A quick introduction to

Hands-on training workshop on seasonal forecasting: Data access, bias correction and downscaling - Santander (Spain)
8-12 Sep 2014

Contents

1 Introduction

During this practice we will undertake some operations related with the handling, analysis and visualization of climate data in the `R` environment. The first part is envisaged as a quick start to the R environment, showing the main types of data structures and some basic operations related to them.

In the second part, we will load some climate data structures and will perform some basic data handling and visualization examples on them.

1.1 Main R features

`R` is a software package similar to other programs (i.e. MATLAB, Octave, Python), specially developed for the statistical analysis.

It is an integrated environment: it has been developed as a whole entity and not as a collection of tools. It includes:

- An efficient system for data storage and manipulation
- A collection of tools to manage arrays
- Integrated tools for data analysis
- Screen graphs and portable format graphs generation
- A simple and effective programming language (with scripting capabilities)

It is free software: Both in the sense of *free* beer, and in the sense of *free* speech. With free software, you are granted[1]:

- The freedom to run the program, for any purpose (freedom 0).
- The freedom to study how the program works, and adapt it to your needs (freedom 1).Access to the source code is a precondition for this.
- The freedom to redistribute copies so you can help your neighbour (freedom 2).
- The freedom to improve the program, and release your improvements to the public, so that the whole community benefits (freedom 3). Access to the source code is a precondition for this.

Available for different platforms : (source code and pre-compiled binaries): UNIX, MacOS, Windows

The `R` system is divided into 2 conceptual parts:

1. The "base" `R` system that you download from CRAN[2]
2. Everything else ...
   - There are about 4000 packages on CRAN that have been developed by users and programmers around the world.
   - There are also many packages associated with the Bioconductor project[3].

[1] Free Software Foundation `http://www.fsf.org`
[2] The Comprehensive R Archive Network `http://cran.r-project.org/`
[3] `http://bioconductor.org`

- 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

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`.
- There are also "Recommend" packages: `boot`, `class`, `cluster`, `codetools`, `foreign`, `KernSmooth`, `lattice`, `mgcv`, `nlme`, `rpart`, `survival`, `MASS`, `spatial`, `nnet`, `Matrix`.

Note: Before starting to work with `R`, it is advised to create a new dedicated directory where all the work should be included. In fact, if several projects are to be developed at the same time, every project should have its own directory.

1.2 Getting help

Once `R` is installed, there is a comprehensive built-in help system. At the program's command prompt you can use any of the following:

`help.start()` general help

`help(foo)` help about function foo

`?foo` same thing

`apropos("foo")` list all functions containing string foo

`example(foo)` show an example of function foo

`RSiteSearch("foo")` searches for help manuals and archived mailing lists

`vignette()` show available vingettes. Vignettes are optional supplementary documentation available in some packages

`vignette("foo")` show specific vignette

1.3 Sample Datasets

`R` comes with a number of sample datasets that you can experiment with. Type `data()` to see the available datasets. The results will depend on which packages are loaded. Type `help(datasetname)` for details on a sample dataset.

2 Data structures in R

`R` is an object-oriented language: an object in R is anything (constants, data structures, functions, graphs) that can be assigned to a variable:

Data Objects: used to store real or complex numerical values, logical values or characters. These objects are always vectors.

Language Objects: functions, expressions

The most basic object is a vector. Empty vectors can be created with the `vector()` function. A vector can only contain objects of the same class ...

> **but**: The one exception are lists, which is represented as a vector but can contain objects of different classes (in fact, that is usually why we use them).

Vectors: one-dimensional arrays used to store collection data of the same mode. `R` has five basic or **atomic** classes of objects:

1. Numeric vectors (mode: numeric, real numbers)
2. Integer vectors (mode: integer, integers)
3. Character vectors (mode: character, text strings)
4. Logical vectors (model: logical, `TRUE/FALSE`)
5. Complex vectors (mode: complex)

Matrices: two-dimensional arrays to store collections of data of the same mode. They are accessed by two integer indices.

Arrays: similar to matrices but they can be multi-dimensional (more than two dimensions)

Factors: vectors of categorical variables designed to group the components of another vector with the same size

Lists: ordered collection of objects, where the elements can be of different types

Data Frames: generalization of matrices where different columns can store data of different modes (what we would usually call a table of data).

Functions: objects created by the user and reused to make specific operations.

In addition, `R` objects can have *attributes*. The most typical ones are:

- `names`
- `dimnames`
- `dimensions` (e.g. matrices, arrays)
- `class`
- `length`
- Any other user-defined attributes/metadata ...

Attributes of an object can be accessed using the `attributes()` function.

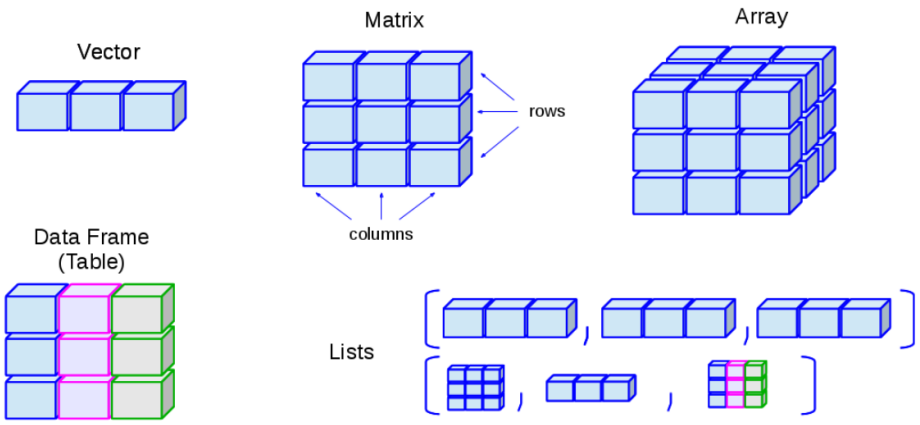

Figure 1: Schematic representation of the different data structures in R. Note that different atomic classes are differentiated by their color.

2.1 Vectors

2.1.1 Numeric vectors

There are several ways to assign values to a variable. For instance, the following assigns to object `x` a value of 3.14. `x` is a vector of length 1:

```
x <- 3.14
3.14 -> x
x = 3.14
assign("x", 3.14)
```

To show the values on screen:

```
x

## [1] 3.14

print(x)

## [1] 3.14
```

Objects are also printed on screen when created if the whole statement is put between parenthesis:

```
(x <- 3.14)

## [1] 3.14
```

Warning: R is case-sensitive:

```r
x <- 1
X <- 6
x + X

## [1] 7
```

2.1.2 Integer vector

```r
x = 1
class(x) # Not an integer

## [1] "numeric"

x = 1L # An integer
x = as.integer(1) # Same as above line
class(x)

## [1] "integer"
```

2.1.3 Logical Vectors

```r
a <- seq(1:10) # generate a sequence
a

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

class(a)

## [1] "integer"

b <- (a > 5)
b

##  [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE
## [10]  TRUE

class(b)

## [1] "logical"

a[b] # show values that fulfill the condition

## [1]  6  7  8  9 10

a[a > 5] # the same, but avoiding intermediate variable

## [1]  6  7  8  9 10
```

Test for exact equality:

```
identical(c(1,4,2,6), as.integer(c(1,4,2,6)))

## [1] FALSE

identical(c(1L,4L,2L,6L), as.integer(c(1,4,2,6)))

## [1] TRUE
```

2.1.4 Character Vectors

```
a <- "This is an example" # generate a character vector
a # show vector content

## [1] "This is an example"
```

We can concatenate vectors after converting them into character vectors, using `paste`:

```
x <- 1.5
y <- -2.7
paste("Point is (", x, ", ", y, ")", sep = "")

## [1] "Point is (1.5, -2.7)"
```

Some useful character vectors (type `?Constants`):

```
month.name

##  [1] "January"   "February"  "March"     "April"
##  [5] "May"       "June"      "July"      "August"
##  [9] "September" "October"   "November"  "December"

month.abb[6:8] # Summer months

## [1] "Jun" "Jul" "Aug"

letters[seq(1, length(letters), 2)]

##  [1] "a" "c" "e" "g" "i" "k" "m" "o" "q" "s" "u" "w" "y"
```

2.1.5 Typical operations to create vectors

Next, we show examples of typical operations with numeric vectors:

To generate a vector with several numeric values, using the concatenate command `c`:

```
a <- c(10, 11, 15, 19)
a

## [1] 10 11 15 19
```

The operations are always done over all the elements of the numeric array:

```
a * a # evaluate the square value of every element in the vector

## [1] 100 121 225 361

1 / a # evaluate the inverse value of every element in the vector

## [1] 0.10000 0.09091 0.06667 0.05263

b <- a-1 # subtract 1 from every element and assign the result to b
b

## [1]  9 10 14 18
```

To generate numeric sequences:

```
2:10

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

5:1

## [1] 5 4 3 2 1

seq(from = 1, to = 10, by = 3) # generate a sequence

## [1]  1  4  7 10

seq(1, 10, 3) # (parameters names can be avoided if order is kept

## [1]  1  4  7 10

seq(length = 10, from = 1, by = 3) # generate a fixed length sequence

##  [1]  1  4  7 10 13 16 19 22 25 28

# For more details and further examples:
help(seq)
```

To generate repetitions, use the command **rep**:

```
a <- 1:3; b <- rep(a, times = 3); c <- rep(a, each = 3)
```

Note that in the above example we have run three commands in the same line. They have been separated by a ";", equivalent to a line break (... also

note that this makes the code less readable). The content of the three variables defined is:

```
a

## [1] 1 2 3

b

## [1] 1 2 3 1 2 3 1 2 3

c

## [1] 1 1 1 2 2 2 3 3 3
```

If we need to know which are the objects that are currently defined in our workspace, we can list them:

```
ls()

## [1] "a" "b" "c" "x" "X" "y"
```

Undesired objects can be deleted using the `rm` function:

```
rm(a,c)
ls()

## [1] "b" "x" "X" "y"
```

To remove all objects loaded:

```
rm(list = ls())
ls()

## character(0)
```

2.1.6 Special numbers

`Inf` represents infinity.

```
x <- 2.8
x / 0

## [1] Inf

x / Inf

## [1] 0
```

The value `NaN` represents an undefined value (*not a number*)

```r
0/0
```

```
## [1] NaN
```

```r
Inf/Inf
```

```
## [1] NaN
```

> `NaN` can also be thought of as a missing value, but in R there is a reserved word for missing (*not available*) values: `NA`

```r
is.na(c(1, 5.53, NA))
```

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

2.1.7 Mixing modes: Coercion

What about the following?

```r
c(1.7, "a") # character
```

```
## [1] "1.7" "a"
```

```r
c(TRUE, 2) # numeric
```

```
## [1] 1 2
```

```r
c("a", TRUE) # character
```

```
## [1] "a"    "TRUE"
```

When different objects are mixed in a vector, *coercion* occurs so that every element in the vector is of the same class:

> **Explicit Coercion** Objects can be explicitly coerced from one class to another using the `as.*` functions, if available.

```r
x <- 0:6
class(x)
```

```
## [1] "integer"
```

```r
as.numeric(x)
```

```
## [1] 0 1 2 3 4 5 6

as.logical(x)

## [1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE

as.character(x)

## [1] "0" "1" "2" "3" "4" "5" "6"

as.complex(x)

## [1] 0+0i 1+0i 2+0i 3+0i 4+0i 5+0i 6+0i
```

Nonsensical coercion results in missing (`NA`) values (possibly with a warning):

```
x <- c("a", "b", "c")
as.numeric(x)

## Warning:  NAs introduced by coercion

## [1] NA NA NA

as.logical(x)

## [1] NA NA NA
```

2.2 Matrices

A matrix is a bi-dimensional collection of data:

```
# Define a matrix with 2 rows and 3 columns:
mdat <- matrix(c(1, 2, 3, 11, 12, 13), nrow = 2, ncol = 3)
mdat

##      [,1] [,2] [,3]
## [1,]    1    3   12
## [2,]    2   11   13

# The same matrix, but altering the order it is filled:
mdat2 <- matrix(c(1, 2, 3, 11, 12, 13), nrow = 2, ncol = 3, byrow = TRUE)
mdat2

##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]   11   12   13

# Matrix dimensions:
dim(mdat)

## [1] 2 3
```

```r
# Number of columns:
ncol(mdat)

## [1] 3

# Number of rows:
nrow(mdat)

## [1] 2
```

The elements of vectors and matrices are *recycled* when it is required by the involved dimensions:

```r
x <- c(1,1,1)
y <- c(x,7,7,7,x)
y

## [1] 1 1 1 7 7 7 1 1 1
```

Matrices can also be created directly from vectors by adding a dimension attribute:

```r
m <- 1:10
m

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

dim(m) <- c(2, 5)
m

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

Binding by rows and columns: Matrices can be created by column-binding or row-binding with `cbind()` and rbind().

```r
x <- 1:3
y <- 10:12
cbind(x, y)

##      x  y
## [1,] 1 10
## [2,] 2 11
## [3,] 3 12

rbind(x, y)

##   [,1] [,2] [,3]
## x    1    2    3
## y   10   11   12
```

Operating with vectors of different lengths. The longer object length must be a multiple of shorter object length:

```
length(x)

## [1] 3

length(y)

## [1] 3

x + y + 0.5

## [1] 11.5 13.5 15.5
```

In the above example, vector `x` is used 3 times, vector `y` is used only once and the scalar (one-element vector) 0.5 is used 9 times.

Recycling with matrices:

```
(a <- matrix(1:8, nrow = 4, ncol = 4))

##      [,1] [,2] [,3] [,4]
## [1,]    1    5    1    5
## [2,]    2    6    2    6
## [3,]    3    7    3    7
## [4,]    4    8    4    8
```

If no argument `data` is given, an empty matrix is created:

```
matrix(ncol = 3, nrow = 4)

##      [,1] [,2] [,3]
## [1,]   NA   NA   NA
## [2,]   NA   NA   NA
## [3,]   NA   NA   NA
## [4,]   NA   NA   NA
```

2.3 Arrays

Arrays are like matrices, but they can have one, two or more dimensions.

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

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

```
dim(arr)
```

```
## [1] 2 3 4
```

```
length(arr)
```

```
## [1] 24
```

2.4 Factors

Factors are vectors that contain categorical information useful to group the values of other vectors of the same size. One can think of a factor as an integer vector where each integer has a label. Factors are treated specially by modelling functions like `lm()` and `glm()`. Using factors with labels is better than using integers because factors are self-describing. For instance, having a variable that has values "Male" and "Female" is better than a variable that has values 1 and 2.

Let's see an example:

```
month.name # A vector containing the months of the year
```

```
##  [1] "January"   "February"  "March"     "April"
##  [5] "May"       "June"      "July"      "August"
##  [9] "September" "October"   "November"  "December"
```

```
class(month.name) # A character, not a vector
```

```
## [1] "character"
```

```
(month.factor <- factor(month.name))
```

```
##  [1] January   February  March     April     May
```

```
##  [6] June      July      August    September October
## [11] November  December
## 12 Levels: April August December February January ... September

levels(month.factor)

##  [1] "April"     "August"    "December"  "February"
##  [5] "January"   "July"      "June"      "March"
##  [9] "May"       "November"  "October"   "September"
```

The factor created follows the alphabetical order by default, but in this case this is not desirable. Factors can be ordered or unordered, In this case, we want it to be ordered. We can specify the levels explicitly, and its order will be preserved

```
(month.factor2 <- factor(rep(month.name, each = 3), levels = month.name))

##  [1] January   January   January   February  February
##  [6] February  March     March     March     April
## [11] April     April     May       May       May
## [16] June      June      June      July      July
## [21] July      August    August    August    September
## [26] September September October   October   October
## [31] November  November  November  December  December
## [36] December
## 12 Levels: January February March April May June ... December

 # The order is OK, but the factor is not yet formally ordered
is.ordered(month.factor2)

## [1] FALSE

factor.ordered <- ordered(month.factor2)
is.ordered(factor.ordered)

## [1] TRUE
```

We could use this additional information to perform an statistical analysis segregating the data according to these types. This will be covered lately in the Functions section.

2.5 Lists

Lists are ordered collections of objects, where the elements can be of a different type (a list can be a combination of matrices, vectors, other lists, ...) They are created using the `list()` function. In this example, suppose we have different data from a meteorological station:

```
station <- list(name="Parayas", alt=2, lonlat=c(-3.0,43.1), GSN = TRUE)
station
```

```
## $name
## [1] "Parayas"
##
## $alt
## [1] 2
##
## $lonlat
## [1] -3.0 43.1
##
## $GSN
## [1] TRUE
```

The different elements of the list can be accessed using the `$` symbol.

> **Tip**:Pressing the TAB key after the `$` shows the names of `station`

```
names(station) # return element names

## [1] "name"   "alt"    "lonlat" "GSN"

length(station) # check how many elements has 'station'

## [1] 4

str(station) # Provide a summary of the structure of 'station'

## List of 4
##  $ name  : chr "Parayas"
##  $ alt   : num 2
##  $ lonlat: num [1:2] -3 43.1
##  $ GSN   : logi TRUE

# Retrieve the data contained in each element
station$name

## [1] "Parayas"

station$altitude

## NULL

station$lonlat

## [1] -3.0 43.1

station$ECAnetwork

## NULL
```

New elements can be added in a simple way, just defining them:

```
station$GSN_id <- "SP00000841"
names(station)

## [1] "name"   "alt"    "lonlat" "GSN"    "GSN_id"
```

Lists can be concatenated to generate bigger lists:

```
station2 <- list(name="Espinama", alt=1014, lonlat=c(-3.1,42.9), GSN=FALSE)
stations.list <- list(station, station2)
length(stations.list)

## [1] 2
```

As the elements in a list can be R objects of any kind, lists are extremely versatile.

2.6 Data frames

A Data Frame is an special type of list very useful for the statistical work. There are some restrictions to guarantee that they can be used for this statistical purpose. Among other restrictions, a Data Frame must verify that:

1. List components must be vectors (numeric, character or logical vectors), factors, numeric matrices or other data frames.
2. Vectors, which are the variables in the data frame, must be of the same length.

Basically, in a Data Frame all the information is displayed as a table where the columns have the same number of rows and can contain objects of different modes (numeric, character, ...). Data Frames can be created using the `data.frame()` function. Lets see how to define a data frame with two elements, a numeric vector and a character vector (note that both must be same length vectors):

```
df <- data.frame(numbers=c(10,20,30,40),text=c("a","b","c","a"), stringsAsFactors=FALSE)
str(df)

## 'data.frame': 4 obs. of  2 variables:
##  $ numbers: num  10 20 30 40
##  $ text   : chr  "a" "b" "c" "a"

df$text # character vector not converted to a factor

## [1] "a" "b" "c" "a"
```

In a data frame, character vectors are automatically converted into factors by default, and the number of levels is determined as the number of unique different values in such a vector. Note that thi default behaviour has been modified in the example above by setting the argument `stringsAsFactors = FALSE`. Compare the above example with the following:

```
df2 <- data.frame(numbers=c(10,20,30,40),text=c("a","b","c","a"))
df2$text

## [1] a b c a
## Levels: a b c
```

Characteristics of the Data Frame structure:

```
mode(df) # storage mode of the object

## [1] "list"

typeof(df) # (R internal) storage mode of the object

## [1] "list"

class(df) # Object class

## [1] "data.frame"
```

2.7 Functions

Functions are created using the `function()` directive and are stored as `R` objects just like anything else. In particular, they are `R` objects of class "function".

```
# NOTE: pseudocode follows
f <- function(<arguments>) {
      ## Do something interesting
}
```

Functions in `R` are "first class objects", which means that they can be treated much like any other `R` object. Importantly:

- functions can be passed as arguments to other functions.
- functions can also be nested, so that you can define a function inside of another function.

The return value of a function is the last expression in the function body to be evaluated.

Functions have named arguments which potentially have default values:

- The *formal arguments* are the arguments included in the function definition.
- The `formals` function returns a list of all the formal arguments of a function
- Not every function call in `R` makes use of all the formal arguments: unction arguments can be missing or might have default values

3 Basic operations with objects

3.1 Subsetting

There are a number of operators that can be used to extract subsets of R objects:

[always returns an object of the same class as the original; can be used to select more than one element (there is one exception, see sec. 3.1.2)

[[is used to extract elements of a list or a data frame; it can only be used to extract a single element and the class of the returned object will not necessarily be a list or data frame

$ is used to extract elements of a list or data frame by name; semantics are similar to that of [[.

3.1.1 Subsetting vectors

```
x <- c("a", "b", "c", "c", "d", "a")
x[1] # First element in vector x

## [1] "a"

x[2] # Second

## [1] "b"

x[1:4] # First four elements

## [1] "a" "b" "c" "c"

x[x > "a"] # Elements fulfilling condition

## [1] "b" "c" "c" "d"

u <- x > "a"
u

## [1] FALSE  TRUE  TRUE  TRUE  TRUE FALSE

x[u]

## [1] "b" "c" "c" "d"
```

3.1.2 Subsetting matrices

Matrices can be subsetted in the usual way with (i, j) type indices (i for rows, j for columns)

```
x <- matrix(1:6, 2, 3)
x[1, 2]
```

```
## [1] 3

x[2, 1]

## [1] 2
```

If an index is missing, it means "all" of that dimension:

```
x[1, ]

## [1] 1 3 5

x[, 2]

## [1] 3 4
```

By default, when a single element of a matrix is retrieved, it is returned as a vector of length 1 rather than a 1×1 matrix. This behavior can be turned off by setting `drop = FALSE`.

```
x <- matrix(1:6, 2, 3)
x[1, 2]

## [1] 3

x[1, 2, drop = FALSE]

##      [,1]
## [1,]    3
```

Similarly, subsetting a single column or a single row will return a vector, not a matrix (by default):

```
x <- matrix(1:6, 2, 3)
x[1, ]

## [1] 1 3 5

x[1, , drop = FALSE]

##      [,1] [,2] [,3]
## [1,]    1    3    5
```

3.1.3 Subsetting lists

```
x <- list(foo = 1:4, bar = 0.6)
x[1]

## $foo
## [1] 1 2 3 4
```

```
x[[1]]

## [1] 1 2 3 4

x$bar

## [1] 0.6

x[["bar"]]

## [1] 0.6

x["bar"]

## $bar
## [1] 0.6
```

```
x <- list(foo = 1:4, bar = 0.6, baz = "hello")
x[c(1, 3)]


## $foo
## [1] 1 2 3 4
##
## $baz
## [1] "hello"
```

Extracting multiple elements of a list: The [[operator can be used with computed indices; $ can only be used with literal names.

```
x <- list(foo = 1:4, bar = 0.6, baz = "hello")
name <- "foo"
x[[name]] # computed index for foo

## [1] 1 2 3 4

x$name # element 'name' doesn't exist!

## NULL

x$foo

## [1] 1 2 3 4
```

Subsetting Nested Elements of a List

The [[can take an integer sequence:

```r
x <- list(a = list(10, 12, 14), b = c(3.14, 2.81))
x[[c(1, 3)]]

## [1] 14

x[[1]][[3]]

## [1] 14

x[[c(2, 1)]]

## [1] 3.14
```

3.1.4 Partial matching of names

Partial matching of names is allowed with [[and $:

```r
x <- list(tenochtitlan = 1:5)
x[["t"]]

## NULL

x[["t", exact = FALSE]]

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

3.2 Handling missing values

A common task is to remove missing values (NAs):

```r
x <- c(1, 2, NA, 4, NA, 5)
bad <- is.na(x)
x[!bad]

## [1] 1 2 4 5

na.omit(x) # Another strategy for the same problem

## [1] 1 2 4 5
## attr(,"na.action")
## [1] 3 5
## attr(,"class")
## [1] "omit"

na.exclude(x) # Yet another one

## [1] 1 2 4 5
## attr(,"na.action")
## [1] 3 5
## attr(,"class")
## [1] "exclude"
```

What if there are multiple things and you want to take the subset with no missing values?

```
x <- c(1, 2, NA, 4, NA, 5)
y <- c("a", "b", NA, "d", NA, "f")
good <- complete.cases(x, y)
good

## [1]  TRUE  TRUE FALSE  TRUE FALSE  TRUE

x[good]

## [1] 1 2 4 5

y[good]

## [1] "a" "b" "d" "f"
```

Data frames with missing values:

```
airquality[1:6, ] # This is a built-in dataset in R base

##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
## 5    NA      NA 14.3   56     5   5
## 6    28      NA 14.9   66     5   6

good <- complete.cases(airquality)
airquality[good, ][1:6, ]

##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
## 7    23     299  8.6   65     5   7
## 8    19      99 13.8   59     5   8
```

3.3 Vectorized operations

Many operations in `R` are vectorized making code more efficient, concise, and easier to read.

```
x <- 1:4; y <- 6:9
x + y

## [1]  7  9 11 13
```

```
x > 2

## [1] FALSE FALSE  TRUE  TRUE

x >= 2

## [1] FALSE  TRUE  TRUE  TRUE

y == 8

## [1] FALSE FALSE  TRUE FALSE

x * y

## [1]  6 14 24 36

x / y

## [1] 0.1667 0.2857 0.3750 0.4444
```

3.3.1 Vectorized matrix operations

```
x <- matrix(1:4, 2, 2); y <- matrix(rep(10, 4), 2, 2)
x * y ## element-wise multiplication

##      [,1] [,2]
## [1,]   10   30
## [2,]   20   40

x / y

##      [,1] [,2]
## [1,]  0.1  0.3
## [2,]  0.2  0.4

x %*% y ## true matrix multiplication

##      [,1] [,2]
## [1,]   40   40
## [2,]   60   60
```

3.4 Handling dates and times

R has developed a special representation of dates and times.

- Dates are represented by the `Date` class
- Times are represented by the `POSIXct` or the `POSIXlt` classes
- Dates are stored internally as the number of days since 1970-01-01

- Times are stored internally as the number of seconds since 1970-01-01

Dates are represented by the `Date` class and can be coerced from a character string using the `as.Date()` function:

```
x <- as.Date("1970-01-01")
x

## [1] "1970-01-01"
```

Alternatively, any character representation of dates and/or times can be converted to the `POSIXct` or the `POSIXlt` using `strptime()`:

```
# Any 'sui-generis' character representations of dates can be handled:
x <- c("1jan1960", "2jan1960", "31mar1960", "30jul1960")
x

## [1] "1jan1960"  "2jan1960"  "31mar1960" "30jul1960"

z <- strptime(x, "%d%b%Y")
z # by default, it assigns the locale time zone, but this can be changed

## [1] "1960-01-01 CET" "1960-01-02 CET" "1960-03-31 CET"
## [4] "1960-07-30 CET"

z.UTC <- strptime(x, "%d%b%Y", tz = "UTC")
z.UTC

## [1] "1960-01-01 UTC" "1960-01-02 UTC" "1960-03-31 UTC"
## [4] "1960-07-30 UTC"
```

It is difficult to remember all the different formatting strings used when dealing with times and dates, but fortunately the documentation (`help(strptime)`) is very complete.

As a rule of thumb, `POSIXct` is best for storing the time information, while `POSIXlt` is very powerful for working with time information. Conversion between both types is straightforward:

```
ti <- Sys.time()
class(ti)

## [1] "POSIXct" "POSIXt"

ti

## [1] "2014-09-07 13:56:13 CEST"

ti2 <- as.POSIXlt(ti)
class(ti2)

## [1] "POSIXlt" "POSIXt"

ti2$year # starts in 1900
```

```
## [1] 114

ti2$year + 1900 # 'readable' year

## [1] 2014

ti2$mon # Caution: months start in 0=January (not in 1)

## [1] 8

ti2$mon + 1 # month in 1-12 format

## [1] 9

ti2$yday # day of the year

## [1] 249

ti2$wday # day of the week

## [1] 0

ti2$isdst # Daylight saving time flag

## [1] 1

# And many more (see help(DateTimeClasses))
```

- `POSIXct` is just a very large integer under the hood; it is a useful class when you want to store times in something like a data frame
- `POSIXlt` is a list underneath and it stores a bunch of other useful information like the day of the week, day of the year, month, day of the month and many more ....

There are also a number of generic functions that work on dates and times, for instance:

`weekdays`: give the day of the week

`months`: give the month name

`quarters`: give the quarter number

Operations with times and dates

Operations with times even keep track of leap years, leap seconds, daylight savings, and time zones. Time intervals are represented by the class `difftime`:

```
# Differences of days
x <- as.Date("2012-03-01")
y <- as.Date("2012-02-28")
class(x - y)
```

```
## [1] "difftime"

x - y

## Time difference of 2 days

# Hours (Difference of local time and UTC)
x <- as.POSIXct("2012-10-25 06:00:00")
y <- as.POSIXct("2012-10-25 06:00:00", tz = "GMT")
y - x

## Time difference of 2 hours

# Milliseconds
a <- Sys.time()
b <- Sys.time()
b-a

## Time difference of 0.001161 secs
```

4 Control structures

Control structures in `R` allow to control the flow of execution of the program, depending on runtime conditions. Common structures are:

if, else : testing a condition

for : execute a loop a fixed number of times

while : execute a loop while a condition is true

repeat : execute an infinite loop

break : break the execution of a loop

next : skip an iteration of a loop

return : exit a function

Most control structures are not used in interactive sessions, but rather when writing functions or longer expressions.

4.1 `if-else`

if/else statements have the same syntax:

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

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

This is a valid if/else structure:

```
if(x > 3) {
      y <- 10
} else {
      y <- 0
}
```

So is this one:

```
y <-  if(x > 3) {
      10
} else {
      0
}
```

Of course, note that the `else` clause is not necessary:

```
if(<condition1>) {

}
if(<condition2>) {

}
```

4.2 `for`

`for` loops take an iterator variable and assign it successive values from a sequence or vector. `for` loops are most commonly used for iterating over the elements of an object (list, vector, ...).

```
for(i in 1:10) {
      print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
```

This loop takes the i variable and in each iteration of the loop gives its values 1, 2, 3,..., 10, and then exits. The following three loops have the same behavior:

```
x <- c("a", "b", "c", "d")

for(i in 1:4) {
      print(x[i])
}
```

```
## [1] "a"
## [1] "b"
## [1] "c"
## [1] "d"
```

```
for(i in seq_along(x)) {
      print(x[i])
}
```

```
## [1] "a"
## [1] "b"
```

```
## [1] "c"
## [1] "d"

for(letter in x) {
      print(letter)
}

## [1] "a"
## [1] "b"
## [1] "c"
## [1] "d"

for(i in 1:4) print(x[i])

## [1] "a"
## [1] "b"
## [1] "c"
## [1] "d"
```

`for` loops can be nested:

```
x <- matrix(1:6, 2, 3)
for(i in seq_len(nrow(x))) {
      for(j in seq_len(ncol(x))) {
            print(x[i, j])
      }
}

## [1] 1
## [1] 3
## [1] 5
## [1] 2
## [1] 4
## [1] 6
```

Be careful with nesting though. Nesting beyond 2-3 levels is often very difficult to read/understand, and very often there is a more efficient way of doing it, as we shall see later.

4.3 `while`

`while` loops begin by testing a condition. If it is true, then they execute the loop body. Once the loop body is executed, the condition is tested again, and so forth.

```
# This loop starts in 0.
count <- 0
while(count < 10) {
      print(count)
      count <- count + 1 # In each iteration sums 1 to the previous value
}
```

```
## [1] 0
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9

# It ends when 10 is reached at the 10th iteration
```

While loops can potentially result in infinite loops if not written properly. Use with care!. Sometimes there will be more than one condition in the test:

```
z <- 5
while(z >= 3 && z <= 10) {
      print(z)
      coin <- rbinom(1, 1, 0.5)
      if(coin == 1) { ## random walk
            z <- z + 1
      } else {
            z <- z - 1
      }
}
```

```
## [1] 5
## [1] 4
## [1] 5
## [1] 6
## [1] 5
## [1] 6
## [1] 5
## [1] 4
## [1] 5
## [1] 6
## [1] 5
## [1] 4
## [1] 5
## [1] 4
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 6
## [1] 5
## [1] 6
## [1] 5
```

```
## [1] 6
## [1] 7
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 8
## [1] 9
## [1] 8
## [1] 7
## [1] 6
## [1] 5
## [1] 4
## [1] 3
```

Conditions are always evaluated from left to right.

4.4 repeat-break

`repeat` initiates an infinite loop; these are not commonly used in statistical applications but they do have their uses. The only way to exit a repeat loop is to call `break`.

```
# NOTE: pseudocode follows:
x0 <- 1
tol <- 1e-8
repeat {
      x1 <- <computeEstimate> # This is any function that computes an estimate
      if(abs(x1 - x0) < tol) {
            break
      } else {
      x0 <- x1
      }
}
```

However, the loop in the previous slide is a bit dangerous because theres no guarantee it will stop. Better to set a hard limit on the number of iterations (e.g. using a `for` loop) and then report whether convergence was achieved or not.

4.5 next-return

`next` is used to skip an iteration of a loop:

```
for(i in 1:30) {
      if(i <= 20) {
            ## Skip the first 20 iterations
            next
      }
      print(i)
}
```

```
## [1] 21
## [1] 22
## [1] 23
## [1] 24
## [1] 25
## [1] 26
## [1] 27
## [1] 28
## [1] 29
## [1] 30
```

`return` signals that a function should exit and return a given value

5 Other "looping" functions

Writing `for`, `while` loops is useful when programming but not particularly easy when working interactively on the command line. There are some functions which implement looping to make life easier (and sometimes faster).

The functions shown below are very useful and particular of R. Beginning useRs may have difficulty deciding which one is appropriate for their situation or even remembering them all. For this reason, here we present only three very common ones. From the R documentation, and as you gain more skill on their usage, you will find variants of these three basic ones that can cover virtually all data handling needs (or at least 99% of them...). Searching for some materials to write this manual, I came across an excellent explanation in a popular R forum, that you may also find useful. But first, here we go with the basics:

5.1 `apply`

`apply` is used to a evaluate a function over the *margins* of an array.

- It is most often used to apply a function to the rows or columns of a matrix
- It can be used with general arrays, e.g. taking the average of an array of matrices
- It is not really faster than writing a loop, but it works in one line!

About its arguments:

```
str(apply)

## function (X, MARGIN, FUN, ...)
```

`X` is an array

`MARGIN` is an integer vector indicating which margins should be retained.

`FUN` is a function to be applied

`...` (known as *ellipsis*) is for other arguments to be passed to `FUN`

Examples:

```
x <- matrix(rnorm(200), 20, 10)
apply(x, 2, mean) # Mean value of each column

##  [1] -0.03251  0.20723 -0.12776 -0.11488 -0.13957 -0.20509
##  [7] -0.02893  0.03115  0.07112 -0.01157

apply(x, 1, max) # Maximum value of each row

##  [1] 1.5773 1.7977 0.8766 1.4651 2.0667 1.3346 1.5818 1.4841
##  [9] 1.2645 0.5423 1.3503 2.4591 0.8664 1.4603 1.8390 1.8308
## [17] 1.5763 1.9046 1.2650 1.1249
```

For sums and means (most common operations) of matrix dimensions, we have some shortcuts.

> **Note**: The following shortcuts are much faster when dealing with very large matrices, so try to use them when summing/averaging instead of `apply`

- `rowSums(x)` is identical to `apply(x, 1, sum)`
- `rowMeans(x)` is identical to `apply(x, 1, mean)`
- `colSums(x)` is identical to `apply(x, 2, sum)`
- `colMeans(x)` is identical to `apply(x, 2, mean)`

The ellipsis argument (`...`) indicates that other arguments can be passed to the function on which we are using `apply`. For instance, for computing the quartiles 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]    [,5]     [,6]    [,7]
## 25% -1.2070 -0.5794 -1.197 -0.1537 -0.9052 -1.00841 -0.5609
## 75%  0.6937  0.2281  0.587  0.5772  0.3168  0.02961  0.1395
##        [,8]    [,9]   [,10]   [,11]   [,12]   [,13]   [,14]
## 25% -0.3091 -0.3853 -0.1427 -0.1931 -0.6952 -0.4499 -0.8351
## 75%  0.5327  1.3071  0.8050  1.1111  0.8944 -0.1328 -0.1030
##       [,15]   [,16]  [,17]   [,18]   [,19]    [,20]
## 25% -0.8254 0.05252 0.3449 -0.3241 -0.5464 -0.06166
## 75%  0.5162 1.00979 1.0009  0.8873  1.1164  1.10552
```

More examples: average matrix in a 3D array:

```
a <- array(rnorm(2 * 2 * 10), c(2, 2, 5))
a
```

```
## , , 1
##
##         [,1]    [,2]
## [1,] -0.1374 -0.6557
## [2,] -0.7266  0.1129
##
## , , 2
##
##          [,1]    [,2]
## [1,] -0.38048 -0.2348
## [2,] -0.08666  0.9355
##
## , , 3
##
##         [,1]    [,2]
## [1,] -1.0407 -1.2188
## [2,]  0.9364  0.1055
##
## , , 4
##
##         [,1]     [,2]
## [1,] -0.3169 -0.52966
## [2,]  0.7255 -0.04184
##
## , , 5
##
##         [,1]    [,2]
## [1,]  1.5423  0.1623
## [2,] -0.3519 -0.3529
```

```
apply(a, c(1, 2), mean)
```

```
##          [,1]    [,2]
## [1,] -0.06665 -0.4953
## [2,]  0.09934  0.1518
```

```
rowMeans(a, dims = 2)
```

```
##          [,1]    [,2]
## [1,] -0.06665 -0.4953
## [2,]  0.09934  0.1518
```

5.2 tapply

`tapply` is used to apply a function over subsets of a vector.

About its arguments:

```
str(tapply)
```

```
## function (X, INDEX, FUN = NULL, ..., simplify = TRUE)
```

`X` is a vector

`INDEX` is a factor or a list of factors (or else they are coerced to factors)

`FUN` is a function to be applied

`...` contains other arguments to be passed to `FUN`

`simplify` should we simplify the result?

For instance, very commonly used to calculate group means:

```
x <- c(rnorm(10), runif(10), rnorm(10, 1))
f <- gl(3, 10)
tapply(x, f, mean)
```

```
##       1       2       3
## -0.4911  0.4395  0.4634
```

The same but without simplication:

```
tapply(x, f, mean, simplify = FALSE) # A list instead of a vector
```

```
## $`1`
## [1] -0.4911
##
## $`2`
## [1] 0.4395
##
## $`3`
## [1] 0.4634
```

Another example. Find group ranges:

```
tapply(x, f, range)
```

```
## $`1`
## [1] -2.597  1.279
##
## $`2`
## [1] 0.04901 0.88676
##
## $`3`
## [1] -1.478  2.134
```

5.3 `lapply`

`lapply` takes three arguments: a list `X`, a function (or the name of a function) `FUN`, and other arguments via the ellipsis (`...`) argument. If `X` is not a list, it will be coerced to a list using `as.list`.

```
lapply

## function (X, FUN, ...)
## {
##     FUN <- match.fun(FUN)
##     if (!is.vector(X) || is.object(X))
##         X <- as.list(X)
##     .Internal(lapply(X, FUN))
## }
## <bytecode: 0x8adbc98>
## <environment: namespace:base>
```

The actual looping is done internally in C code.

`lapply` always returns a list, regardless of the class of the input, of the same length as the input.

```
x <- list(a = 1:5, b = rnorm(10))
lapply(x, mean)

## $a
## [1] 3
##
## $b
## [1] 0.4088

x1 <- list(a = 1:4, b = rnorm(10), c = rnorm(20, 1), d = rnorm(100, 5))
lapply(x1, mean)

## $a
## [1] 2.5
##
## $b
## [1] 0.4586
##
## $c
## [1] 1.059
##
## $d
## [1] 5.005

lapply(1:4, runif)

## [[1]]
## [1] 0.279
##
## [[2]]
## [1] 0.9749 0.4661
##
## [[3]]
## [1] 0.9304 0.2589 0.6824
##
```

```
## [[4]]
## [1] 0.09122 0.92242 0.90340 0.27565
```

```
lapply(1:4, runif, min = 0, max = 10)
```

```
## [[1]]
## [1] 9.268
##
## [[2]]
## [1] 8.9258 0.1845
##
## [[3]]
## [1] 8.7663 6.0072 0.4404
##
## [[4]]
## [1] 3.363 8.755 6.506 2.166
```

`lapply` and (related functions like `sapply` make heavy use of *anonymous* functions (i.e., functions internally used withi `lapply`, not assigned an identifier):

```
x <- list(a = matrix(1:4, 2, 2), b = matrix(1:6, 3, 2))
lapply(x, function(an) an[,1] + 10)
```

```
## $a
## [1] 11 12
##
## $b
## [1] 11 12 13
```

5.4 `sapply`

`sapply` will try to simplify the result of `lapply` if possible:

- If the result is a list where every element is length 1, then a vector is returned
- If the result is a list where every element is a vector of the same length (¿ 1), a matrix is returned.
- If it can't figure things out, a list is returned

```
x <- list(a = 1:4, b = rnorm(10), c = rnorm(20, 1), d = rnorm(100, 5))
lapply(x, mean) # Compare the outputs
```

```
## $a
## [1] 2.5
##
## $b
```

```
## [1] -0.06776
##
## $c
## [1] 0.9226
##
## $d
## [1] 5.093
```

```
sapply(x, mean)
```

```
##        a        b        c        d
##  2.50000 -0.06776  0.92260  5.09257
```

6 Credits and additional resources

6.1 Credits

This material has been partly elaborated with pieces of information gathered from other sources. In particular:

- Ceballos, M. and N. Cardiel (2013). R Introduction (unpublished lectures manual).
- COURSERA (2013). Computing for data Analysis (on-line course)
- Stackoverflow. URL: `http://stackoverflow.com/questions/tagged/r`

6.2 Additional resources

Look in this useful page for a compilation of available free on-line `R` learning resources: `http://www.introductoryr.co.uk/R_Resources_for_Beginners.html`

Session info

```
sessionInfo()

## R version 3.1.1 (2014-07-10)
## Platform: i686-pc-linux-gnu (32-bit)
##
## locale:
##  [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
##  [3] LC_TIME=en_US.UTF-8        LC_COLLATE=en_US.UTF-8
##  [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
##  [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
##  [9] LC_ADDRESS=C               LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets
## [6] methods   base
##
## other attached packages:
## [1] knitr_1.6
##
## loaded via a namespace (and not attached):
## [1] evaluate_0.5.5 formatR_1.0    highr_0.3
## [4] stringr_0.6.2  tools_3.1.1
```