### R Basics

# Obtaining and Installing R

The first thing to do in order to use R is to get a copy of it. This can be done on:

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

Windows, Linux and Mac installation follow similar steps and may include the selection of a CRAN mirror. Choose any of them, preferrably the closest to your current location.

We recommend that you also use Rstudio. Rstudio is an interface that makes R easier to use. It includes a code editor, debugging and visualization tools. R Studio is availablde at:

https://www.rstudio.com/products/rstudio/download/

Once R and RStudio installed open RStudio and get familiarized with R. The environment is quite plain and simple. There is a main application window and, within it, a console window.

# The R command line

The command line is where you interact with R. This is designated in red and has a ">" symbol. At the command line you type in code telling R what to do. You can use R as calculator to perform basic mathematical operations. Try typing some basic arithmetic tasks at the command line. Hit enter and R will compute the requested result:

```
3+7
```

```
[1] 10
```

```
2*9
```

```
[1] 18
```

If you enter something incorrect, or that R does not understand, you will get an error message rather than a result:

```
hello
```

```
Error in eval(expr, envir, enclos): object 'hello' not found
```

## Commenting text

In computer programming, a comment is a programmer-readable explanation or annotation in the source code of a computer program. They are added with the purpose of making the source code easier for humans to understand, and are generally ignored by compilers and interpreters. The syntax of comments in various programming languages varies considerably. In R we use the `#` operator to comment text. The text/code written after a `#` sign is not executed

```
#comment: we are guessing  1+2
1+2
```

```
[1] 3
```

### Built-in functions

Almost everything in R is done through functions. When calling a function, you'll need the name of the function and a list of arguments for the function to work. One of the more basic functions in R is the `print` function used to print characters or strings.

```
print('Hello world')
```

```
[1] "Hello world"
```

You can also create your own functions (See Section 5) For advanced functions it is often necessary to install specific packages (see next section).

### Use the R documentation

If you need some help regarding an R function you can use the command `?` to access the R documentation

```
?print
```

### Packages

R packages are a collection of R functions, complied code and sample data. They are stored under a directory called "library" in the R environment. By default, R installs a set of packages during installation. More packages are added later, when they are needed for some specific purpose. When we start the R console, only the default packages are available by default. Other packages which are already installed have to be loaded explicitly to be used by the R program that is going to use them.

All the packages available in R language are listed at

https://cran.r-project.org/web/packages/available_packages_by_name.html.

To install a new package, use the function `install.packages()`

```
install.packages("ggplot2")
```

To load a (recently installed) package, use the function `library`

```
library(ggplot2)
```

To see a list of already installed packages use the following command:

```
library()
```

## Types of data in R

### Scalars

The simplest type of object is a scalar. A scalar is an object with one value. To create a scalar data object, simply assign a value to a variable using the assignment operator "<-". Note that the equals sign is not the assignment operator in R although it is used for other functionalities.

```
x<-5
y<-2
```

You can manipulate scalar objects in R and perform all sorts of algebraic calculations.

```
x-y
```

```
[1] 3
```

```
x*y-7
```

```
[1] 3
```

```
z<- x+y
```

## Characters

A character object is used to represent string values in R.

```
fname<-"Joe"
lname <-"Smith"
```

## Logical objects

A logical object can only take on two values, TRUE or FALSE. Logical data can be entered simply as T or F (no quotes).

```
true <-T
false <-F
```

## Vectors

Vectors are the data objects probably most used in R. A vector can be defined as a set of scalars arranged in a one-dimensional array. Essentially a scalar is a one-dimensional vector. Data values in a vector are all the same mode, but a vector can hold data of any mode. Vectors may be entered using the `c()` function (or "combine values" in a vector) and the assignment operator.

```
vector <- c(2,5,5,3,3,6,2,3,5,6,3)
vector
```

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

```
v1<-c(1,2,3)
v2<-c(4,5,6)
```

You can perform all kind of operations with vectors in R.

```
# Accessing an element of a vector
v1[2]
```

```
[1] 2
```

```
#sum of two vectors
z<-v1+v2
z
```

```
[1] 5 7 9
```

```
#product of two vectors
z<-v1*v2
z
```

```
[1]  4 10 18
```

Note that most operations on vectors work position-wise (position by position). If we perform a multiplication, we would get a vector with each of its positions being the multiplication of the corresponding positions in the multiplied vectors.

If two vectors are of unequal length (NOT RECOMMENDED) the shorter one will be recycled in order to match the longer vector. For example, the following vectors u and v have different lengths, and their sum is computed by recycling values of the shorter vector u.

```
v1<-c(1,2,3)
v2 <-c(4,5)
v1+v2
```

```
Warning in v1 + v2: longer object length is not a multiple of shorter
object length
```

```
[1] 5 7 7
```

You can also create a vector by concatenating existing vectors with the `c()` function:

```
z<- c(v1, v2)
z
```

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

## Factors

Conceptually, factors are variables in R which take on a limited number of different values; such variables are often refered to as categorical variables. One of the most important uses of factors is in statistical modeling; since categorical variables enter into statistical models differently than continuous variables, storing data as factors insures that the modeling functions will treat such data correctly.

```
 mons = c("March", "April", "January", "November", "January", "September", "October", "September", "Nov
mons <-factor(mons)
mons
```

```
 [1] March     April     January   November  January   September October
 [8] September November  August    January   November  November  February
[15] May       August    July      December  August    August    September
[22] November  February  April
11 Levels: April August December February January July March ... September
```

Notice that this creates "levels" based on the factor values (these are the values of categorical variables).

One basic operations on table is bu use of the function `table()`

```
table(mons)
```

```
mons
    April    August  December  February   January      July     March
        2         4         1         2         3         1         1
      May  November   October September
        1         5         1         3
```

## Matrices and arrays

Matrices are collections of data values in two dimensions. In mathematics, matrices have many applications, and a good course in linear algebra is required to fully appreciate the usefulness of matrices. An array is a

matrix with more than two dimensions. Formatting data as matrices and arrays provides an efficient data structure to perform calculations in a computationally fast and efficient manner.

To create a matrix in R, use the `matrix()` function, which takes as arguments a data vector and specification parameters for the number of rows and columns.

```
mat<-matrix(c(2,3,1,5),nrow=2,ncol=2)
mat
```

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

To create an array in R, use the `array()` function, where you give the data as the first argument, a vector with the sizes of the dimensions as the second argument. The number of dimension sizes in that argument gives you the number of dimensions. For example, you make an array with four columns, three rows, and two "tables" like this:

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

```
, , 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
```

Basic operations on matrices and arrays are also elementwise.

```
m1 <- matrix(1:9, ncol=3)
m2 <- matrix(9:1, ncol=3)
m1
```

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

```
m2
```

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

```
m1+m2
```

```
     [,1] [,2] [,3]
[1,]   10   10   10
[2,]   10   10   10
[3,]   10   10   10
```

## Lists

Lists are the "everything" data objects. A list is a generic vector containing other objects. A list, unlike a vector, can contain data with different modes under the same variable name and encompass other data objects. Lists are useful for organizing information. Creating a list is very simple; just use the list () function to assign to a variable the list values.

```
my_list <- list(1, 2, "hello", c(1,2), m1)
my_list
```

```
[[1]]
[1] 1

[[2]]
[1] 2

[[3]]
[1] "hello"

[[4]]
[1] 1 2

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

Note that list values are indexed with double bracket sets rather than single bracket sets used by other data objects. In order to reference a list member directly, we have to use the double square bracket "[[]]" operator. If we use one single square braquet operator, we retrieve a list slice.

```
my_list[[1]]
```

```
[1] 1
```

```
my_list[[3]]
```

```
[1] "hello"
```

```
my_list[[5]]
```

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

```
my_list[5]
```

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

## Dataframes

Data frames are versatile data objects you can use in R. A data frame is a table or a two-dimensional array-like structure in which each column contains values of one variable and each row contains one set of values from each column.

A dataframe will have the following characteritsics:

- The column names should be non-empty.
- The row names should be unique.
- The data stored in a data frame can be of numeric, factor or character type.
- Each column should contain same number of data items.

To create a dataframe, we'll use the function `data.frame()`

```
id <- c(1:5)
name <- c("Nick", "Dan", "Michelle", "Ryan", "Mary")
salary <- c(623.3,515.2,611.0,729.0,843.25)
sdata <- data.frame(id=id, names=name, salary=salary)
sdata
```

```
  id    names salary
1  1     Nick 623.30
2  2      Dan 515.20
3  3 Michelle 611.00
4  4     Ryan 729.00
5  5     Mary 843.25
```

Note that it is necessary to specify the names of the fields in the dataframe.

To get the structure of the data frame, we'll use the function `str()`

```
str(sdata)
```

```
'data.frame':   5 obs. of  3 variables:
 $ id    : int  1 2 3 4 5
 $ names : Factor w/ 5 levels "Dan","Mary","Michelle",..: 4 1 3 5 2
 $ salary: num  623 515 611 729 843
```

We can also print the summary of the dataframe

```
summary(sdata)
```

```
       id         names       salary
 Min.   :1   Dan     :1   Min.   :515.2
 1st Qu.:2   Mary    :1   1st Qu.:611.0
 Median :3   Michelle:1   Median :623.3
 Mean   :3   Nick    :1   Mean   :664.4
 3rd Qu.:4   Ryan    :1   3rd Qu.:729.0
 Max.   :5                Max.   :843.2
```

To extract a specific column from a dataframe using the column name, we'll use the `$` operator

```
sdata_names <- sdata$name
sdata_names
```

```
[1] Nick     Dan      Michelle Ryan     Mary
Levels: Dan Mary Michelle Nick Ryan
```

```
sdata_salaries <- sdata$salary
sdata_salaries
```

```
[1] 623.30 515.20 611.00 729.00 843.25
```

```
# Extract Specific columns.
result <- data.frame(sdata$name,sdata$salary)
result
```

```
  sdata.name sdata.salary
1       Nick       623.30
2        Dan       515.20
3   Michelle       611.00
4       Ryan       729.00
5       Mary       843.25
```

For big dataframes, it may be useful to use the function `head()` to print only the first 6 elements (lines) and see the aspect of the data.frame. As an example we'll use the '`iris` built-in data frame. This famous (Fisher's or Anderson's) iris data set gives the measurements in centimeters of the variables sepal length and width and petal length and width, respectively, for 50 flowers from each of 3 species of iris. The species are Iris setosa, versicolor, and virginica. `iris` is a data frame with 150 cases (rows) and 5 variables (columns) named Sepal.Length, Sepal.Width, Petal.Length, Petal.Width, and Species.

```
head(iris)
```

```
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
```

Other operations on dataframes include retrieving the names of the fields, the number of columns or the number of rows of he dataframes.

```
names(iris)
```

```
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"
[5] "Species"
```

```
nrow(iris)
```

```
[1] 150
```

```
ncol(iris)
```

```
[1] 5
```

## Checking and changing types of data

Sometimes you may forget or not know what type of data you are dealing with,so R provides functionality for you to check this. If you just want to know what class is a data object, use the function `class()`

```
class(x)
```

```
[1] "numeric"
```

```
class(fname)
```

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

```
class(fname)
```

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

```
class(true)
```

```
[1] "logical"
```

```
class(v1)
```

```
[1] "numeric"
```

```
class(mons)
```

```
[1] "factor"
```

```
class(my_list)
```

```
[1] "list"
```

```
class(mat)
```

```
[1] "matrix"
```

```
class(iris)
```

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

If you want to know if an object is of a specific class, there is a set of `is.what` functions, which provide identification of data object types and modes.

```
is.numeric(x)
```

```
[1] TRUE
```

```
is.numeric(fname)
```

```
[1] FALSE
```

```
is.character(fname)
```

```
[1] TRUE
```

```
is.logical(true)
```

```
[1] TRUE
```

```
is.vector(v1)
```

```
[1] TRUE
```

```
is.data.frame(iris)
```

```
[1] TRUE
```

Sometimes you may want to change the data object type or mode. To do this R provides a series of "as.what" functions where you convert your existing data object into a different type or mode. Don't forget to assign the new data object to a variable (either overwriting the existing variable or creating a new one) because otherwise the data object conversion will only be transient.

```
y <- as.numeric(x)
y
```

```
[1] 5
```

```
as.factor(v1)
```

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

```
Levels: 1 2 3
```

```
as.data.frame(mat)
```

```
  V1 V2
1  2  1
2  3  5
```

## Controling the workspace

The workspace is your current R working environment and includes any user-defined objects (vectors, matrices, data frames, lists, functions). At the end of an R session, the user can save an image of the current workspace that is automatically reloaded the next time R is started. Commands are entered interactively at the R user prompt. Up and down arrow keys scroll through your command history.

You will probably want to keep different projects in different physical directories. Here are some standard commands for managing your workspace.

```
#get the working directory
getwd()
```

```
# set the working directory
#setwd("/Users/hb493/Desktop/R/projects/Rbasics")
getwd()
```

```
[1] "/Users/hb493/Desktop/github/Rbasics"
```

```
#list the objects in hte current workspace
ls()
```

```
 [1] "false"          "fname"          "id"             "lname"
 [5] "m1"             "m2"             "mat"            "mons"
 [9] "my_array"       "my_list"        "name"           "result"
[13] "salary"         "sdata"          "sdata_names"    "sdata_salaries"
[17] "true"           "v1"             "v2"             "vector"
[21] "x"              "y"              "z"
```

```
# save the workspace to the file .RData in the cwd
save.image()
```

```
#save history of commands in an R script
savehistory(file="Rbasics.R")

#Run the commands on a script file
source("Rbasics.R")

# save specific objects to a file
# if you don't specify the path, the current wd is assumed
save(list(mon, v1, v2, mat, sdata) ,file="Rbasics.RData")

# load a workspace into the current session
# if you don't specify the path, the cwd is assumed
load("Rbasics.RData")
```

# Programming with R

## Logical operators

Logical and comparison operators are used when you want to selectively execute code based on certain conditions.

The main comparison operators are:

- equal: `==`
- not equal: `!=`
- greater/less than: `> <`
- greater/less than or equal: `>= <=`

The logical operators are

- and: `&`
- or: `|`
- not: `!`

To execute code on a certain condition we'll use `if` statements: If statements operate on length-one logical vectors. The syntax in R for if statements is:

```
if(cond1=true) { cmd1 } else { cmd2 }
```

Contrarily to other programmming languages such as python, identation is not critical in R.

Examples:

```
if(3==0) {
    print(1)
} else {
    print(2)
}
```

```
[1] 2
```

```
a <- 2
b <- 3
c <- 4
if(a<b & b<c) x<-a+b+c
x
```

```
[1] 9
```

```
if(a!=2) u<-a
u
```

```
Error in eval(expr, envir, enclos): object 'u' not found
```

## Looping

Control by repetition, or looping, allows you to efficiently repeat code without having to write the same code over and over. In R, two common looping expressions are `while` and `for`.

The syntax in R for a while looping is

```
while (condition controlling flow is true) { perform task }
```

Example:

```
x<-0
while(x<=5){x<-x+1}
x
```

```
[1] 6
```

For loops are used to iterate through a process a specified number of times. A counter variable (usually designated by a lowercase letter "i") is used to count how many times the loop is executed. The syntax in R for a for looping is:

```
for (i in start:finish) { perform task }
```

Example:

```
# create an empty vecotr that we are going to fill in the loop
y<-c()
for(i in 1:10)
{
  y[i]<-i
}
y
```

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

## Functions

### Built-in functions

Even in the base package (with no additional packages installed) R supplies a large number of pre-written functions for you to use. Additional packages are filled with additional functions, usually with related functions for related tasks packaged together. The simplest functions are functions that perform basic mathematical tasks:

```
#square root
a<- sqrt(9)
a
```

```
[1] 3
```

```
#absolute value
abs(-2)
```

```
[1] 2
```

```
# Trigonometric functions
sin(pi)
```

```
[1] 1.224647e-16
```

```
cos(pi)
```

```
[1] -1
```

```
tan(pi)
```

```
[1] -1.224647e-16
```

```
# exponential
exp(0)
```

```
[1] 1
```

```
exp(1)
```

```
[1] 2.718282
```

```
#logarithms
log(exp(2))
```

```
[1] 2
```

```
log2(8)
```

```
[1] 3
```

```
log10(100)
```

```
[1] 2
```

```
#round
ceiling(5.7)
```

```
[1] 6
```

```
floor(5.7)
```

```
[1] 5
```

```
round(5.7)
```

```
[1] 6
```

```
# basic statistics
x <- c(1,2,3,2,4,5,2,3,4,3,4,5)
mean(x)
```

```
[1] 3.166667
```

```
sd(x)
```

```
[1] 1.267304
```

```
median(x)
```

```
[1] 3
```

### Writing functions

Whether you need to accomplish a particular task and there is not a dedicated function or library already existing or because you are not aware of this functionality or have no time to find the solution, you may need to write your own functions.

The syntax for creating a question is :

```
function_name <- function(arguments)
{
  computations on the arguments
  some other code
}
```

For example,

```r
# Define a simple function
square<-function(n)
{
  # Compute the square of integer `n`
  n*n
}
square(10)
```

```
[1] 100
```