

Statistical Programming with R

Basics

Emmanuel Kemel (HEC Paris, CNRS)

Notions seen in these slides

- objects in R
- reading/saving/exporting files
- manipulating files

Characteristics

- Language and software
- free and open source
- mutli-platform (Windows, Linux, Mac)
- The official website is r-project.org
- Archives about R are stored on CRAN: Comprehensive R Archive Network
a network of servers around the world that store identical, up-to-date, versions of code and documentation for R

Why using R?

- A structured language
- Easier to get off the path: if a given command does not exist, write your own one
- Several ways to proceed: follow your own logic.
- One software for data management, statistics, regressions, graphics, editing

Installation

- The installation file is available at <https://cran.r-project.org/>
- Two new versions per year (Spring and Fall): no need to update systematically
- There is a 32bit and a 64bit version.

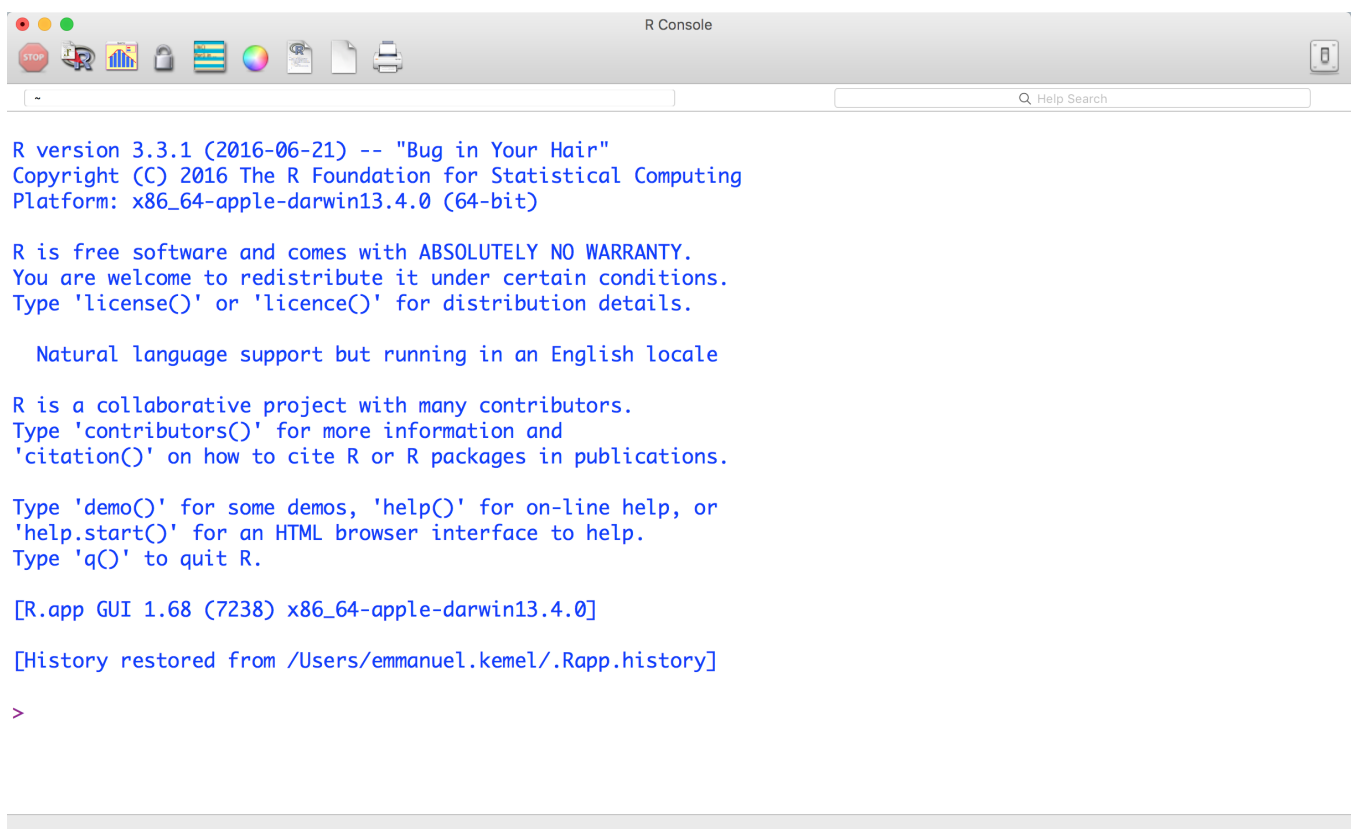
Packages

- R can be “extended” by user defined functions, grouped in packages
- [Link to available_packages_by_name](#)
- Packages
 - are available on CRAN
 - come with a normalized documentation
- Most popular packages are added as standard R functions in recent versions.

Resources

- R's official website contains several manuals, including An Introduction to R
- Books
 - Applied Econometrics with R
- Blogs and websites with tutorials, examples, discussions
 - Quick-R
 - R-bloggers
 - IDRE-UCLA
- A lot of posts on stackoverflow.com
- When you have a question, simply Google it ...

Very first contact



The screenshot shows the R Console window on a Mac. The title bar says "R Console". The menu bar includes "STOP", "R", "Help", "File", "Edit", "View", "Window", and "Help Search". The main text area displays the following information:

```
R version 3.3.1 (2016-06-21) -- "Bug in Your Hair"
Copyright (C) 2016 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin13.4.0 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

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.

[R.app GUI 1.68 (7238) x86_64-apple-darwin13.4.0]
[History restored from /Users/emmanuel.kemel/.Rapp.history]

>
```

Figure: Screen shot from R shell

Very first contact

The shell can be used directly as a calculator.

```
1+1
```

```
[1] 2
```

You can see that:

- expressions can be entered directly in the shell and are executed
- there is no need to load or define a data set

Help

Each core or package function comes with an help file

- `help(topic)` or `?topic` opens the page related to the topic
- `help.search("topic")` or `??topic` search for pages containing "topic"

Using scripts

It is (more than) highly recommended to use scripts, it

- keeps memory of past commands
- allows to re-run an entire analysis after modifications on the dataset
- allows to share code with your co-authors and (geeky) friends
- allows you to adapt scripts from one project to another

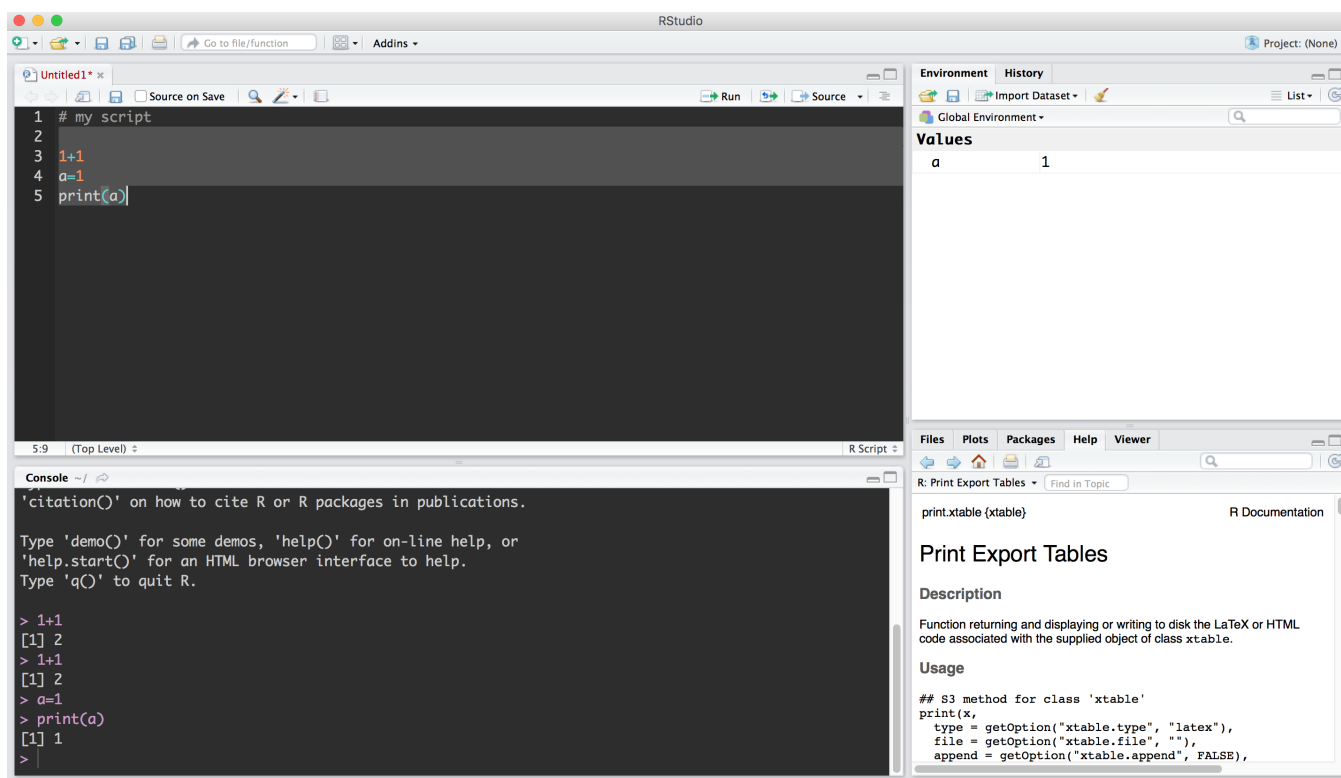
R studio

RStudio

Integrated environment for:

- writing scripts (syntax-highlight editor)
- run scripts
- have an overview on the environment (files, data in memory, graphics)
- edit documents

RStudio



Outline

- 1 Getting started with R
- 2 Objects in R
 - Modes
 - Variables and Memory
 - Classes of data objects
- 3 Data Management
 - Importing and exporting data
 - Managing data: example

Modes in R

- R (like Sata) the following modes of storing data
 - numeric, that can be integers or doubles
 - character for strings of characters
 - logical for logical values
- Other types can be used for (more) specific applications (e.g complex)
- The mode of an object can be accessed with function **mode()**
- Other modes do not deal with data, such as functions or expressions.
- Missing values are represented by **NA** (Not Available), whatever the mode.

Numeric values

- R contains several “built in” numeric values.

```
mode(1+1)
[1] "numeric"
mode(log(1))
[1] "numeric"
mode(pi)
[1] "numeric"
mode(NaN)
[1] "numeric"
mode(Inf)
[1] "numeric"
```


Examples of numeric values

Numeric values support

- operators: $+$, $-$, $*$, $/$, $^$ (power), `exp()`, `log()`, `%%` (modulo), ...
- functions that apply to numeric objects. (e.g `min()`, `max()`, `mean()`, `median()`)

```
1+Inf
[1] Inf
exp(-Inf)
[1] 0
exp(log(0))
[1] 0
pi%%2
[1] 1.141593
```

NA and NaN are absorbing

```
1+NA
[1] NA
sqrt(-1)+1
Warning in sqrt(-1):  production de NaN
[1] NaN
(-1)^(0.5)+1
[1] NaN
```

This can be annoying but this is for your own good!

Logical

Logical values

- take values **TRUE** (also denoted **T**, or **1**) or **FALSE** (also denoted **F**, or **0**)
- support logical operations: `==`, `>`, `>=`, `<=`, `&` (and), `|` (or), `!` (negation), `xor()`
- are returned by test functions. Ex: `is.na()`, `is.numeric()`, `is.character()`

```
1==2 | 2>0
```

```
[1] TRUE
```

```
(0<0.5) & (0.5<1)
```

```
[1] TRUE
```

```
log(pi)<Inf & !is.na(10^6)
```

```
[1] TRUE
```

```
is.numeric("Emmanuel") | mode("Emmanuel")==="numeric"
```

```
[1] FALSE
```

Characters

Characters are strings of characters

- a character string must contain quotes, either “ “ or ’ ’
- characters have operations of their own: `paste()`, `substr()`,

```
mode("Bonjour Emmanuel")
[1] "character"
mode("TRUE")
[1] "character"
substr("Bonjour Emmanuel",1,7)
[1] "Bonjour"
paste("Bonjour", "Emmanuel")
[1] "Bonjour Emmanuel"
toupper("e")
[1] "E"
sub("behavioral","behavioural","behavioral economics")
[1] "behavioural economics"
```

Conversions of data modes

- Manual conversion

Functions prefixed by “as.” allow to force the conversion of types, when possible

```
as.numeric(TRUE)
```

```
[1] 1
```

```
as.numeric("TRUE")
```

```
Warning: NAs introduits lors de la conversion automatique
```

```
[1] NA
```

```
1*"2"
```

```
Error in 1 * "2": argument non num'érique pour un opérateur binaire
```

```
1*as.numeric("2")
```

```
[1] 2
```

```
as.character(pi)
```

```
[1] "3.14159265358979"
```

Conversions of data modes (continued)

- Automatic conversion

```
1+ !is.na(pi)
[1] 2
paste("my favorite band is U",2)
[1] "my favorite band is U 2"
paste("my favorite movie is",pi)
[1] "my favorite movie is 3.14159265358979"
```

It is better to clearly define the mode of data when there is a risk of ambiguity, and to make conversions explicit.

Rules of conversion

convert to	function	rules	
numeric	as.numeric	FALSE→0	
		TRUE→1	
		"1","2"... →1,2...	
		"A","B"... →NA	
logic	as.logical	0 →FALSE	
		1,... →TRUE	
		"FALSE","F" →FALSE	
		"TRUE","T" →TRUE	
		other →NA	
character	as.character	1,2,... →"1", "2", ...	
		FALSE →"FLASE"	
		TRUE →"TRUE"	

Table: Defaults rules of conversion

Non-data modes

Other modes include:

- functions: they are character strings that operate when used with `()`
- commands that can be executed later
- formulas: that are used for running statistical methods

```
mode(sqrt)
[1] "function"
sqrt(pi)
[1] 1.772454
command=expression(sqrt(pi))
eval(command)
[1] 1.772454
eq="x~y+z"
mode(eq)
[1] "character"
Eq=as.formula(eq)
mode(Eq)
```


Outline

- 1 Getting started with R
- 2 Objects in R
 - Modes
 - Variables and Memory
 - Classes of data objects
- 3 Data Management
 - Importing and exporting data
 - Managing data: example

Variables: creation and assignment

It can be useful to store data in variables, whose content can be saved, displayed, modified, used for calculations.

Creation and assignment of a variable is made by the same operation

```
a<-TRUE;
mode(a)
[1] "logical"
length(a)
[1] 1
a # equivalent to print(a)
[1] TRUE
a=a+2
print(a); length(a)
[1] 3
[1] 1
```

Naming variables

Variable names

- cannot start with a number
- are case sensitive
- can replace the pre-defined variables
- commands `a=1` ; `a<-1` and `1->a` are equivalent

```
a=2; A=1
a==A
[1] FALSE
a.1<-10^30
pi=3
cos(2*pi)
[1] 0.9601703
sqrt=a
sqrt(a)
[1] 1.414214
```

Memory

In R, all objects are kept in memory.

- Objects currently in memory can be listed with function `ls()`
- Objects can be erased from memory using `rm()`
- the memory taken by an object can be obtained by `object.size()`

Examples

```
ls()
[1] "A"          "Eq"          "a"           "a.1"
[5] "command"    "eq"          "pi"          "scriptname"
[9] "sqrt"

a=2
print(a)
[1] 2

rm(a)
ls()
[1] "A"          "Eq"          "a.1"          "command"
[5] "eq"         "pi"          "scriptname"  "sqrt"

print(a)
Error in print(a):  objet 'a' introuvable
```

Outline

- 1 Getting started with R
- 2 Objects in R
 - Modes
 - Variables and Memory
 - Classes of data objects
- 3 Data Management
 - Importing and exporting data
 - Managing data: example

Classes

Whatever their mode, data can be stored in objects of different classes.

The class characterizes:

- the way data are organized and accessed
- the action of generic functions on the object (ex: **summary()**)

A class is characterized by a series of attributes that describe the properties of the data.

By default, every object in R is a vector

Other main classes have richer structures (i.e. more attributes) than vectors:

- factors
- matrixes
- arrays
- data.frames
- lists

The empty object NULL is of class NULL.

Vectors are the default data objects in R

- There is no scalar, only vectors of length 1.

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

R reads this as a sum of two vectors of length 1 whose elements are summed and returns a vector of length 1, with value 3.

- The function for building vectors is `c()`
- Vectors are characterized by their `length()`

```
c(1,2,3)  
[1] 1 2 3  
a=c(1,2,3)  
print(a)  
[1] 1 2 3  
length(a)  
[1] 3  
c(a,4)  
[1] 1 2 3 4
```


Functions for generating vectors

Several function can be useful to generate specific vectors

- sequences: operator : , seq()
- repetition of vectors
- random numbers : rnorm()

```
1:10
[1] 1 2 3 4 5 6 7 8 9 10
(-1):10
[1] -1 0 1 2 3 4 5 6 7 8 9 10
-(1:10)
[1] -1 -2 -3 -4 -5 -6 -7 -8 -9 -10
seq(0,1,0.1)
[1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
rep(1:5,2)
[1] 1 2 3 4 5 1 2 3 4 5
rep(1:5, each=2)
[1] 1 1 2 2 3 3 4 4 5 5
```

The power of vectors

The arguments of vector-generating functions can be... vectors

```
rep(1:5, times=2:6)
```

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

```
rnorm(n=10, mean=10, sd=2)
```

```
[1]  9.129136  9.098704 13.405812 14.396450  9.624422
```

```
[6]  8.292922 10.051256 11.871937  8.416862 10.723474
```

```
rnorm(n=10, mean=seq(10,100,10), sd=2)
```

```
[1] 12.40038 16.10754 29.58395 38.99577 52.03266
```

```
[6] 58.73090 70.52123 76.64917 90.38045 101.33532
```

R recycles vectors

- In R, operations involving vectors deal with each element separately: in theory, the vectors involved in a operation should have the same length.
- However, for matter of flexibility, R will recycle (i.e replicate) the shorter vector(s) in order to adapt it to the size of the longest one.

```
1:4+1
```

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

```
1:4+1:2
```

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

```
1:4+1:3
```

```
Warning in 1:4 + 1:3: la taille d'un objet plus long n'est pas multiple de  
la taille d'un objet plus court
```

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

```
1:10<4
```

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

```
[10] FALSE
```

```
1:10<c(2,8)
```

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

```
[10] FALSE
```

Indexes of vectors

- Values of a vector can be accepted through their index, using operator `[]`
- Vectors of indexes can be used... to refer to the indexes of a vector.

```
a=seq(0,1,0.2)
a[1]
[1] 0
a[2:5]
[1] 0.2 0.4 0.6 0.8
a[c(1,3,5)]
[1] 0.0 0.4 0.8
a[-c(1,3,5)]
[1] 0.2 0.6 1.0
a[rep(2,5)]
[1] 0.2 0.2 0.2 0.2 0.2
a[a[6]]
[1] 0
a[a[4]]
numeric(0)
```

Subsetting vectors

- It is possible to select specific values of a vector by selecting the relevant indexes

```
a=1:10
a[c(T,F,T,T,F)]
[1] 1 3 4 6 8 9

a>5
[1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE
[10]  TRUE

a[a>5]
[1] 6 7 8 9 10

a[2]=NA
a[!is.na(a)]
[1] 1 3 4 5 6 7 8 9 10
```

Exercise

Consider that the results of a race take the form of two vectors

- the gender to the participant (1 for female)
- the time of the participant

For each participant i , $gender[i]$ is its gender and $time[i]$ its time.

- What is the gender of the winner?

HINT: the min of a vector x is given by $\min(x)$

```
gender=c(1,0,1,0,0,0,1,1,0,0)
time=c(20,14,19,21,12,15,16,9,22,10)
```

Solution

```
gender[time==min(time)]  
[1] 1  
# or equivalently  
gender[which.min(time)]  
[1] 1
```

This illustrates that

- there can be several ways to code
- more fancy functions can simplify the code but are not necessary

Sorting vectors

There are two main functions that can be used for sorting vectors or their indexes

- `sort()` sorts the vector
- `order()` returns the indexes of the sorted vector

Example: getting back to the previous example, and assume that index i refers to the number of the participant

- give the list of participant numbers, sorted according to arrival order
- give the sorted arrival times
- what is the time of the time of the participant ranked 5? (Hint, there exists a function **`rank()`**)

Response

```
order(time)
[1]  8 10  5  2  6  7  3  1  4  9
sort(time)
[1]  9 10 12 14 15 16 19 20 21 22
time[order(time)]
[1]  9 10 12 14 15 16 19 20 21 22
time[rank(time)==5]
[1] 15
sort(time)[5]
[1] 15
```

Naming vectors

It is possible to assign names to the indexes of vectors, with function **names()**

Values can then be accessed through their names.

```
grades=c(12,14,16)
names(grades)=c("Pierre","Paul","Jacques")
grades[c(1,3)]
  Pierre Jacques
    12      16
grades[c("Pierre","Jacques")]
  Pierre Jacques
    12      16
grades[c(1,"Jacques")]
  <NA> Jacques
    NA      16
```

Attributes

- When giving names to a vector, we “enrich” its structure.
- In R, this kind of added elements are the attributes of the object.
- They can be accessed with function **attributes()**

```
grades=c(12,14,16)
attributes(grades)
NULL
object.size(grades)
80 bytes
names(grades)=c("Pierre","Paul","Jacques")
attributes(grades)
$names
[1] "Pierre" "Paul"   "Jacques"
object.size(grades)
440 bytes
```

We will now see other data objects, that have richer structures than vectors, i.e. more attributes.

Factors

- Factors are vectors whose values are labelled with a level
- The level is the attribute that differentiates factors from standard vectors.
- They are well suited for discrete (or categorical) variables, whether ordered or not
- Examples:
 - gender
 - grades ranging from A to F
 - Time (year/months) or geographic units (zip code)
- Factors are characterized by their length, and their levels.

Examples

```
gender=c(1,2,1,1,2)
summary(gender)

  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  1.0    1.0    1.0    1.4    2.0    2.0

gender=factor(gender, labels=c("Female","Male"))
summary(gender)

Female    Male
      3      2

levels(gender)

[1] "Female" "Male"

levels(gender)=c("Women","Men","Unknown")
summary(gender)

  Women    Men Unknown
      3      2      0

attributes(gender)

$levels
[1] "Women"  "Men"    "Unknown"

$class
[1] "factor"

summary(gender[drop=T])

Women    Men
      3      2
```

Note that the levels is different from the name. The former labels the values, the other labels the indexes.

Matrixes

- Matrixes are vectors whose values are spread over 2 dimensions: rows and columns
 - They are defined by the command **matrix()**
 - They supports classical operation, that are run cell by cell
 - They have their own operators: **t()**, **det()**, **solve()**
- Matrixes are characterized by their mode and their dimensions.

Building matrices

The most explicit way to define a matrix is to use function **matrix()**

```
v=1:10
m1=matrix(v, ncol=5)
print(m1)
      [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10

m2=matrix(v, ncol=5, byrow=T)
print(m2)
      [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    6    7    8    9   10

m3=diag(1:3)
print(m3)
      [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    2    0
[3,]    0    0    3
```

Operations over matrixes

- Usual operations are run cell by cell
- here again, R recycles

```
matrix(1:10,ncol=5)+1
```

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

```
matrix(1:10,ncol=5)+1:2
```

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

Matrix operations

- Matrix algebra is included by default in R

```
m=matrix(1:4,ncol=2)
det(m)
[1] -2
t(m)
      [,1] [,2]
[1,]    1    2
[2,]    3    4
diag(rep(1,2))%*%m
      [,1] [,2]
[1,]    1    3
[2,]    2    4
m%*%1:2
      [,1]
[1,]    7
[2,]   10
solve(m)%*%m
      [,1] [,2]
[1,]    1    0
[2,]    0    1
```

Combining matrixes

Matrixes can be combined:

- by addition of rows, with function **rbind()**
- by addition of columns, with function **cbind()**

```
m1=matrix(1:4,ncol=2)
m2=matrix(1:6,ncol=2)
m3=matrix(1:6,ncol=3)
m1_m2=rbind(m1, m2)
dim(m1_m2)

[1] 5 2

m1_m3=cbind(m1, m3)
dim(m1_m3)

[1] 2 5
```

Taking subparts of matrixes

Values of a matrix can be obtained using the indexes of the matrix. When possible, R returns the simplest object class, unless `drop=F`

```
m=matrix(1:9,ncol=3)
```

```
m[1,1]
```

```
[1] 1
```

```
m[,2]
```

```
[1] 4 5 6
```

```
m[,2, drop=F]
```

```
 [,1]
```

```
[1,] 4
```

```
[2,] 5
```

```
[3,] 6
```

```
m[c(2,3),]
```

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

```
[1,] 2 5 8
```

```
[2,] 3 6 9
```

Exercise

- create a matrix results, that contains the gender and the time of participants
- give the gender of the winner
- order the matrix by ranks of participants

Result

```
result=cbind(gender, time)
result[result[,2]==min(result[,2]),1]
```

```
gender
      1
```

```
result[order(result[,2]),]
```

	gender	time
[1,]	1	9
[2,]	2	10
[3,]	2	12
[4,]	2	14
[5,]	1	15
[6,]	2	16
[7,]	1	19
[8,]	1	20
[9,]	1	21
[10,]	1	22

Column and row names

- As for vectors, it is possible to add names to the dimension of a matrix.
- These are either `colnames` or `rownames`
- Then, lines and columns can be referred to by their names.

```
m=matrix(1:9,ncol=3)
colnames(m)=c("var1","var2","var3")
rownames(m)=c("obs1","obs2","obs3")
m["obs2",c(1,2)]
var1 var2
  2     5

m["obs2",c("var1","var2")]
var1 var2
  2     5

#attributes(m)
```

Limitation of vectors and matrixes

- Vectors and matrix only accept ONE mode of data at the same time.
- If several modes are stored in a same vector conversions will be forced.

```
c(1,TRUE, "Test")
[1] "1"      "TRUE" "Test"

m=matrix(1:10,ncol=5)
m[1,2]=NA
print(m)
      [,1] [,2] [,3] [,4] [,5]
[1,]    1  NA    5    7    9
[2,]    2    4    6    8   10

m[1,2]="NA"
print(m)
      [,1] [,2] [,3] [,4] [,5]
[1,] "1"  "NA" "5"  "7"  "9"
[2,] "2"  "4"  "6"  "8" "10"
```

Lists

- Lists are the most flexible objects in R
- Each element can contain a vector, a factor, a matrix, or even a list
- Elements can have different modes
- Elements of a list can be accessed with operator `$` or equivalently operator `[[]]`
- operator `[]` returns a list that contains the selected elements.

Examples

```
year=2020
rooms=c("T010","T012","T010")
students=cbind(names=c("Pierre","Paul","Jacques"),
course=list(year,rooms,students)
length(course)
[1] 3
course$year
NULL
course$participants[2,2]
NULL
```

Data Frames

- A data frame is a list of vectors that have the same length, but not necessarily the same mode.
- Because of these two characteristics, they can be treated like matrixes or like lists
- They are the standard way to store data sets in R

Examples

```
scores=data.frame(names=c("Pierre","Paul","Jacques"),
age=c(20,21,19),grade=c("A","C","C"), gender=1)
scores[1,3] # because scores has a matrix shape
[1] A
Levels: A C

scores[1,"grade"] # because columns have names
[1] A
Levels: A C

scores[1,]$grade # because scores is a list
[1] A
Levels: A C

scores$grade[1] # because each component of the list is a vector
[1] A
Levels: A C
```

Outline

- 1 Getting started with R
- 2 Objects in R
 - Modes
 - Variables and Memory
 - Classes of data objects
- 3 Data Management
 - Importing and exporting data
 - Managing data: example

Specifying the working directory

The current working directory can be

- accessed by **getwd()**
- specified by **setwd()**

The later take a string of characters in arguments.

Beware: for R, “\” means “return”. Therefore, under Windows, paths should use either / or \\.

The content of a directory can be listed with function **list.files()** that returns a vectors with the names of the files in the current (or specified) folder (and possibly subfolders)

Importing files

- Comma-separated files (.csv) can be imported with **read.table()**
Main options included:
 - header
 - sep that specifies the separator
 - dec that specifies which symbol is used for separating decimals
- Fixed width formatted data are read with **read.fwf()**
 - the width of each variable must be specified
- R data files (.rda) are simply loaded in memory with **load()**
- More exotic files can be read/written with specific packages
 - XLConnect for Excel files and their sheets
 - foreign for Stata (.dat) and SPSS files.

Recommendations

After loading a dataset

- 1 check the dimensions
- 2 check the variable names
- 3 check that decimals are not coded as characters

Exercise

- Each month the Bureau of Labor Statistics in the U.S. Department of Labor conducts the “Current Population Survey” (CPS), which provides data on labor force characteristics of the population, including the level of education, unemployment, and earnings.
- CPS08 contains data on hourly wage of worked, as well as their gender and education level, from respondents sampled in March 2008.
- Code:
 - 1 for female
 - 1 for bachelor degree

Solution

```
setwd("~/Dropbox/Rprogramming/data")
data=read.table("CPS08.csv",header=T, sep=";", dec=",")
dim(data)
[1] 7711    5
```

Exporting datasets

- Datasets can be exported to ASCII files (.txt or .csv) with **write.table()**
Note that by default, **write.table()** adds rownames, which will create a column with line numbers when rownames is NULL
- Specific packages must be used to export in other formats.
- Note that when R creates a file, former version will be replaced.

Saving R objects

The easiest way to save R objects is to use `save()`

The function takes in arguments:

- the name of the objects that are to be saved
- the name of the .rda file that will be written on the disc

It is good practice to give the same name to the object that is saved and the .rda file.

If you do not remember the name of the object that was saved: check the memory or use `get()`

```
save(data, file="CPS08.rda")
rm(data)
load("CPS08.rda")
names(CPS08)
Error in eval(expr, envir, enclos): objet 'CPS08' introuvable
names(data)
[1] "ahe"      "year"      "bachelor" "female"    "age"
CPS08=get(load("CPS08.rda"))
names(CPS08)
[1] "ahe"      "year"      "bachelor" "female"    "age"
```

Outline

- 1 Getting started with R
- 2 Objects in R
 - Modes
 - Variables and Memory
 - Classes of data objects
- 3 Data Management**
 - Importing and exporting data
 - Managing data: example**

For the remainder of the session

- We will apply the methods that have been described to load and analyze a given dataset
- There is no new notions coming up, but several functions that simplify coding

```
setwd("~/Dropbox/Rprogramming/data")  
data=read.table("titanic.csv", header=T, sep=";")
```

First look at the data

After loading a `data.frame`, it is good practice to

- ① look at its dimensions
- ② If relevant, look at variable names
- ③ If relevant, run a summary with function `summary()`
- ④ If you need to see in order to believe, look at the dataset in the data editor (in RStudio, or with `fix()`)
- ⑤ Look at the first 3 rows

Example

```
dim(data)
[1] 799    5

names(data)
[1] "Name"          "PClass"          "Age"
[4] "Sex"           "Survived..1.yes."

#summary(data)
#fix(data)
data[1:3,]
```

	Name	PClass	Age	Sex
1	Pavlovic, Mr Stefo	3rd	NA	male
2	Lefebvre, Miss Mathilde	3rd	NA	female
3	Nieminen, Miss Manta Josefina	3rd	NA	female

	Survived..1.yes.
1	0
2	0
3	0

Focusing on a subsample

In order to select a subsample of the data set

- use a logical conditions on lines
- use function `subset()` that takes as arguments the name of the data.frame and the condition

For missing variables,

- **`na.omit()`** remove all the lines for which at least one of the variables have a missing value.
- **`complete.cases()`** returns the indexes of the rows without NAs.

Function **`unique()`** removes identical lines.

Example

Create separated files for

- passengers whose age is known
- survivors
- survivors from the first class

Example

```
data1=data[!is.na(data$Age),]  
data1=subset(data, !is.na(Age) )  
data2=data[data[,5]==1,]  
names(data)[5]="Survived"  
data3=subset(data, Survived==1 & PClass=="1st" )
```

Focusing on some variables

In order to select a subset of variables, use a matrix attributes of the data.frame and select/remove the relevant indexes of columns

- by their number
- by their name

```
#removing names  
data=data[, -1]  
data$Name=NULL
```

Manipulation of variables

- For creating a new variable, use operator **\$** as for a list
`data$new=1`
- The same method can be used to change the value of a variable
`data$Adult=data$Age>=18`
Note that in this case, the old value of the variable is erased
- For conditional assignments, a useful function is **if.else()** can be useful.

Examples

```
data$Adult=data$Age>=21
summary(data$Adult)
  Mode  FALSE  TRUE  NA's
logical    95  356   348

data$Age_type=factor(ifelse(data$Age>=21,"Adult","Underage"))
summary(data$Age_type)
  Adult Underage  NA's
   356     95    348
```

A useful function: cut()

- **cut()** can be used to transform a continuous variable into a discrete one

```
data$Age_type2=cut(data$Age,breaks=c(8,16,21,35,60))
summary(data$Age_type2)
```

(8,16]	(16,21]	(21,35]	(35,60]	NA's
20	66	188	141	384

Referring to a data set

Using operator `$` for each variable can make expression lengthy
It is possible to use functions that indicate which data.frame is used, to that variables can be used without prefix

- `attach(data)` loads all the variables the data.set in memory
- `with(data, expression)` can be used for a given expression
- `transform(data, var=expression)` can be used for transforming a variable
- for selecting/removing observations/variables, function `subset(data, conditions_on_lines, select=c(colnames))` can be useful

Combining data sets

Consider two data.frames data1 and data2.

- If data1 and data2 have the same variables and different observations, they can be stacked with **rbind()**
Note that the names of the two data.frames should perfectly match
- If data1 and data2 have the same (ordered) observations and different variables, they can be “juxtaposed” with **cbind()**
- Otherwise, the two datasets should be merged, according to one or several key i.e. variables that are common in the two data sets and allow for matching.
This is done with function **merge()**

Example with merge

```
load("titanic_incomplete1.rda"); load("titanic_incomplete2.rda")

merge1=merge(titanic_incomplete1,titanic_incomplete2)
dim(merge1)
[1] 102  5

merge2=merge(titanic_incomplete1, titanic_incomplete2, all.x=TRUE)
dim(merge2)
[1] 500  5

merge3=merge(titanic_incomplete1,titanic_incomplete2, all.x=TRUE, all.y=TRUE)
dim(merge3)
[1] 798  5
```