  2.3    0 5.27
Row 2  1.4  -12 6.00
> dimnames(x)<-NULL; x
   [,1] [,2] [,3]
[1,]  2.3    0 5.27
[2,]  1.4  -12 6.00
```

Dimension names of a matrix

```
> colnames(x) <- paste( 'Col', 1:3, sep='' )
> dimnames(x)
[[1]]
NULL

[[2]]
[1] "Col1" "Col2" "Col3“

> x
   Col1 Col2 Col3
[1,]  2.3    0 5.27
[2,]  1.4  -12 6.00
```

```
> rownames(x) <- paste( 'R', 1:2, sep='' )
> dimnames(x)
[[1]]
[1] "R1" "R2“

[[2]]
[1] "Col1" "Col2" "Col3“

> x
   Col1 Col2 Col3
R1  2.3    0 5.27
R2  1.4  -12 6.00
```

Dimension names of a matrix

```
> names(dimnames(x))<-c('X', 'Y')
> x
   Y
X    Col1 Col2 Col3
  R1  2.3    0 5.27
  R2  1.4  -12 6.00
> dimnames(x)
$X
[1] "R1" "R2"

$Y
[1] "Col1" "Col2" "Col3"
```

Selecting Matrix Elements

With a vector, we subscript with a single index. It seems natural that we should use two subscripts for a matrix, separated with a comma. If we omit a subscript, we get the whole row or column. For example:

```
> ( m <- matrix (c(1.1, 1.2, 2.1, 2.2, 3.1, 3.2), 2, 3) )
     [,1] [,2] [,3]
[1,]  1.1  2.1  3.1
[2,]  1.2  2.2  3.2
> m[2, 3]                 # Just like algebra!
[1] 3.2
> row1 <- m[1,]           # First row
[1] 1.1 2.1 3.1
> col3 <- m[,3]           # Third column
[1] 3.1 3.2
```

If the result has length 1 in any dimension, this is *dropped* unless we use the argument `drop=FALSE` :

```
> row1 <- m[1,,drop = FALSE]       # row1 is now a 1x3 matrix
     [,1] [,2] [,3]
[1,]  1.1  2.1  3.1
```

As with vectors, we can use negative integers, logical vectors or character strings to select certain rows or columns. For example

```
> m[-2, -1]                    # Exclude 2nd row and 1st column
[1] 2.1 3.1

> m[, m[1,]>2]                 # Select the columns for which 1st row is > 2
     [,1] [,2]
[1,]  2.1  3.1
[2,]  2.2  3.2
```

Extracting a row of a matrix

- To get an entire row of a matrix, you name the row and leave out the column.
- For example, in the matrix A below, to get the first row, just enter A[1,].

```
> A
     [,1] [,2] [,3] [,4]
[1,]   -5   -2    1    4
[2,]   -4   -1    2    5
[3,]   -3    0    3    6
> A[1,]
[1] -5 -2  1  4
```

```
> A[2,]
[1] -4 -1  2  5
> is.vector(A[2,])
[1] TRUE
> is.matrix(A[2,])
[1] FALSE
```

Changing the values of a row of a matrix

```
> A[2,]
[1] -4 -1  2  5
> A[2,]<-0
> A
     [,1] [,2] [,3] [,4]
[1,]   -5   -2    1    4
[2,]    0    0    0    0
[3,]   -3    0    3    6
> A[2,] <- 1:4
> A
     [,1] [,2] [,3] [,4]
[1,]   -5   -2    1    4
[2,]    1    2    3    4
[3,]   -3    0    3    6
```

Extracting a Column of a Matrix

- To get an entire column of a matrix, you name the column and leave out the row.
- For example, in the matrix A below, to get the first column, just enter A[,1].

```
> A
     [,1] [,2] [,3] [,4]
[1,]   -5   -2    1    4
[2,]   -4   -1    2    5
[3,]   -3    0    3    6
> A[,1]
[1] -5 -4 -3
```

```
> A[,4]
[1] 4 5 6
> is.vector(A[,4])
[1] TRUE
> is.matrix(A[,4])
[1] FALSE
```

Changing the values of a Column of a Matrix

```
> A[,4]
[1] 4 4 6
> A[,4]<-c(0,10,100)
> A
     [,1] [,2] [,3] [,4]
[1,]   -5   -2    1    0
[2,]    1    2    3   10
[3,]   -3    0    3  100
```

Extracting sub-matrices using numeric vectors

```
> x<-matrix(1:30, ncol=6)
> x
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    6   11   16   21   26
[2,]    2    7   12   17   22   27
[3,]    3    8   13   18   23   28
[4,]    4    9   14   19   24   29
[5,]    5   10   15   20   25   30
> index1<-c(2,5,4)
> index2<-c(1:2,6,2)
> x[index1,index2]
     [,1] [,2] [,3] [,4]
[1,]    2    7   27    7
[2,]    5   10   30   10
[3,]    4    9   29    9
```

Extracting sub-matrices using logical vectors

```
> x<-matrix(1:30, ncol=6)
> sel1<- c(F,F,T,T,F)
> sel2<-c(T,F,F,T,F,T)
> x[sel1,sel2]
     [,1] [,2] [,3]
[1,]    3   18   28
[2,]    4   19   29
> z1<-rnorm(5)
> z2<-rnorm(6)
> x[z1>0, z2<0]
   [,1] [,2] [,3]
[1,]    1   21   26
[2,]    2   22   27
[3,]    5   25   30
```

Removing rows and/or columns

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

```
> x[,-c(3,4)]
     [,1] [,2]
[1,]    1    2
[2,]    5    6
[3,]    9   10
> x[-c(1:2),-c(3,4)]
[1]  9 10
```

Submatrices using combinations of different types of syntax

```
> x[ c(1,3), -2]
     [,1] [,2] [,3] [,4] [,5]
[1,]    1   11   16   21   26
[2,]    3   13   18   23   28
> x[ c(T,T,F,F,F), c(3,1)]
     [,1] [,2]
[1,]   11    1
[2,]   12    2
```

```
> index<-1:6
> x[ -3, index>3]
     [,1] [,2] [,3]
[1,]   16   21   26
[2,]   17   22   27
[3,]   19   24   29
[4,]   20   25   30
```

Extracting elements under logical conditions

```
> x[ x>40 ]
integer(0)
> x[ x>20 ]
 [1] 21 22 23 24 25 26 27 28 29 30
> x[x>20]<-0
> x
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    6   11   16    0    0
[2,]    2    7   12   17    0    0
[3,]    3    8   13   18    0    0
[4,]    4    9   14   19    0    0
[5,]    5   10   15   20    0    0
```

Changing values of elements selected by using logical conditions

```
z<- c(4, 5, 5, 4, 4, 2, 3, 5, 2, 3, 7, 4, 7, 4, 7, 5, 7, 7, 5, 6)
dim(z) <- c(5,4)
z
> z[z==4]
[1] 4 4 4 4 4
> z[z==4]<-0
> z
     [,1] [,2] [,3] [,4]
[1,]    0    2    7    5
[2,]    5    3    0    7
[3,]    5    5    7    7
[4,]    0    2    0    5
[5,]    0    3    7    6
```

```
> z[z==3]
[1] 3 3
> z[z==3]<-c(333,333333)
> z
     [,1]   [,2] [,3] [,4]
[1,]    0      2    7    5
[2,]    5    333    0    7
[3,]    5      5    7    7
[4,]    0      2    0    5
[5,]    0 333333    7    6
```

Which function in matrices

```
> x<-c(19, 14, 15, 14, 20, 20, 24, 19, 20, 15, 28, 17, 18, 21, 31,
+ 16, 17, 22, 18, 14, 13, 26, 20, 20, 21, 24, 13, 16, 27, 17)
> x<-matrix(x, nrow=5, ncol=6)
> z<-   x>20
# Δίνει τη θέση στο διάνυσμα όπου το z είναι TRUE (το αποτέλεσμα είναι
    ένα διάνυσμα)
> which(z)
[1]  7 11 14 15 18 22 25 26 29
> which(z, arr.in=T)
     row col
 [1,]   2   2
 [2,]   1   3
 [3,]   4   3
 [4,]   5   3
 [5,]   3   4
 [6,]   2   5
 [7,]   5   5
 [8,]   1   6
 [9,]   4   6
```

Which function in matrices

```
> z
     [,1]  [,2]  [,3]  [,4]  [,5]  [,6]
[1,] FALSE FALSE  TRUE FALSE FALSE  TRUE
[2,] FALSE  TRUE FALSE FALSE  TRUE FALSE
[3,] FALSE FALSE FALSE  TRUE FALSE FALSE
[4,] FALSE FALSE  TRUE FALSE FALSE  TRUE
[5,] FALSE FALSE  TRUE FALSE  TRUE FALSE
> dim(z)
[1] 5 6
> which(z)
[1]  7 11 14 15 18 22 25 26 29
```

```
> which(z, arr.in=T)
     row col
[1,]   2   2
[2,]   1   3
[3,]   4   3
[4,]   5   3
[5,]   3   4
[6,]   2   5
[7,]   5   5
[8,]   1   6
[9,]   4   6
```

Creating matrices with same rows/columns

```
> z<-c(4,3,5,1,4,7,23)
> k<-3
> matrix(z,length(z),k)
     [,1] [,2] [,3]
[1,]    4    4    4
[2,]    3    3    3
[3,]    5    5    5
[4,]    1    1    1
[5,]    4    4    4
[6,]    7    7    7
[7,]   23   23   23

> matrix(z,k, length(z),byrow=T)
     [,1] [,2] [,3] [,4] [,5] [,6] [,7]
[1,]    4    3    5    1    4    7   23
[2,]    4    3    5    1    4    7   23
[3,]    4    3    5    1    4    7   23
```

More Matrix Input Commands

1) We can also create a matrix as a row/column combination of vectors:

```
> rbind(c(56,2,8),c(16,14,7))
     [,1] [,2] [,3]
[1,]   56    2    8
[2,]   16   14    7

> cbind(c(56,2,8),c(16,14,7))
     [,1] [,2]
[1,]   56   16
[2,]    2   14
[3,]    8    7
```

The `rbind()` and `cbind()` functions can also be used to append matrices together.

Combining vector and matrices

```
> z<-cbind(math,gender); z
     math gender
 [1,]    6      2
 [2,]    7      1
 [3,]    7      1
 [4,]   10      1
 [5,]    7      1
 [6,]    7      2
 [7,]    9      2
 [8,]    6      1
 [9,]   10      2
[10,]    9      1
> is.matrix(z)
[1] TRUE
> dim(z)
[1] 10  2
```

```
> z<-rbind(math,gender)
> is.matrix(z)
[1] TRUE
> dim(z)
[1]  2 10
> z
       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
math      6    7    7   10    7    7    9    6   10     9
gender    2    1    1    1    1    2    2    1    2     1
>
```

Combining vector and matrices

```
> x<-c(5,3,4,7)
> y<-c(-2,-1,0,4)
> cbind(x,y)
     x  y
[1,] 5 -2
[2,] 3 -1
[3,] 4  0
[4,] 7  4

> cbind(y,x)
     y x
[1,] -2 5
[2,] -1 3
[3,]  0 4
[4,]  4 7
```

```
> rbind(x,y,x)
   [,1] [,2] [,3] [,4]
x    5    3    4    7
y   -2   -1    0    4
x    5    3    4    7
```

Combining vector and matrices

```
> x<-c(5,3,4)
> y<-c(-2,-1,0,4,-2)
> cbind(x,y)
     x  y
[1,] 5 -2
[2,] 3 -1
[3,] 4  0
[4,] 5  4
[5,] 3 -2
Warning message:
In cbind(x, y) :
  number of rows of result is not a multiple of vector length (arg 1)
```

```
> cbind(y,x)
      y x
[1,] -2 5
[2,] -1 3
[3,]  0 4
[4,]  4 5
[5,] -2 3
Warning message:
In cbind(y, x) :
  number of rows of result is not a multiple of vector length (arg 2)
```

Diagonal Matrices

BE CAREFUL: The diag command changes behavior according to the input

- **x=number** => Identity matrix of dimensions x times x
- x**=vector** => Diagonal matrix with the elements of x placed on the diagonal.
- **x=matrix** => vector with the diagonal elements of x

Diagonal Matrices

- **x=number** => Identity matrix of dimensions x times x

```
> diag(5)
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    0    0    0    0
[2,]    0    1    0    0    0
[3,]    0    0    1    0    0
[4,]    0    0    0    1    0
[5,]    0    0    0    0    1
```

Diagonal Matrices

- x**=vector** => Diagonal matrix with the elements of x placed on the diagonal.

```
> diag(1:5)
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    0    0    0    0
[2,]    0    2    0    0    0
[3,]    0    0    3    0    0
[4,]    0    0    0    4    0
[5,]    0    0    0    0    5
```

Diagonal Matrices

- **x=matrix** => vector with the diagonal elements of x

```
x<-matrix(1:30, ncol=6)
> diag( x )
[1]  1  7 13 19 25
```

diag(diag(x)) => ?

Diagonal Matrices

```
> x<-matrix(1:25,5,5); x
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    6   11   16   21
[2,]    2    7   12   17   22
[3,]    3    8   13   18   23
[4,]    4    9   14   19   24
[5,]    5   10   15   20   25
>
> diag(x)
[1]  1  7 13 19 25
```

```
> diag(diag(x))
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    0    0    0    0
[2,]    0    7    0    0    0
[3,]    0    0   13    0    0
[4,]    0    0    0   19    0
[5,]    0    0    0    0   25
```

Diagonal Matrices

```
> x<-matrix(1:30,5,6)
> diag(diag(x), nrow(x), ncol(x))
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    0    0    0    0    0
[2,]    0    7    0    0    0    0
[3,]    0    0   13    0    0    0
[4,]    0    0    0   19    0    0
[5,]    0    0    0    0   25    0
```

```
> y<-x
> diag(y)
[1]  1  7 13 19 25
> diag(y)<-0
> x-y
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    0    0    0    0    0
[2,]    0    7    0    0    0    0
[3,]    0    0   13    0    0    0
[4,]    0    0    0   19    0    0
[5,]    0    0    0    0   25    0
```

Diagonal Matrices

```
> x<-matrix(rpois(30,5),5,6)
> y<-matrix(0, 5,6)
> diag(y) <- diag(x)
> y
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    5    0    0    0    0    0
[2,]    0    2    0    0    0    0
[3,]    0    0    4    0    0    0
[4,]    0    0    0    8    0    0
[5,]    0    0    0    0    7    0
>
```

Elementwise numerical operations for matrices

Arithmetic Operators & matrices: +, - , *, /, %/%, %%

Can be used with matrices of the same dimension

Each operation is applied in each element corresponding to the same row and column

Elementwise numerical operations for matrices

```
> x1<-matrix(rbinom(12,10,0.5), ncol=3,nro=4)
> x2<-matrix(1:12, ncol=3,nro=4)
> x1
     [,1] [,2] [,3]
[1,]    4    4    2
[2,]    3    7    7
[3,]    4    5    6
[4,]    6    6    7
> x2
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
> x1+x2
     [,1] [,2] [,3]
[1,]    5    9   11
[2,]    5   13   17
[3,]    7   12   17
[4,]   10   14   19
```

Elementwise numerical operations for matrices

```
> x1<-matrix(rbinom(12,10,0.5), ncol=3,nro=4)
> x2<-matrix(1:12, ncol=3,nro=4)
> x1
     [,1] [,2] [,3]
[1,]    4    4    2
[2,]    3    7    7
[3,]    4    5    6
[4,]    6    6    7
> x2
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
```

```
> x1*x2
     [,1] [,2] [,3]
[1,]    4   20   18
[2,]    6   42   70
[3,]   12   35   66
[4,]   24   48   84
> x1/x2
         [,1]      [,2]      [,3]
[1,] 4.000000 0.8000000 0.2222222
[2,] 1.500000 1.1666667 0.7000000
[3,] 1.333333 0.7142857 0.5454545
[4,] 1.500000 0.7500000 0.5833333
```

Elementwise numerical operations for matrices

```
> x1<-matrix(rbinom(12,10,0.5), ncol=3,nro=4);x1
     [,1] [,2] [,3]
[1,]    2    5    4
[2,]    7    2    8
[3,]    8    4    6
[4,]    8    6    5
> x1^2
     [,1] [,2] [,3]
[1,]    4   25   16
[2,]   49    4   64
[3,]   64   16   36
[4,]   64   36   25
```

```
> x2<-matrix(1:12, ncol=3,nro=4);x2
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
> x1^x2
     [,1]    [,2]       [,3]
[1,]    2    3125     262144
[2,]   49      64 1073741824
[3,]  512   16384  362797056
[4,] 4096 1679616  244140625
```

Elementwise numerical operations for matrices

```
> x1<-matrix(rbinom(12,10,0.5), ncol=3,nro=4);x1
     [,1] [,2] [,3]
[1,]    2    5    4
[2,]    7    2    8
[3,]    8    4    6
[4,]    8    6    5
> x1^2
     [,1] [,2] [,3]
[1,]    4   25   16
[2,]   49    4   64
[3,]   64   16   36
[4,]   64   36   25
```

```
> x2<-matrix(1:12, ncol=3,nro=4);x2
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
> x1^x2
     [,1]    [,2]       [,3]
[1,]    2    3125     262144
[2,]   49      64 1073741824
[3,]  512   16384  362797056
[4,] 4096 1679616  244140625
```

Checking the compatibility of dimensions

```
> x1^(t(x2))
Error in x1^(t(x2)) : non-conformable arrays
> x3<-t(x2)
> #
> # checking the equality of dimensions
> dim(x1)==dim(x2)
[1] TRUE TRUE
> all(dim(x1)==dim(x2))
[1] TRUE
> all(dim(x1)==dim(x3))
[1] FALSE
> all(dim(x1)==dim(t(x3)))
[1] TRUE
```

Elementwise arithmetic functions

```
> log(x1)
         [,1]      [,2]     [,3]
[1,] 0.6931472 1.6094379 1.386294
[2,] 1.9459101 0.6931472 2.079442
[3,] 2.0794415 1.3862944 1.791759
[4,] 2.0794415 1.7917595 1.609438
> (log(x1)+t(x2)-x1)/(t(x2)-x1)
Error in log(x1) + t(x2) : non-conformable arrays
> gamma(x1)
     [,1] [,2] [,3]
[1,]    1   24    6
[2,]  720    1 5040
[3,] 5040    6  120
[4,] 5040  120   24
```

Operators and functions for matrices

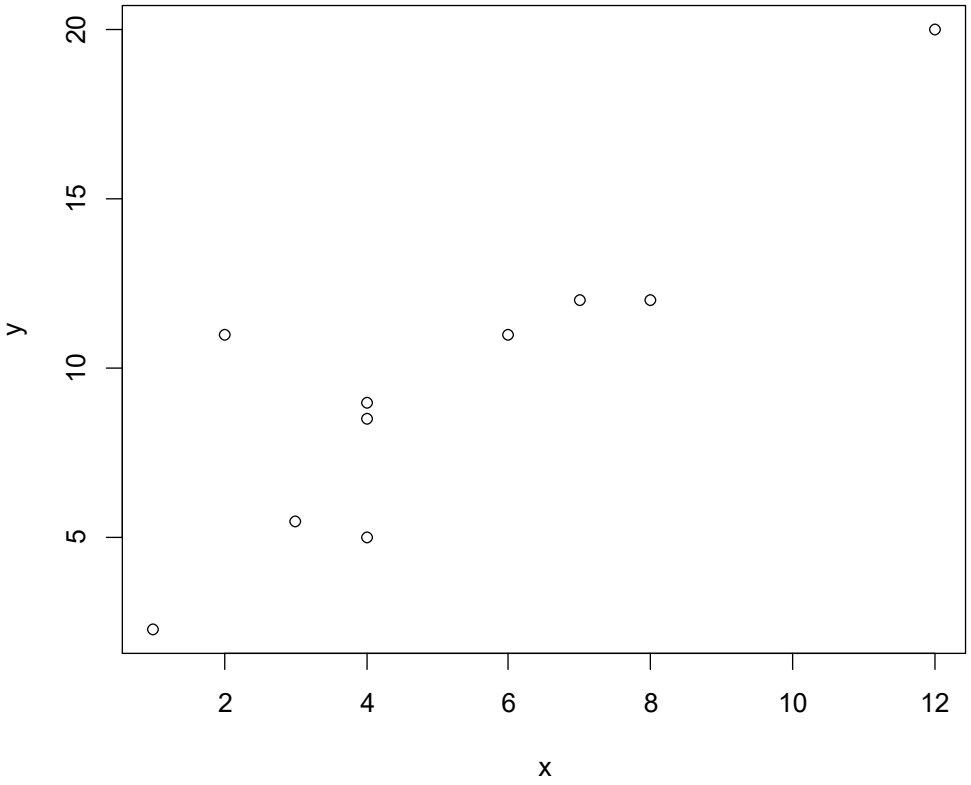
Operator	Description
%*%	Multiplication of Matrices (Πολλαπλασιασμός πινάκων)
t()	Transpose of a Matrix (ανάστροφος)
solve()	Inverse of a Matrix (αντίστροφος)
det() determinant()	Determinant of a matrix (ορίζουσα)
trace()	Trace of a matric
eigen()	Eigenvalues and eigenvectors

Operators and functions for matrices

```
> x<-matrix(c(1,2,3,4,5,6), ncol=2)
> x
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> dim(x)
[1] 3 2
> y<-matrix(c(0,1,1,1), ncol=2)
> y
     [,1] [,2]
[1,]    0    1
[2,]    1    1
```

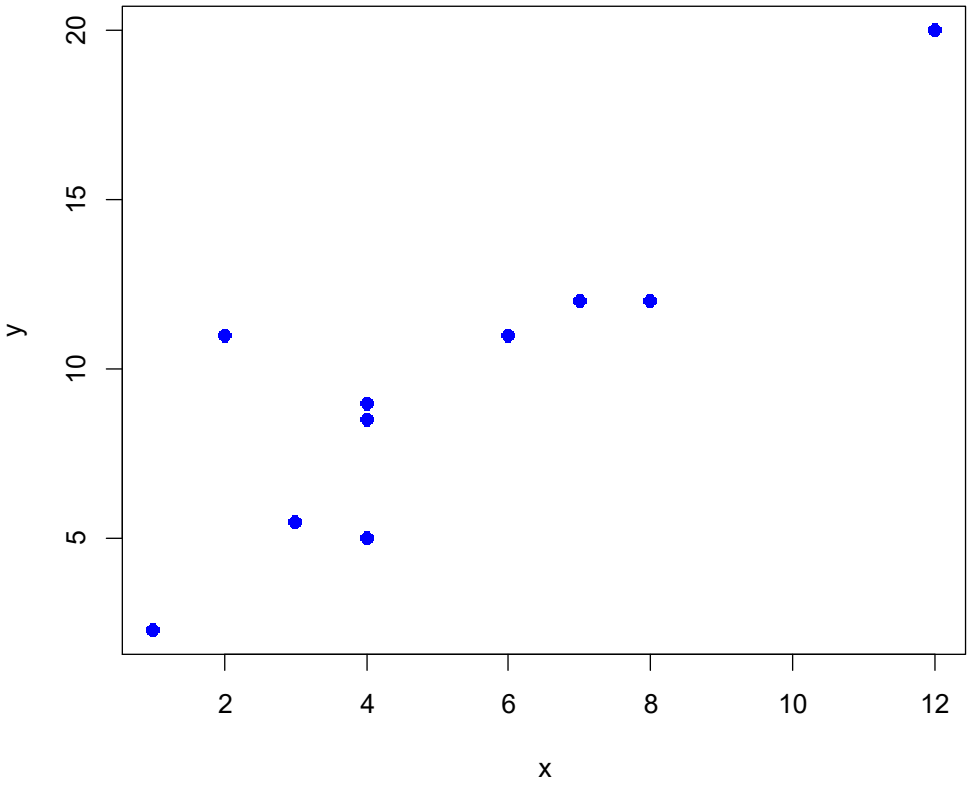
```
> x%*%y
     [,1] [,2]
[1,]    4    5
[2,]    5    7
[3,]    6    9
> t(x)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
> solve(y)
     [,1] [,2]
[1,]   -1    1
[2,]    1    0
```

Best fitted line using matrices



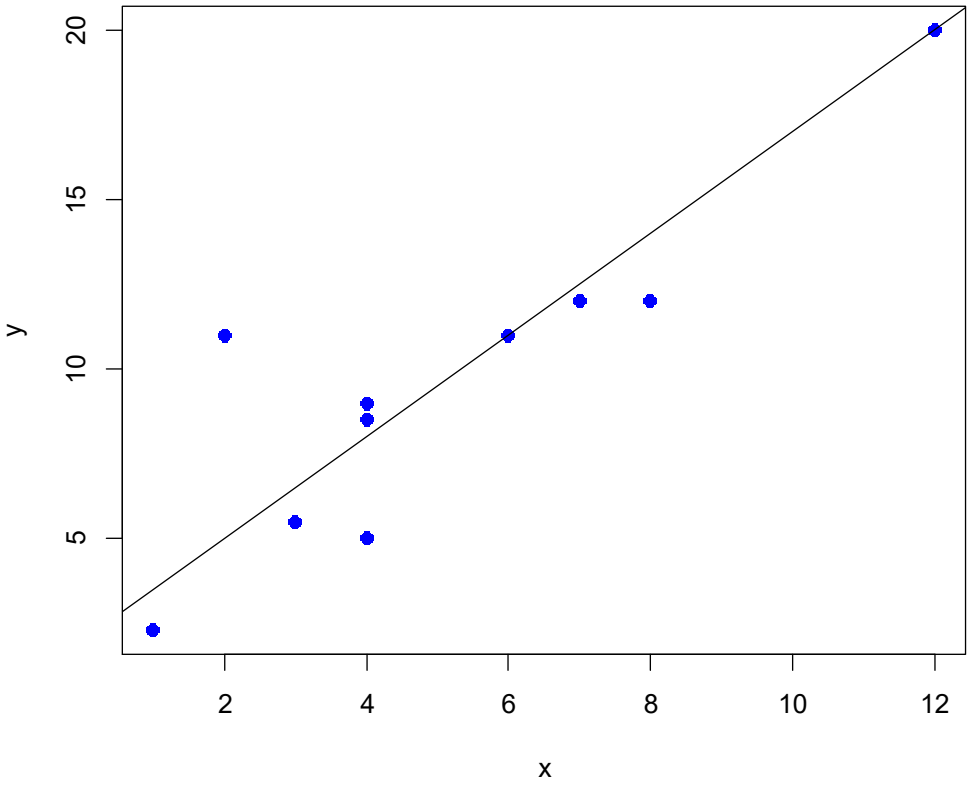

```
> x<- c(4,6,7,12,4,3,2,4,1,8)
> y<- c(8.5,11,12,20,9,5.5,
    11,5,2.3,12)
> plot(x,y)
# διάγραμμα σημείων
```

Best fitted line using matrices



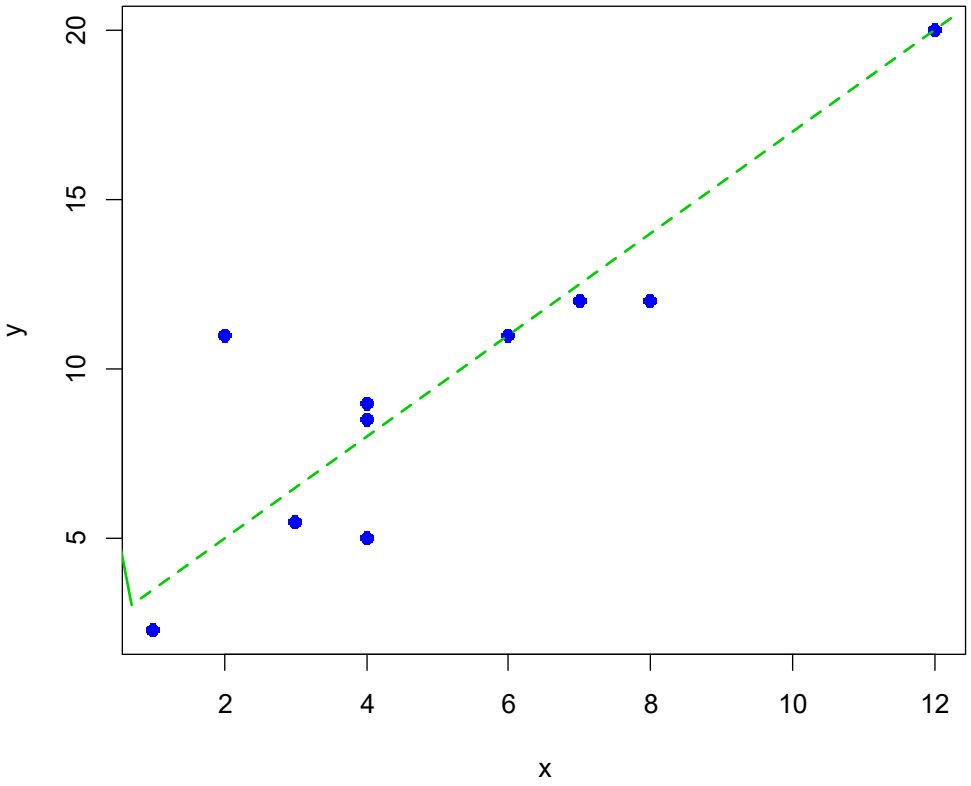

```
> x<- c(4,6,7,12,4,3,2,4,1,8)
> y<- c(8.5,11,12,20,9,5.5,
    11,5,2.3,12)
plot(x,y, pch=16, cex=1.1,
    col='blue')

# pch: type of point
# cex: size of points
# col: color
```

Best fitted line using matrices



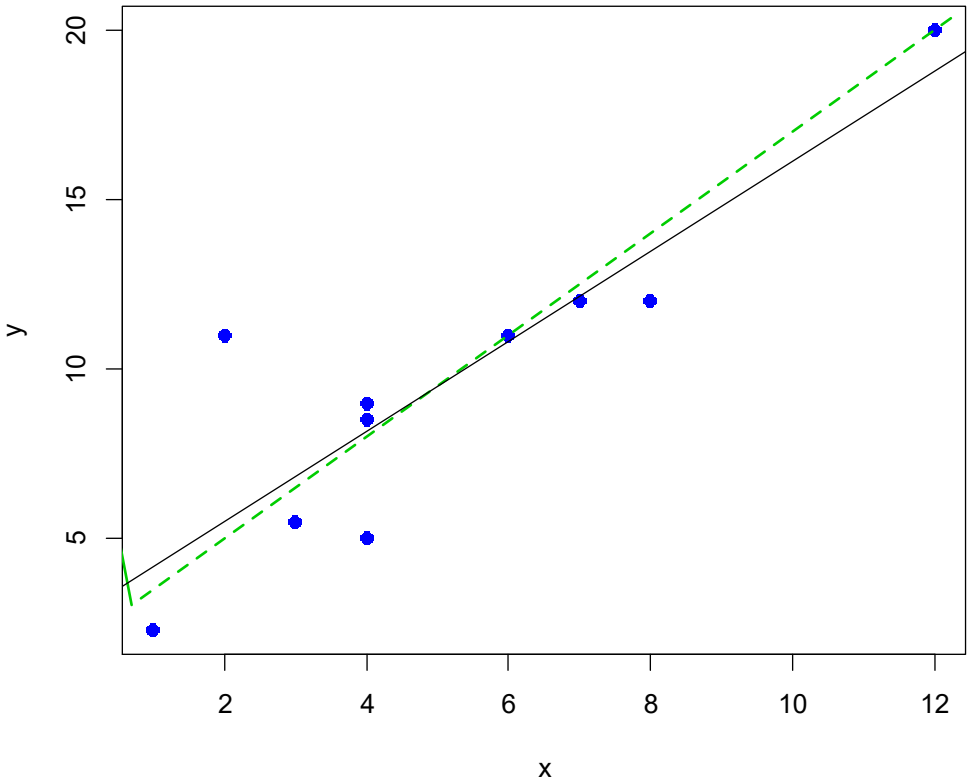

```
> x<- c(4,6,7,12,4,3,2,4,1,8)
> y<- c(8.5,11,12,20,9,5.5,
    11,5,2.3,12)
plot(x,y, pch=16, cex=1.1,
    col='blue')
abline(a=2,b=1.5)
# adds the line y=a+bx
```

Best fitted line using matrices



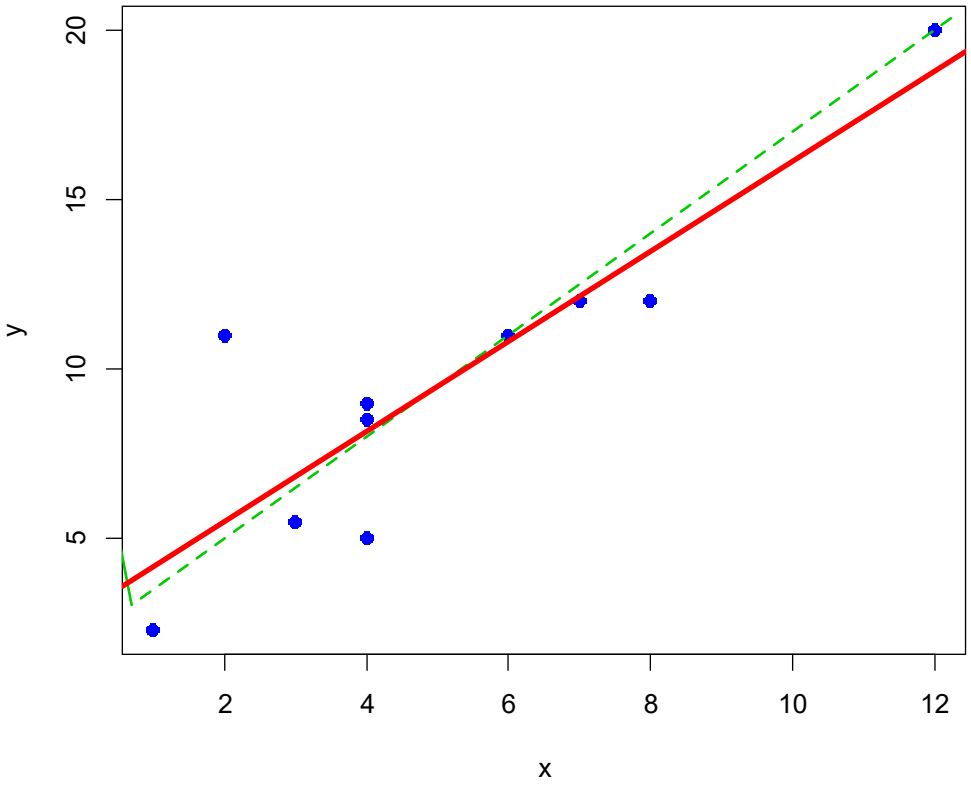

```
> x<- c(4,6,7,12,4,3,2,4,1,8)
> y<- c(8.5,11,12,20,9,5.5,
    11,5,2.3,12)
plot(x,y, pch=16, cex=1.1,
    col='blue')
abline(a=2,b=1.5, col=3,
    lwd=2, lty=2)
```

Best fitted line using matrices




```
> x<- c(4,6,7,12,4,3,2,4,1,8)
> y<- c(8.5,11,12,20,9,5.5,
    11,5,2.3,12)
plot(x,y, pch=16, cex=1.1,
    col='blue')
abline(a=2,b=1.5, col=3,
    lwd=2, lty=2)
> lm(y~x)
Call:
lm(formula = y ~ x)
Coefficients:
(Intercept)          x
     2.876      1.324
> abline(lm(y~x)) # add lse line
```

Best fitted line using matrices

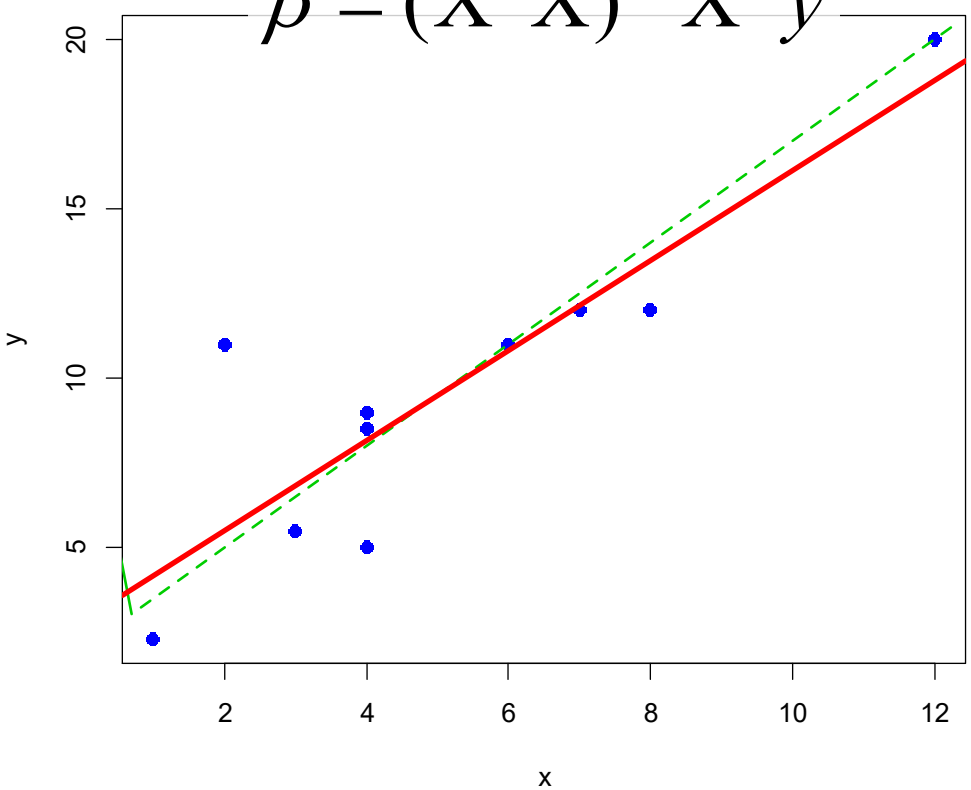



```
> x<- c(4,6,7,12,4,3,2,4,1,8)
> y<- c(8.5,11,12,20,9,5.5,
    11,5,2.3,12)
plot(x,y, pch=16, cex=1.1,
    col='blue')
abline(a=2,b=1.5, col=3,
    lwd=2, lty=2)
abline(lm(y~x), col='red',
    lwd=3.5)
```

Best fitted line using matrices

We can calculate the least square error coefficients of the regression line by using the formula:

$$\hat{\beta} = (\mathrm{X}^{\mathrm{T}}\mathrm{X})^{-1}\mathrm{X}^{\mathrm{T}}y$$



y is the vector of length n with all values of y

X is a nx2 matrix with the first column equal to one and the second equal to the values of x

$$X = \begin{bmatrix} 1 & x_1 \\ 1 & x_1 \\ \vdots & \vdots \\ 1 & x_n \end{bmatrix}$$

Then

$$y = X\beta$$

With elements

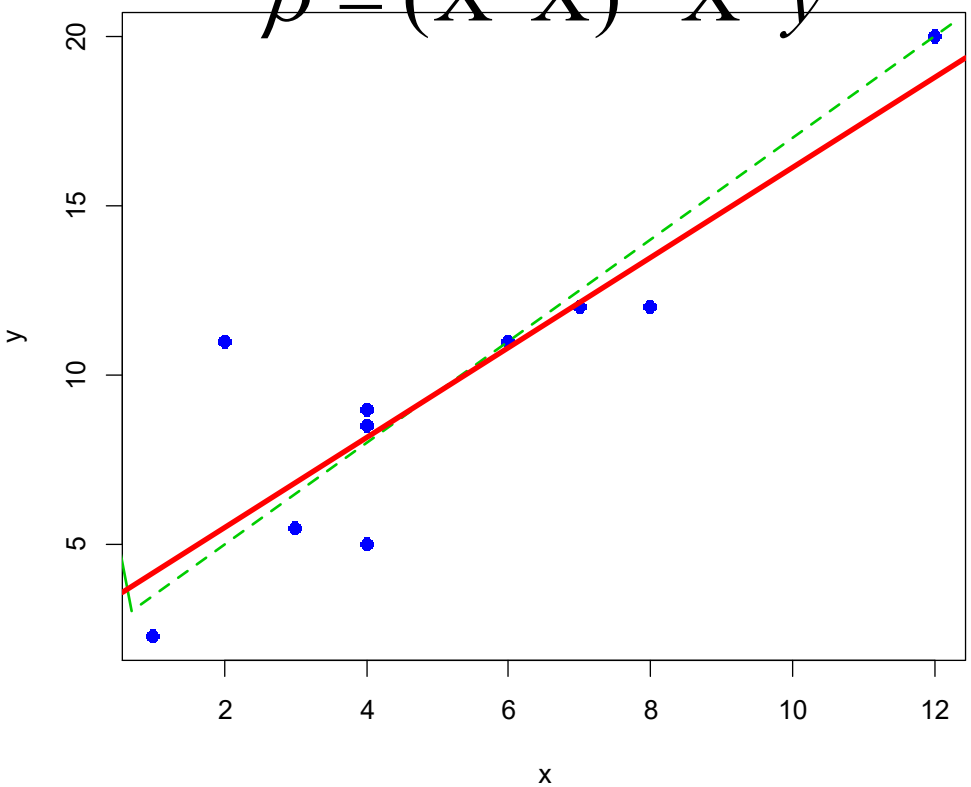
$$y_i = \beta_1 + \beta_2 x_i$$

Therefore β_1 is the intercept (a in abline) and β_2 is the slope (b in abline)

Best fitted line using matrices

We can calculate the least square error coefficients of the regression line by using the formula:

$$\hat{\beta} = (X^T X)^{-1} X^T y$$




```
> x
 [1]  4  6  7 12  4  3  2  4  1  8
> X <- cbind(1,x)
> X
      x
 [1,] 1  4
 [2,] 1  6
 [3,] 1  7
 [4,] 1 12
 [5,] 1  4
 [6,] 1  3
 [7,] 1  2
 [8,] 1  4
 [9,] 1  1
[10,] 1  8
> as.vector(solve(t(X)%*%X)%*%t(X)%*%y)
[1] 2.876396 1.324236
> lm(y~x)

Call:
lm(formula = y ~ x)

Coefficients:
(Intercept)            x
      2.876        1.324
```

Matrix Multiplication (in detail)

```
>  x1 <- matrix( 1:6, 3,2 )
>  x2 <- matrix( 1:8, 2,4 )
> dim(x1)
[1] 3 2
> dim(x2)
[1] 2 4
> #
> dim(x1)[2] == dim(x2)[1]
[1] TRUE
> ncol(x1)==nrow(x2)
[1] TRUE
> # Matrix multiplication
> x1%*%x2
     [,1] [,2] [,3] [,4]
[1,]    9   19   29   39
[2,]   12   26   40   54
[3,]   15   33   51   69
```

```
> # element 1,1
> sum(x1[1,]*x2[,1])
[1] 9
> # element 1,2
> sum(x1[1,]*x2[,2])
[1] 19
> # element 1,3
> sum(x1[1,]*x2[,3])
[1] 29
> # element i,j
> i<-3;  j<-4
> sum(x1[i,]*x2[,j])
[1] 69
```

Matrix Determinant

```
> z<-matrix( rnorm(16), 4,4) ;z
           [,1]       [,2]       [,3]       [,4]
[1,] -0.01408545  0.02668823 -0.7546984 -0.19378219
[2,] -0.13434812 -1.15283863 -0.1818433  0.09941875
[3,] -0.34249532  0.02157407 -0.8390066 -1.05998029
[4,] -1.21523318 -0.83692523  0.8401202  0.79445761
> det(z)
[1] 0.9791742
> z2 <-z; z2[,4] <- 4*z2[,1]-2*z2[,2]+0.5*z2[,3]
> z2
           [,1]       [,2]       [,3]       [,4]
[1,] -0.01408545  0.02668823 -0.7546984 -0.4870674
[2,] -0.13434812 -1.15283863 -0.1818433  1.6773631
[3,] -0.34249532  0.02157407 -0.8390066 -1.8326327
[4,] -1.21523318 -0.83692523  0.8401202 -2.7670222
```

```
> solve(z2)
Error in solve.default(z2) :
  system is computationally
    singular: reciprocal condition
    number = 2.18847e-18
> det(z2)
[1] 1.634387e-16
> z3<-z2
> z3[,4] <- z3[,4] + rnorm(4,0,0.00001)
> det(z3)
[1] 1.486288e-05
> z4<-z2
> z4[,4] <- z3[,4] + rnorm(4,0,0.01)
> det(z4)
[1] -0.004201555
```

Matrix Determinant (2)

```
> # logarithm of the determinant of z
> determinant(z, logarithm = TRUE)
$`modulus`
[1] -0.02104573
attr(,"logarithm")
[1] TRUE

$sign
[1] 1

attr(,"class")
[1] "det“
```

```
> # determinant of z
> determinant(z, logarithm = F)
$`modulus`
[1] 0.9791742
attr(,"logarithm")
[1] FALSE

$sign
[1] 1

attr(,"class")
[1] "det"
>
```

Matrix Determinant (3)

```
> det(z)
[1] 0.9791742
> detz <- determinant(z, logarithm = F)

> mode(detz);class(detz)
[1] "list"
[1] "det“

> #
> # Absolute value of the determinant
> detz$modulus
[1] 0.9791742
attr(,"logarithm")
[1] FALSE
> detz$sign
[1] 1
```

```
> detz$modulus*detz$sign
[1] 0.9791742
attr(,"logarithm")
[1] FALSE
> # getting only the value
> detz$modulus[[1]]*detz$sign
[1] 0.9791742
> detz <- determinant(z)
> detz$sign * exp( detz$mod[[1]] )
[1] 0.9791742
> det(z)
[1] 0.9791742
```

Determinant of 2x2 matrix

```
> test<- matrix(c(1,0.07,0.07,1),2,2)
> test
     [,1] [,2]
[1,] 1.00 0.07
[2,] 0.07 1.00
> det(test)
[1] 0.9951
> test[1,1]*test[2,2]-test[1,2]*test[2,1]
[1] 0.9951
>
```

Trace of a matrix

The trace function is used for other operation

trace {base} R Documentation

Interactive Tracing and Debugging of Calls to a Function or Method

Description

A call to `trace` allows you to insert debugging code (e.g., a call to `browser` or `recover`) at chosen places in any function. A call to `untrace` cancels the tracing. Specified methods can be traced the same way, without tracing all calls to the generic function. Trace code (`tracer`) can be any R expression. Tracing can be temporarily turned on or off globally by calling `tracingState`.

Usage

```
trace(what, tracer, exit, at, print, signature,
      where = topenv(parent.frame()), edit = FALSE)
untrace(what, signature = NULL, where = topenv(parent.frame()))

tracingState(on = NULL)
.doTrace(expr, msg)
returnValue(default = NULL)
```

You can see the builtin used objects with apropos('*')

> 'trace' %in% apropos('*')

[1] TRUE

Trace of a matrix

```
> A<-matrix( rnorm(9), 3 )
> # install.packages('psych')
> library(psych)
> tr(A)
[1] -0.8617493
>
> sum(diag(A))
[1] -0.8617493
> trace <- function ( A ) { sum(diag(A)) }
> trace(A)
[1] -0.8617493
```

```
> trace
function ( A ) { sum(diag(A)) }
> rm(trace)
> # Here we save  only the result
> trace <- sum(diag(A))
> trace
[1] -0.8617493
> rm(trace)
> mytrace <- function ( A ) { sum(diag(A)) }
> mytrace(A)
[1] -0.8617493
```

Logical matrices

```
> A <- round(matrix( rnorm(16), 4 ),1)
> A
     [,1] [,2] [,3] [,4]
[1,]  1.0  0.3  0.1 -1.7
[2,]  0.8  0.3  1.5 -1.0
[3,] -0.9 -0.6 -0.1  1.0
[4,] -0.3  0.8  0.5 -1.6
> logicA <- A>0
> logicA
      [,1]  [,2]  [,3]  [,4]
[1,]  TRUE  TRUE  TRUE FALSE
[2,]  TRUE  TRUE  TRUE FALSE
[3,] FALSE FALSE FALSE  TRUE
[4,] FALSE  TRUE  TRUE FALSE
>
```

```
> mode(A);class(A)
[1] "numeric"
[1] "matrix"
> mode(logicA);class(logicA)
[1] "logical"
[1] "matrix"
>
> logicB <- matrix( c(T,T,F,T), 2 )
> logicB
     [,1]  [,2]
[1,] TRUE FALSE
[2,] TRUE  TRUE
```

Checking the equality of matrices

```
> A<-matrix(1:30, 5,6)
> B<-matrix(1:30, 5,6)
> A==B
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,] TRUE TRUE TRUE TRUE TRUE TRUE
[2,] TRUE TRUE TRUE TRUE TRUE TRUE
[3,] TRUE TRUE TRUE TRUE TRUE TRUE
[4,] TRUE TRUE TRUE TRUE TRUE TRUE
[5,] TRUE TRUE TRUE TRUE TRUE TRUE
> all(A==B)
[1] TRUE
```

Checking the equality of matrices

```
> x<-matrix(rnorm(9),3,3)
> x%*%solve(x)
             [,1] [,2]         [,3]
[1,]  1.000000e+00    0 -5.551115e-17
[2,]  0.000000e+00    1  0.000000e+00
[3,] -5.551115e-17    0  1.000000e+00
> x%*%solve(x)-diag(3)
             [,1] [,2]         [,3]
[1,]  0.000000e+00    0 -5.551115e-17
[2,]  0.000000e+00    0  0.000000e+00
[3,] -5.551115e-17    0  0.000000e+00
> abs(x%*%solve(x)-diag(3))<0.00001
     [,1] [,2] [,3]
[1,] TRUE TRUE TRUE
[2,] TRUE TRUE TRUE
[3,] TRUE TRUE TRUE
> all(abs(x%*%solve(x)-diag(3))<0.00001)
[1] TRUE
```

```
> round(x%*%solve(x),10)==diag(3)
     [,1] [,2] [,3]
[1,] TRUE TRUE TRUE
[2,] TRUE TRUE TRUE
[3,] TRUE TRUE TRUE
> all(round(x%*%solve(x),10)==diag(3))
[1] TRUE
```

Character matrices

```
> letters[1:12]
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l"
> charA <- matrix( letters[1:12],3 )
> charA
     [,1] [,2] [,3] [,4]
[1,] "a"  "d"  "g"  "j"
[2,] "b"  "e"  "h"  "k"
[3,] "c"  "f"  "i"  "l"
>
> mode(charA); class(charA)
[1] "character"
[1] "matrix"
>
```

```
> logicB
     [,1]  [,2]
[1,] TRUE FALSE
[2,] TRUE  TRUE
> as.character(logicB)
[1] "TRUE"  "TRUE"  "FALSE" "TRUE"
> charB <- matrix( as.character(logicB),2 )
> charB
     [,1]   [,2]
[1,] "TRUE" "FALSE"
[2,] "TRUE" "TRUE
```

Sorting in matrices

```
> onoma <- c( 'Giorgos', 'Yiannis','Maria', 'Eleni' )
> age <- c( 31,21,22,17 )
> male <- c( 1,1,0,0 )
> A <- cbind(age,male)
> A
     age male
[1,]  31    1
[2,]  21    1
[3,]  22    0
[4,]  17    0
> rownames( A ) <- onoma
> A
        age male
Giorgos  31    1
Yiannis  21    1
Maria    22    0
Eleni    17    0
```

```
> B <- cbind(sort(A[,1]), A[,2])
> B
        [,1] [,2]
Eleni     17    1
Yiannis   21    1
Maria     22    0
Giorgos   31    0
```

```
> B <- cbind(sort(A[,1]), sort(A[,2]))
> B
        [,1] [,2]
Eleni     17    0
Yiannis   21    0
Maria     22    1
Giorgos   31    1
```

```
> A[order(A[,1]),]
        age male
Eleni    17    0
Yiannis  21    1
Maria    22    0
Giorgos  31    1
```

Sorting in matrices

```
> order(A[,1])
[1] 4 2 3 1
> i <- order(A[,1])
> A[i,]
        age male
Eleni    17    0
Yiannis  21    1
Maria    22    0
Giorgos  31    1
```

```
# Ordering by names
> rownames(A)
[1] "Giorgos" "Yiannis" "Maria"   "Eleni"
> sort(rownames(A))
[1] "Eleni"   "Giorgos" "Maria"   "Yiannis"
> i <- order(rownames(A))
> i
[1] 4 1 3 2
> A[i,]
        age male
Eleni    17    0
Giorgos  31    1
Maria    22    0
Yiannis  21    1
> A[i,1]
  Eleni Giorgos   Maria Yiannis
     17     31     22     21
> A[i,2]
  Eleni Giorgos   Maria Yiannis
      0      1      0      1
```

Sorting in matrices

```
> B<-t(A)
> B
     Giorgos Yiannis Maria Eleni
age       31      21    22    17
male       1       1     0     0
> i<-order( B[1,] )
> i
[1] 4 2 3 1
> B[,i]
     Eleni Yiannis Maria Giorgos
age     17      21    22      31
male     0       1     0       1
```

Data Analysis

Data analysis involves number crunching and this chapter is about functions that let us take data in great gulps and process the whole gulp at once.

An inconvenient aspect of **`mean`** and **`sd`** is when they are given data in a matrix (or data frame):

```
> head(women)
  height weight
1     58    115
2     59    117
3     60    120
4     61    123
5     62    126
6     63    129
> mean(women)
 [1] NA
Warning message:
In mean.default(women) : argument is not numeric or logical: returning NA
```

Operations can only be applied to vectors, and so must be applied to each column in turn. This is the purpose of the *apply* functions: **`apply`**, **`lapply`**, **`sapply`** and **`tapply`**.

The `apply` function

The **`apply`** function allows functions to operate on specified parts of an array. For example, the matrix `state.x77` contains eight columns of data describing the 50 U.S. states in 1977:

```
> state.x77
          Population Income Illiteracy Life Exp Murder HS Grad Frost   Area
Alabama         3615   3624        2.1    69.05   15.1    41.3    20  50708
Alaska           365   6315        1.5    69.31   11.3    66.7   152 566432
Arizona         2212   4530        1.8    70.55    7.8    58.1    15 113417
Arkansas        2110   3378        1.9    70.66   10.1    39.9    65  51945
California     21198   5114        1.1    71.71   10.3    62.6    20 156361
Colorado        2541   4884        0.7    72.06    6.8    63.9   166 103766
…
```

Suppose we want the median for each variable. We can use **`apply`**:

```
apply(x = array, MARGIN = vector, FUN = fun, …)
```

where

1. `x` is the name of the array

2. `MARGIN` is an integer vector specifying the sections of the array (dimensions we want to keep) :

   `MARGIN = 1` corresponds to rows

   `MARGIN = 2` corresponds to columns etc.

3. `FUN` is the function, or name of the function we want to apply to each section;

4. … are any additional arguments needed by `FUN`.

```
> apply(state.x77, 2, median)
Population  Income Illiteracy  Life Exp  Murder  HS Grad   Frost     Area
   2838.50  4519.0       0.95   70.675    6.850   53.250  114.50 54277.00

> apply(state.x77, 2, quantile, c(0.25, 0.75))
    Population  Income Illiteracy Life Exp Murder HS Grad  Frost     Area
25%     1079.5 3992.75      0.625  70.1175  4.350   48.05  66.25 36985.25
75%     4968.5 4813.50      1.575  71.8925 10.675   59.15 139.75 81162.50
```

The apply() function: Further examples

```
> x<-matrix(c(1,2,3,4,5,6), ncol=2)
> x
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> apply(x,1,sum)
[1] 5 7 9
> apply(x,2,sum)
[1]  6 15
```

Compute row sum.

Compute column sum.

The apply() function

```
> x <- matrix(rnorm(200), 20, 10)
> apply(x, 2, mean)
[1] 0.04868268 0.35743615 -0.09104379
[4] -0.05381370 -0.16552070 -0.18192493
[7] 0.10285727 0.36519270 0.14898850
[10] 0.26767260

> apply(x, 1, sum)
[1] -1.94843314 2.60601195 1.51772391
[4] -2.80386816 3.73728682 -1.69371360
[7] 0.02359932 3.91874808 -2.39902859
[10] 0.48685925 -1.77576824 -3.34016277
[13] 4.04101009 0.46515429 1.83687755
[16] 4.36744690 2.21993789 2.60983764
[19] -1.48607630 3.58709251
```

simulates from standard normal (more later)

The apply() function

Quantiles of the rows of a matrix.

```
> x <- matrix(rnorm(200), 20, 10)
> apply(x, 1, quantile, probs = c(0.25, 0.75))
           [,1]       [,2]          [,3]          [,4]
25% -0.3304284 -0.99812467 -0.9186279 -0.49711686  ……..
75%  0.9258157 0.07065724 0.3050407 -0.06585436
```

Col/row sums & means

- rowSums (x) = apply(x, 1, sum)
- rowMeans (x) = apply(x, 1, mean)
- colSums (x) = apply(x, 2, sum)
- colMeans (x) = apply(x, 2, mean)

These shortcut functions are much faster, but you won't notice unless you're using a large matrix.

Arrays

- Arrays are matrices with 3 or more dimensions.
- To create them we use the function array().
- The dimension of the array is given by the parameter dim.
- For example if dim=c(2,3,4), we will have a 3 dimensional array of dimension 2×3×4.

Arrays

```
> X<-array(c(1:12,36:48),dim=c(2,3,4))
> X
, , 1

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

, , 2

     [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12

, , 3

     [,1] [,2] [,3]
[1,]   36   38   40
[2,]   37   39   41

, , 4

     [,1] [,2] [,3]
[1,]   42   44   46
[2,]   43   45   47
```

Attributes of arrays

```
> length(X)
[1] 24
> mode(X)
[1] "numeric"
> class(X)
[1] "array"
> dim(X)
[1] 2 3 4
> dimnames(X)
NULL
```

Array dim & dimnames

These functions are used exactly in the same way as in matrices

Dimnames have now length equal to the number of dimension (i.e. it is a list with 3 elements for X in our example and each element is a character vector with length equal to the corresponding elements of dim

Array dim & dimnames

```
> dim(X)
[1] 2 3 4
>
> dimnames(X)
NULL
> dimnames(X)[[1]] <- paste('row',1:2,sep='')
> X
, , 1
    [,1] [,2] [,3]
row1    1    3    5
row2    2    4    6

, , 2
    [,1] [,2] [,3]
row1    7    9   11
row2    8   10   12

, , 3
    [,1] [,2] [,3]
row1   36   38   40
row2   37   39   41

, , 4
    [,1] [,2] [,3]
row1   42   44   46
row2   43   45   47

> dimnames(X)
[[1]]
[1] "row1" "row2"

[[2]]
NULL

[[3]]
NULL
```

Array dim & dimnames

```
> dimnames(X)[[2]] <- paste('col',1:3,sep='')
> X
, , 1
     col1 col2 col3
row1    1    3    5
row2    2    4    6
 ..........................
```

```
> dimnames(X)
[[1]]
[1] "row1" "row2"

[[2]]
[1] "col1" "col2" "col3"

[[3]]
NULL
```

```
> dimnames(X)[[3]] <- paste('tab',1:4,sep='')
> X
, , tab1

     col1 col2 col3
row1    1    3    5
row2    2    4    6
..........................
```

```
> dimnames(X)
[[1]]
[1] "row1" "row2"

[[2]]
[1] "col1" "col2" "col3"

[[3]]
[1] "tab1" "tab2" "tab3" "tab4"
```

Array dim & dimnames

```
> attributes(X)
$dim
[1] 2 3 4

$dimnames
$dimnames[[1]]
[1] "row1" "row2"

$dimnames[[2]]
[1] "col1" "col2" "col3"

$dimnames[[3]]
[1] "tab1" "tab2" "tab3" "tab4"
```

```
> attributes(X)$dim
[1] 2 3 4
> attributes(X)$dimnames
[[1]]
[1] "row1" "row2"

[[2]]
[1] "col1" "col2" "col3"

[[3]]
[1] "tab1" "tab2" "tab3" "tab4"
```

Array dim & dimnames

```
> dimnames(X) <- NULL
> attributes(X)
$dim
[1] 2 3 4

> dimnames(X)
NULL
> #
> # setting dimnames using lists
> dimnames(X)<-list(
+ c('Row1', 'Row2'),
+ c('Col1','Col2','Col3'),
+ c('Table1','Table2','Table3','Table4')  )
```

```
> X
, , Table1
     Col1 Col2 Col3
Row1    1    3    5
Row2    2    4    6
.........................
, , Table4
     Col1 Col2 Col3
Row1   42   44   46
Row2   43   45   47

> dimnames(X)<-list( c('Row1', 'Row2'),
NULL, NULL )
> X
, , 1
     [,1] [,2] [,3]
Row1    1    3    5
Row2    2    4    6
.........................
```

Array Subsetting (similar to matrices)

In order to obtain an element you need to include within brackets the same number as the dimensions

```
> X[2,3,1]
[1] 6
```

Leaving one dimension empty, it means that you require all elements of this dimension

```
> X[,3,1]
[1] 5 6
> X[2,,1]
[1] 2 4 6
> X[2,3,]
[1]  6 12 41 47
```

Array Subsetting (similar to matrices)

```
> X[2,,]
     [,1] [,2] [,3] [,4]
[1,]    2    8   37   43
[2,]    4   10   39   45
[3,]    6   12   41   47
> X[,3,]
     [,1] [,2] [,3] [,4]
[1,]    5   11   40   46
[2,]    6   12   41   47
> X[,,1]
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

```
> X[2:1,2:3,c(3,1)]
, , 1

     [,1] [,2]
[1,]   39   41
[2,]   38   40

, , 2

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

> X[ ,2, c(T,F,T,F)]
     [,1] [,2]
[1,]    3   38
[2,]    4   39
```

Array Subsetting (using names)

```
> dimnames(X)<-list(
+ c('Row1', 'Row2'),
+ c('Col1','Col2','Col3'),
+ c('Table1','Table2','Table3','Table4')  )
> X["Row1",,]
    Table1 Table2 Table3 Table4
Col1      1      7     36      2
Col2      3      9     38      4
Col3      5     11     40      6
```

```
> X["Row1","Col2",]
Table1 Table2 Table3 Table4
     3      9     38      4
> X["Row1","Col2","Table3"]
[1] 38
> X["Row1","Col2",c("Table3","Table1")]
Table3 Table1
    38      3
```

Which in arrays

```
> X>4
, , Table1
     Col1  Col2 Col3
Row1 FALSE FALSE TRUE
Row2 FALSE FALSE TRUE

, , Table2
    Col1 Col2 Col3
Row1 TRUE TRUE TRUE
Row2 TRUE TRUE TRUE
…… ……................
…… ……................
```

```
> which(X>4)
 [1]  5  6  7  8  9 10 11 12 13 14 15 16 17 22 23 24
> X[which(X>4)]
 [1]  5  6  7  8  9 10 11 12 36 37 38 39 40  5  6  7
```

Which in arrays

```
> which(X>4, arr.ind=T)
     dim1 dim2 dim3
Row1    1    3    1
Row2    2    3    1
Row1    1    1    2
Row2    2    1    2
……................
……................
Row1    1    3    4
Row2    2    3    4
> X[1,3,1]
[1] 5
```

```
> as.vector(X>4)
 [1] FALSE FALSE FALSE FALSE  TRUE  TRUE
TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
TRUE  TRUE  TRUE  TRUE FALSE
[19] FALSE FALSE FALSE  TRUE  TRUE  TRUE
> which(as.vector(X>4))
 [1]  5  6  7  8  9 10 11 12 13 14 15 16 17 22 23 24
> as.vector(X)[which(as.vector(X>4))]
 [1]  5  6  7  8  9 10 11 12 36 37 38 39 40  5  6  7
```

The apply() function for arrays

Average matrix in an array

```
> a <- array(rnorm(2 * 2 * 10), c(2, 2, 10))
> apply(a, c(1, 2), mean)
          [,1]        [,2]
[1,] -0.2353245 -0.03980211
[2,] -0.3339748 0.04364908
```

The apply() function for arrays

```
> a[,,1]+a[,,2]+a[,,3]+a[,,4]
         [,1]     [,2]
[1,] 0.5026851 1.646615
[2,] 0.5753803 2.126716
> apply(a[,,1:4], c(1, 2), sum)
         [,1]     [,2]
[1,] 0.5026851 1.646615
[2,] 0.5753803 2.126716
> sum(a[,,1:4]) # sum is not the same
[1] 4.851396
```

The apply() function for arrays

```
> apply(a[,,1:4], 1, sum) # sums for all elements of each row
[1] 2.149300 2.702096
> apply(a[,,1:4], 2, sum) # sums for all elements of each column
[1] 1.078065 3.773331
> apply(a[,,1:4], 3, sum) # sums for all elements of each table
[1] 4.38399274 0.05762672 0.37768933 0.03208737
> a[,,1]
          [,1]       [,2]
[1,] 0.9356295  2.1267906
[2,] 1.5825914 -0.2610187
```

The apply() function for arrays

```
# this is not equivalent to apply
> a[,,sum(1:4)]
           [,1]       [,2]
[1,] -0.3119881  0.2235598
[2,]  1.6607486 -0.3690628
> sum(1:4)
[1] 10
> a[,,10]
           [,1]       [,2]
[1,] -0.3119881  0.2235598
[2,]  1.6607486 -0.3690628
```

More examples on arrays

```
> iris3[1:2,,]
, , Setosa
     Sepal L. Sepal W. Petal L. Petal W.
[1,]      5.1      3.5      1.4      0.2
[2,]      4.9      3.0      1.4      0.2

, , Versicolor
     Sepal L. Sepal W. Petal L. Petal W.
[1,]      7.0      3.2      4.7      1.4
[2,]      6.4      3.2      4.5      1.5

, , Virginica
     Sepal L. Sepal W. Petal L. Petal W.
[1,]      6.3      3.3      6.0      2.5
[2,]      5.8      2.7      5.1      1.9
```

More examples on arrays

```
> dim(iris3)
[1] 50  4  3
> attributes(iris3)
$dim
[1] 50  4  3

$dimnames
$dimnames[[1]]
NULL

$dimnames[[2]]
[1] "Sepal L." "Sepal W." "Petal L." "Petal W."

$dimnames[[3]]
[1] "Setosa"     "Versicolor" "Virginica“
```

```
> attributes(iris3)$dim
[1] 50  4  3
> dim(iris3)
[1] 50  4  3
> attributes(iris3)$dimnames
[[1]]
NULL
[[2]]
[1] "Sepal L." "Sepal W." "Petal L." "Petal W.“
[[3]]
[1] "Setosa"     "Versicolor" "Virginica"

> dimnames(iris3)
> dimnames(iris3)[[2]]
[1] "Sepal L." "Sepal W." "Petal L." "Petal W."
```

More examples on arrays

```
> a1<-array(1:24,dim=c(3,4,2))
> a1
, , 1

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

, , 2

     [,1] [,2] [,3] [,4]
[1,]   13   16   19   22
[2,]   14   17   20   23
[3,]   15   18   21   24
```

```
> dim(a1)<-c(4,3,2)
> a1
, , 1

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

, , 2

     [,1] [,2] [,3]
[1,]   13   17   21
[2,]   14   18   22
[3,]   15   19   23
[4,]   16   20   24
```

More examples on arrays

```
> # arrays and matrices
> is.matrix(a1)
[1] FALSE
> is.matrix(a1[,,1])
[1] TRUE
> is.array(a1)
[1] TRUE
> as.matrix(a1)
      [,1]
 [1,]    1
 [2,]    2
.............
[23,]   23
[24,]   24
```

```
> x<-1:24
> x
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
17 18 19 20 21 22 23 24
> dim(x)
NULL
> dim(x)<-c(3,4,2)
> x
, , 1
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12

, , 2
     [,1] [,2] [,3] [,4]
[1,]   13   16   19   22
[2,]   14   17   20   23
[3,]   15   18   21   24
```

More examples on arrays

```
> # arrays and matrices
> is.matrix(a1)
[1] FALSE
> is.matrix(a1[,,1])
[1] TRUE
> is.array(a1)
[1] TRUE
> as.matrix(a1)
      [,1]
 [1,]    1
 [2,]    2
.............
[23,]   23
[24,]   24
```

```
> x<-1:24
> x
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
17 18 19 20 21 22 23 24
> dim(x)
NULL
> dim(x)<-c(3,4,2)
> x
, , 1
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12

, , 2
     [,1] [,2] [,3] [,4]
[1,]   13   16   19   22
[2,]   14   17   20   23
[3,]   15   18   21   24
```