BIG DATA

Diploma de Especialización en Big Data

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Introduction to R

José Hernández Orallo-María José Ramirez - Cèsar Ferri Ramírez
cferri@disc.upv.es
D235 DSIC

R Enviroment

- Open source language for data analysis
  - Object oriented (although we can use functional or procedural features as well)
  - Interpreted and multi-platform
  - Powerful and efficient matricial language
  - Incremental and extensible
  - Command-line interface (although there exist some IDEs)

R Enviroment

- When running R we find the command interpreter:
  - Use it as a simple calculator: *2+3*
  - Define variables/objects
  - Create functions
  - Execute previously produced code

Object Creation

- An object can be created in different ways:

```
>1->a”
>a<-1
>a=1
>assign(“a”,1)
```

Object Creation

- Every object has two intrinsic attributes: type and length.
- *typeof* refers to the class of the elements in the object .
- The length is the number of elements in the object.

```
>typeof(a)
[1] "double"
> length(a)
[1] 1
```

- Although this is a simplification of a complex and long story.

Special Values

- Infinite: *Inf*
  - We can operate normally with them
- Undefined (not a number) values: *NaN*
  - Denotes a numeric that is not a number: 0/0
- Missing values: *NA*
  - (Not available), is independent of the data type
- Empty values: *NULL*
  - Denotes an empty object, is skipped (i.e., removed) for vectors,

Special Values

- We can check the value of an object using functions:

```
> t < - c (T ,F , NA , FALSE )
> is . na ( t )
[1] FALSE FALSE TRUE FALSE
> is . nan ( t )
[1] FALSE FALSE FALSE FALSE
> is . nan (0/0)
[1] TRUE
> is . null (0/0)
[1] FALSE
> is . infinite (0/0)
[1] FALSE
> x < - c (1 , 2 , NA , 4 , NA , 5)
> bad <- is . na ( x )
> x [! bad ]
[1] 1 2 4 5
```

Special Values

- *complete.cases* is useful to avoid problematic registers

```
> x <- c (1 , 2 , NA , 4 , NA , 5)
> y <- c (" a " , NA , NA ," d " ," e " ," f ")
> good <- complete . cases (x , y )
> good
[1] TRUE FALSE FALSE TRUE FALSE TRUE
> x [ good ]
[1] 1 4 5
> y [ good ]
[1] " a " " d " " f "
> mydf <- data . frame (x , y )
> good2 <- complete . cases ( mydf )
> good2
[1] TRUE FALSE FALSE TRUE FALSE TRUE
```

Vectors

- The simplest object in R is the vector.
  - Even when we write x<-3 we are creating a vector
- Vectors are used to store values of the same atomic datatype (character, logical, numeric and complex).
- Vectors are created using the function *c( )*:

```
> v < - c (" AVS " , " CDA " , " DIM " , " TVD ")
> v
[1] " AVS " " CDA " " DIM " " TVD "
> length ( v )
[1] 4
```

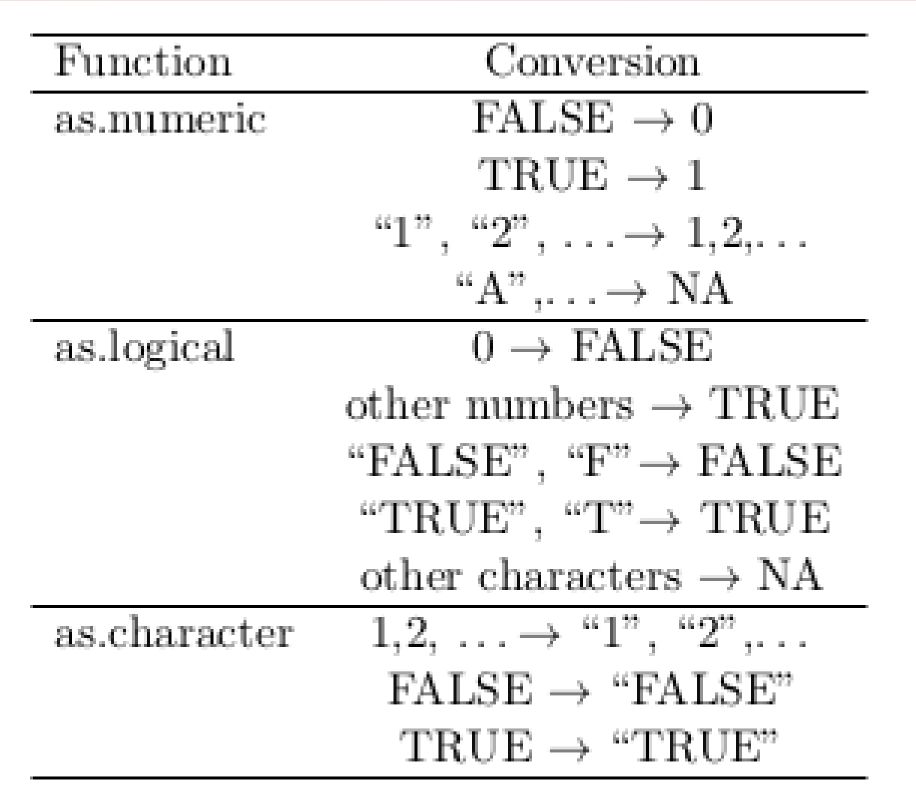
Coercion

- If the types are different, coercion (i.e., conversion) takes place, if possible.

```
> u < - c (1 ,2 ,4 ,6.0)
 >u
[1] 1 2 4 6
 >w < - c (1.3 ,2.5 ,3.9 ,5)
 >w
[1] 1.3 2.5 3.9 5.0
```

Coercion

- If the types are different, coercion (i.e., conversion) takes place, if possible.



Function	Conversion
as.numeric	FALSE → 0
	TRUE → 1
	“1”, “2”, . . . → 1,2,. . .
	“A”,. . . → NA
as.logical	0 → FALSE
	other numbers → TRUE
	“FALSE”, “F” → FALSE
	“TRUE”, “T” → TRUE
	other characters → NA
as.character	1,2, . . . → “1”, “2”,. . .
	FALSE → “FALSE”
	TRUE → “TRUE”

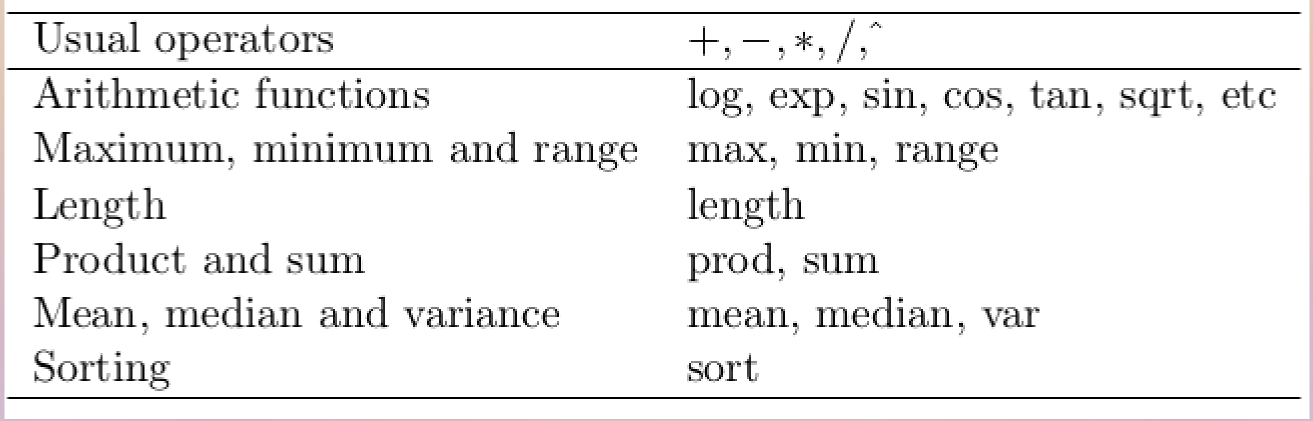
Coercion

- If the types are different, coercion (i.e., conversion) takes place, if possible.

```
>u < - c (1 ,2 ,3 ,4)
>class ( u )
[1] " numeric "
 >as.numeric ( u )
[1] 1 2 3 4
 >as.logical ( u )
[1] TRUE TRUE TRUE TRUE
 >as.character ( u )
[1] "1" "2" "3" "4"
>as.numeric ( x )
[1] NA NA NA
```

Vectors

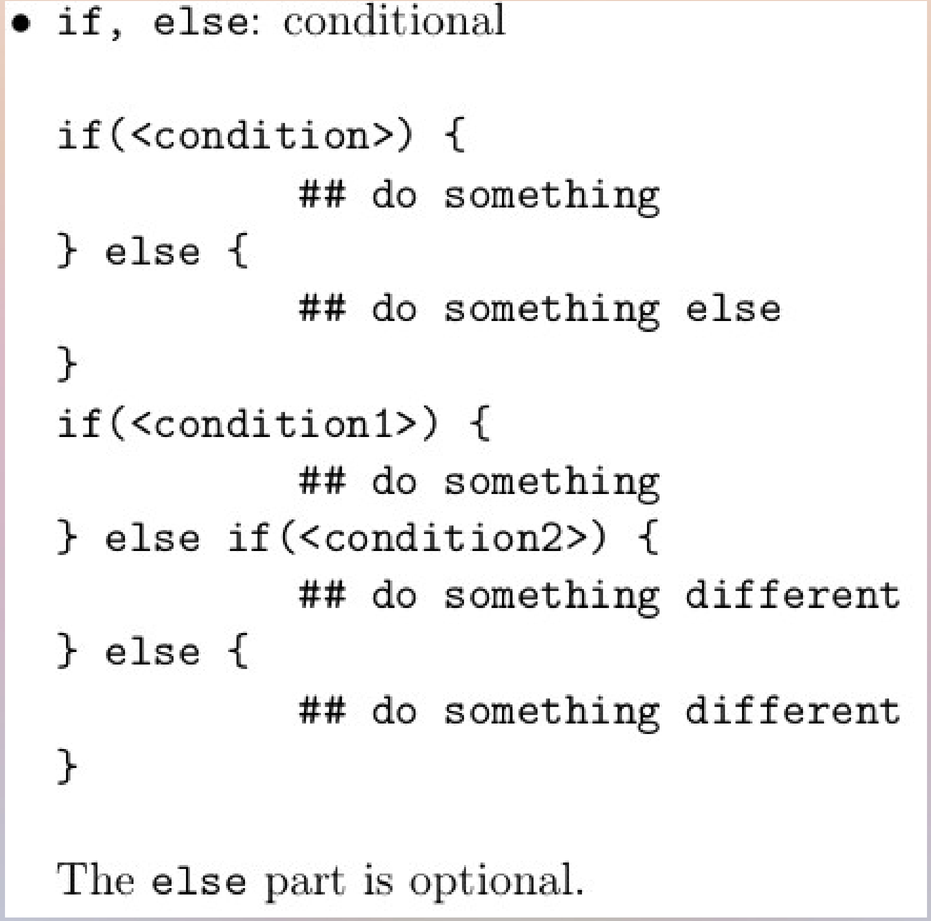
- R features several functions that work with vectors.



Usual operators	$+, -, *, /, \hat{}$
Arithmetic functions	log, exp, sin, cos, tan, sqrt, etc
Maximum, minimum and range	max, min, range
Length	length
Product and sum	prod, sum
Mean, median and variance	mean, median, var
Sorting	sort

Control Structures

- R use use curly brackets {} to group expressions



- if, else: conditional

```
if(<condition>) {
          ## do something
} else {
          ## do something else
}
if(<condition1>) {
          ## do something
} else if(<condition2>) {
          ## do something different
} else {
          ## do something different
}
```

The else part is optional.

Control Structures

- *for*: executes a loop a given number of times. It uses a variable (the iterator) and assigns values successively using a sequence or vector.
- *while*: executes a loop while the condition is true
- *repeat*: executes a loop infinitely. It can only be exited with break.
- *break*: exits a loop.
- *next*: goes to the next iteration of a loop.
- *return*: returns from a function and whatever loops inside the function we may be in.

Control Structures

```
> for ( i in 1:10) cat (i , sep ="\ n ")
> count <- 0

> while ( count < 10) {
+ print ( count )
+ count <- count + 1
+ }
> count < -0

> repeat {
+count < - count +1
+if ( count %%2==0){
+next
+} else {
+print ( count )
+}
+
if ( count >=9) { break }
+
}
```

Factors

- Factor is a vector that is used to specify discrete values over the elements
  - nominal and ordinal factors
- Levels are the possible values they can take
  - They are stored as numerical codes

Factors

```
> pet < - c (" cat " ," dog " ," dog " ," cat " ," cat " ," snake" ,
 " parrot " ," cat ")
> pet
[1]" cat "" dog " " dog " " cat "" cat " " snake " " parrot " " cat "
> Fpet < - factor ( pet )
> Fpet
[1] cat dog dog cat cat snake parrot cat
Levels : cat dog parrot snake
> levels ( Fpet )
[1] " cat "
" dog "
" parrot " " snake "
> mode ( Fpet )
[1] " numeric "
```

Sequences

- R provides several functions to generate sequences

```
> x < -1:15
> x
[1] 1 2 3 4
> y < -5:1
> y
[1] 5 4 3 2 1
```

Sequences

- *Rep* and *seq* are useful functions for sequences

```
> seq (1 , 5 , 0.5)
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
>seq ( length =9 , from =1 , to =5)
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
> rep (5 ,10)
[1] 5 5 5 5 5 5 5 5 5 5
> rep ( c (1 ,2) ,4)
[1] 1 2 1 2 1 2 1 2
```

Sequences

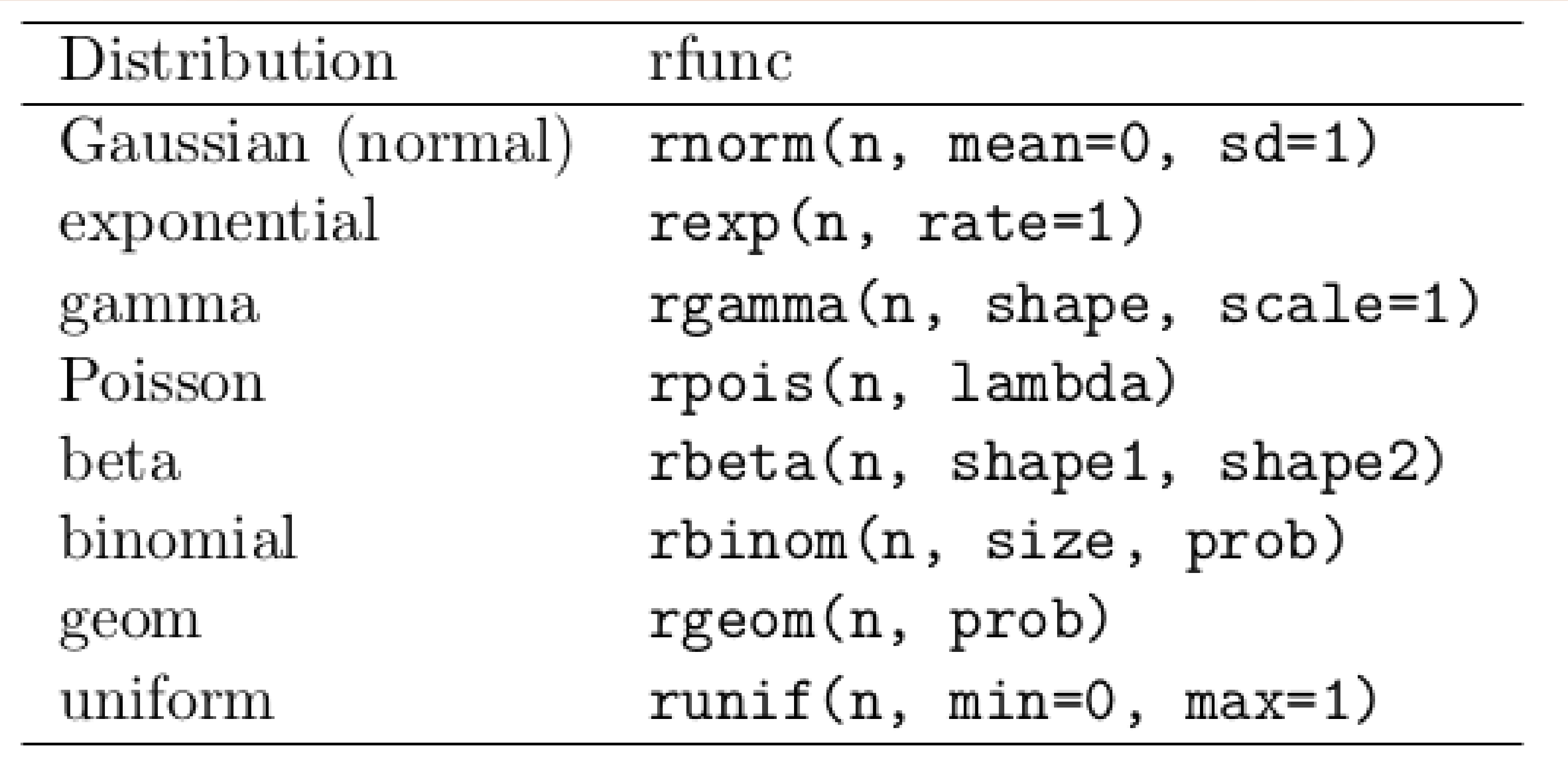
- *gl(k,n)* generates a sequence where *k* is the number of levels of the factor and *n* is the number of repetitions of each level:

```
>gl (2 ,4 , labels = c (" boy " ," girl "))
[1] boy boy boy boy girl girl girl girl
Levels : boy girl
```

Random Sequences

- *R* provides several functions with the general form *Xfunc(n, p1, p2,...)*
  - *dfunc*: probability density
  - *pfunc*: cumulative probability
  - *qfunc*: quartile value
  - *rfunc*: random generation of values
  - *n*: number of data to be generated
  - *p1*, *p2*, ... are the values for the paramaters of the distribution.

Random Sequences



Distribution	rfunc
Gaussian (normal)	rnorm(n, mean=0, sd=1)
exponential	rexp(n, rate=1)
gamma	rgamma(n, shape, scale=1)
Poisson	rpois(n, lambda)
beta	rbeta(n, shape1, shape2)
binomial	rbinom(n, size, prob)
geom	rgeom(n, prob)
uniform	runif(n, min=0, max=1)

```
> runif (5)
[1] 0.9111758 0.4921438 0.3238787 0.4402546 0.1792024
> rnorm (10 , mean =5 , sd =1)
[1] 4.481318 4.319482 4.105529 7.498646 5.668317
4.046477 6.629298 4.627495 6.276711 4.497717
```

Matrices and arrays

- A matrix is defined as a vector with an additional attribute (*dim*)
  - It is actually a numeric vector of length 2 that defines the number of rows and columns
  - *byrow*, *bycoulmn* set the order to fill the matrix

```
> matrix (1:6 , 2 , 3 , byrow = TRUE )
      [ ,1] [ ,2] [ ,3]
[1 ,]   1     2    3
[2 ,]   4     5    6
```

Matrices and arrays

- Operations with matrices:
  - *rbind*: join matrices by rows
  - *cbind*: join matrices by columns
  - *T*: transpose a matrix

Matrices and arrays

- Given the notation for indices as *[i,j]*, we can specify both, one or none

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

Lists

- Lists are a special kind of data type that can contain a series of elements of different classes.
  - They can be indexed by names:

```
> animal <- list ( order = ’ carnivore ’ , family = ’ feline ’ , species = ’ lynx ’)
> animal
$ order
[1] " carnivore "
$ family
[1] " feline "
$ species
[1] " lynx "
> animal [[1]]
[1] " carnivore "
animal $ order
[1] " carnivore "
```

Lists

- We can add elements dynamically to a list using names.
- Lists can be without names and composed of various kinds of elements:

```
> l <- list ()
> l $ a <- 3
> l $ b <- 5
> l $ c <- 8
> l
$a
[1] 3
$b
[1] 5
$c
[1] 8
> mylist <- list (3 , ’a ’)
> mylist [[1]]
[1] 3
> mylist [[2]]
[1] " a "
> mylist [[3]] <- 8.5
```

Lists

- If the classes of all elements are compatible (or coerceable) we can convert a list to a vector with the function *unlist*:

```
> animal $ class < - c ( ’ vertebrate ’ , ’ mammal ’)
> animal
$ order
[1] " carnivore "
$ family
[1] " feline "
$ name
[1] " lynx "
$ class
[1] " vertebrate " " mammal "
> unlist ( animal )
Order        family species class1    class2
"carnivore"  feline " “linx”  class2 " “mammal "
```

Avoiding loops

- We can avoid loops with high order functions
  - Very efficient
- *Apply* variations: *lapply*, *sapply*, *mapply*..
- *sapply(x, fun, ...)*, where *x* is a list (or a vector), *fun* is the name of the function and . . . other arguments

```
> myvector <- c (3.5 , 7.8 , 4.2 , 2.5)
> sapply ( myvector , round )
[1] 4 8 4 2
```

- *lapply(x, fun, …)* always returns a list.

Avoiding loops

- We can count how many elements there are in a vector meeting a condition in various ways:

```
> myvector
[1] 3.5 7.8 4.2 2.5
> myvector + 1
[1] 4.5 8.8 5.2 3.5
> myvector < 5
[1] TRUE FALSE TRUE
TRUE
> sum ( myvector < 5)
[1] 3
> myvector [ myvector < 5]
[1] 3.5 4.2 2.5
```

- *which* returns the indices of the elements meeting a condition

```
> which ( myvector < 5)
[1] 1 3 4
```

Avoiding loops

- What is the result of this operation?

```
> myvec <- myvector
> myvec [ which ( myvec < 5)] <- myvec [ myvec < 5] + 1
```

- It is equivalent to:

```
>ifelse ( myvec < 5 , myvec +1 , myvec )
```

Data frames

- Data structure to store and work with data tables

```
d < - data . frame ( name = c ( ’ Anne ’ , ’ Joe ’ , ’ Mario ’ , ’ Rose ’ , ’ Mary ’) ,
age = c (21 ,34 ,54 ,27 ,41))
> d
name age
1 Anne 21
2Joe 34
3 Mario 54
4 Rose 27
5 Mary 41
```

Data frames

- We can add columns to data frames:

```
> city < - c ( ’ Valencia ’ , ’ Barcelona ’ , ’ Madrid ’ , ’ Valencia ’ , ’ Valencia ’)
> job < - c ( ’ student ’ , ’ dealer ’ , ’ engineer ’ , ’ physician ’ , ’ journalist ’)
> d2 < - cbind ( city , job , d )
```

- We can access data in different ways:

```
> d2 $ city
[1] Valencia Barcelona Madrid Valencia Valencia
Levels : Barcelona Madrid Valencia
```

- R has automatically converted the array to factors

Data frames

- Different ways of filtering data:

```
>d2 [ d2 $ city == ’ Valencia ’ , c (" name " ," age ")]
>d2 [ d2 $ age >35 ,]
>subset ( d2 , city == ’ Valencia ’ , job : age )
```

- To know the size of the data frame:

```
> nrow ( d2 )
[1] 5
> ncol ( d2 )
[1] 4
>dim(d2)
[1] 5 4
```

Names and Levels

- With names, we can change the names of elements of arrays, matrices, lists...

```
> x <- 1:3
> names ( x )
NULL
> names ( x ) <- c (" first " , " second " , " third ")
> x
first second third
1     2     3
> names ( x )
[1] " first " " second " " third "
```

Sorting data

- The function *sort()* is able to order numbers or alphanumeric characters (ASCII)
- Other related functions :
  - *order*(): returns the indices of the sorted vector.
  - *duplicated*(): returns a logical vector with the values that are duplicated.
  - *unique*(): returns the values that are different.

Functions

- Functions in R can be defined in the following way:
  - *function(<arguments>) {## body}.*
- Functions are objects and they can be passed as argument to other functions (higer-order functions)

```
> myfun <- function (x , y )
+ { if (x > y ) return ( - x ) else return ( y )}
> myfun (5 ,3)
[1] -5
> myfun
function (x , y ) { if (x > y ) return ( - x ) else return ( y )}
```

Functions

- We can define parameters by default
- Parameters are passed by value

```
> myfun <- function (x , y=2 )
+ {
+ x<-1
+ return (x+y)}
> z<-3
> myfun(z)
[1] 3
> z
[1] 3
```

Functions

- The ... is used in R to denote any variable number of arguments.
- Paste is a function that allows the user to concatenate any number of strings, or even combine vectors of string:

```
> paste ( c ( ’a ’ , ’b ’ , ’c ’) , c (1 ,2) , sep =":")
[1] " a :1" " b :2" " c :1"
>
> paste ( c ( ’a ’ , ’b ’ , ’c ’) , c (1 ,2))
[1] " a 1" " b 2" " c 1"
```

Home work

- Packages
- R Graphics