# Diplomado en R

*Francisco Javier Luna Vázquez*

*2017-04-17*

# Contents

# Chapter 1

# Prerequisitos

# Chapter 2

# Introducción a R

## 2.1 Primeros pasos

Para interactuar con **R** se dispone de una potente línea de comandos y en un principio, la manera más sencilla de ver **R** es que puede ser usado como una calculadora:

Como sumar:

```
10+5
```

```
## [1] 15
```

o dividir:

```
10/5
```

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

Sin embargo, su potencial va más allá...

Principalmente por que este documento ha sido diseñado en **R** y **RMarkdown** ;)

### 2.1.1 Establecer directorios de trabajo

Al ejecutar **R**, este establece un directorio de trabajo, es decir, establece una carpeta donde guardar los datos ejecutados en **R**, misma que será la carpeta que usara para buscar, leer y escribir los archivos (de ser requerido) durante la sesión (es decir, mientras se mantiene el programa abierto), por ello, para obtener el directorio de trabajo actual basta con usar el siguiente comando:

```
getwd()
```

```
## [1] "/home/frahik/Documentos/R-project/DiplomadoR"
```

Dicho directorio va a variar dependiendo del sistema, por ello, más de una vez será necesario modificar esta ruta, para lo cual se usara el comando `setwd("Ruta")`, el cual varia dependiendo del Sistema Operativo, como buena practica del programador, se recomienda establecer el directorio de trabajado dentro de la carpeta de Documentos, además de que es más sencillo encontrar los proyectos de trabajo, si se usa la ruta **ruta relativa** y no la **ruta física**, si se requiere mover todos los archivos para trabajar en otra computadora, no se tendrá que volver a modificar todo el directorio de trabajo.

**Ejemplo usando la Ruta relativa en Linux**

```
setwd("~/Documentos/Curso") #Linux
```

**Ejemplo usando la *Ruta física* y la *ruta relativa* en Windows**

```
setwd("C:\\Usuario\\Documentos\\Curso") #MALA PRACTICA (Ruta física)
setwd("~/Curso") #BUENA PRACTICA (Ruta relativa del comando anterior)
```

### 2.1.2 Creación de objetos/variables

En **R** podemos crear y manipular objetos asignándole valores, cadenas de texto, funciones y un largo etc. Por ser el primer contacto con **R**, crearemos un objeto (izquierda del símbolo `=`), asignándole como valor la cadena de texto "Hola mundo". Es importante aclarar que es necesario que se pongan dobles comillas `" "` o comillas simples `' '` al escribir una cadena de texto, de otra manera será interpretado como uno o varios objetos, ejemplo:

```
saludo = Hola
```

```
## Error in eval(expr, envir, enclos): objeto 'Hola' no encontrado
```

Por lo que, para insertar la cadena de texto "Hola mundo" en la variable `saludo`, se deberá hacer como se mencionó anteriormente:

```
saludo = "Hola mundo"
```

Ahora para mostrar en consola lo que contiene la *variable* u *objeto* `saludo`, lo escribiremos tal cual y obtendremos una salida muy similar a la siguiente.

```
## [1] "Hola mundo"
```

**NOTAS IMPORTANTES:**

> **R** es sensible a las MAYÚSCULAS y minúsculas, por lo que `saludo` no es igual a `Saludo`, ni a `SALUDO`.
>
> **R** NO requiere explicitar que tipo de valores van a contener las variables,

Otra manera de asignar valores a las variables es mediante el símbolo `<-` que se compone de un menor que y el signo de menos.

```
x <- 10 + 5
```

> «Podemos ver a las variables como una persona, cada persona tiene su nombre para poder ser identificadas sobre el resto de las personas, pero bajo la manera de ordenar del lenguaje R, si dos personas quieren llamarse de la misma manera, la nueva persona tiene que 'matar' a la persona ya existente para poder tomar su lugar.»

Se recomienda incluir un espacio simple a cada lado del operador de asignación para incrementar la legibilidad. Pero NO coloques un espacio entre el `<` y el `-` que forman la flecha, recuerda que a pesar de estar compuesto por dos caracteres es un único símbolo.

Es posible reasignar un valor a la variable que hemos creado, así como reutilizar el valor de la variable para realizar un calculo:

```
y <- x + 5
y
```

Ahora reutilizaremos el valor de la variable `y`:

```
## [1] 20
```

> Recuerde que aunque pareciera algo matemático, los símbolos `<-` y `=` no funcionan como un «*igual*», si no, como un «*equivale a*» o una «*asignación*», por que a partir de ese momento, el valor de la derecha se le asigna al de la izquierda.

## 2.2 Clases de datos

Existen 4 tipos de clases de objetos con las que tendremos que trabajar en R, cada uno tiene sus ventajas y desventajas, así que se verá de manera detalla como crear cada uno de ellos, acceder a sus valores, así como posibles problemas.

### 2.2.1 Vector

Desde que empezamos con la variable `saludo`, estábamos trabajando con vectores, sin embargo, era un vector de indice 1, dado que en `R` no existen como en otros lenguajes las variables individuales, todas a las que se les asigna un valor o más, son tratados inicialmente como vectores.

La manera de representar un vector es de la siguiente forma:

| Valor1 | Valor2 | Valor3 | ... | Valorn |
|---|---|---|---|---|

> Una forma de ver a los vectores, es como la fila de espera del banco, todos tienen un turno único, pero que puede ser transferible, por lo que si mando a llamar el primer turno, solo el cliente con ese turno sera el que pasará a ventanilla, pero mientras, el cliente con turno 10, puede ser que esté guardando el lugar para un amigo y cuando el amigo llegue, el nuevo cliente sustituira al actual cliente con el turno 10.

Una vez entendido el mensaje anterior, existen varias formas de declarar un vector de más de un valor, la manera más sencilla es a través de la función concatenar `c(...)`; la cual es una función genérica que combina los valores separado por comas en un vector.

Ejemplos: Vector con un varios datos y un `NA`.

```
vector <- c(1:9,NA,10:15)
vector
```

```
##  [1]  1  2  3  4  5  6  7  8  9 NA 10 11 12 13 14 15
```

Vector de nombres de personas

```
vectorNombres <- c("Francisco","Claudia","Valeria","Fernando","Julia")
vectorNombres
```

```
## [1] "Francisco" "Claudia"   "Valeria"   "Fernando"  "Julia"
```

Vector con los datos de otros vectores

```
vec1 <- 1
vec2 <- 20:30
vecFinal <- c(vec1,vec2)
vecFinal
```

```
##  [1]  1 20 21 22 23 24 25 26 27 28 29 30
```

Ya que hemos visto como se crea un vector ahora veremos como acceder y trabajar con los valores de un vector.

#### 2.2.1.1 Acceder a uno o varios datos del vector.

Tenemos el siguiente vector que tiene almacenado el nombre de 6 clientes.

```
vector <- c("Francisco","Claudia","Valeria","Fernando","Julia", "Osval")
```

| Francisco | Claudia | Valeria | Fernando | Julia | Osval |
|---|---|---|---|---|---|

En **R** a diferencia de otros lenguajes de programación como **C** o **Java**, los índices de un vector o una matriz siempre inician en 1 y no en 0 como en esos otros lenguajes, por lo que la tabla anterior se podría observar de la siguiente manera:

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| Francisco | Claudia | Valeria | Fernando | Julia | Osval |

Donde el primer indice, hace referencia al primer elemento del vector, el segundo indice al segundo elemento y así de manera consecutiva.

Por lo que para acceder a uno o varios elementos se pueden utilizar los siguientes códigos:

- Un elemento en especifico

```
vector[3]
```

```
## [1] "Valeria"
```

- Varios elementos en secuencia.

```
vector[2:4]
```

```
## [1] "Claudia"  "Valeria"  "Fernando"
```

- Un elemento especificado anteriormente.

```
n <- 3
vector[n]
```

```
## [1] "Valeria"
```

- Varios elementos especificados

```
vector[c(1,2,5)]
```

```
## [1] "Francisco" "Claudia"   "Julia"
```

- Todos los elementos, excluyendo uno especifico.

```
vector[-3]
```

```
## [1] "Francisco" "Claudia"   "Fernando"  "Julia"     "Osval"
```

- Todos los elementos, excluyendo varios.

```
vector[-c(4:6)]
```

```
## [1] "Francisco" "Claudia"   "Valeria"
```

- Almacenar los datos extraídos en otra variable.

```
grupo1 <- vector[-c(4:6)]
grupo1
```

```
## [1] "Francisco" "Claudia"   "Valeria"
```

- Modificar un elemento especifico de un vector

```
vector[4] <-  "Alberto"
vector
```

```
## [1] "Francisco" "Claudia"   "Valeria"   "Alberto"   "Julia"     "Osval"
```

- Modificar varios elementos de un vector

```
vector[4:6] <- NA
vector
```

```
## [1] "Francisco" "Claudia"   "Valeria"   NA          NA          NA
```

- Encontrar **las posiciones** de los valores NA (Not Available) en un vector

```
is.na(vector)
```

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

- Encontrar **las posiciones** de los valores que no sean NA en un vector

```
!is.na(vector)
```

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

- Extrar los elementos de las posiciones en los que los valores **NO** son NA

```
which(!is.na(vector) == TRUE)
```

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

```
#Alternativa
which(is.na(vector) == FALSE)
```

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

- Extrar las posiciones del vector que cumplan con una consulta

```
which(vector == "Francisco")
```

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

- Modificar las posiciones del vector que cumplan con una consulta

```
vector[which(vector == "Francisco")] <- "Francisco Javier"
```

#### 2.2.1.2 Ordenar los datos de un vector

Para ordenar los datos existe la función `sort(...)`.

- Ordenar un vector de forma creciente

```
sort(vector)
```

```
## [1] "Claudia"          "Francisco Javier" "Valeria"
```

- Ordenar un vector de forma decreciente

```
sort(vector, decreasing = T)
```

```
## [1] "Valeria"          "Francisco Javier" "Claudia"
```

- Ordenar un vector y conocer sus antiguas posiciones.

```
sort(vector, decreasing = T, index.return = TRUE)
```

```
## $x
## [1] "Valeria"          "Francisco Javier" "Claudia"
##
## $ix
## [1] 3 1 2
```

Este último método retorna dos listas, la primera es `$x` que representa la lista de los valores ordenados y la segunda es `$ix` que representa las antiguas posiciones de los valores.

#### 2.2.1.3 Tratar los datos de un vector

Tenemos un vector con los gastos de la última semana:

```
gastos <- c(150,120,300,250,400,380,100)
```

que se pueden representar de la siguiente manera:

| | | | | | | |
|---|---|---|---|---|---|---|
| 150 | 120 | 300 | 250 | 400 | 380 | 100 |

En **R** ya existen varias funciones para ser utilizadas, se mostrará el funcionamiento de algunas de ellas y otras más podrán ser encontradas en el Capitulo 5 (Apéndices).

- El menor gasto en la semana

```
min(gastos)
```

```
## [1] 100
```

- El mayor gasto en la semana

```
max(gastos)
```

```
## [1] 400
```

- Obtener el promedio de gasto en la semana.

```
mean(gastos)
```

```
## [1] 242.8571
```

- Obtener el total del gasto de la semana, (Sumar todos los elementos).

```
sum(gastos)
```

```
## [1] 1700
```

- Obtener el total del gasto de una semana laboral (Lunes-Viernes), (Sumar elementos específicos)

```
sum(gastos[1:5])
```

```
## [1] 1220
```

- Obtener la varianza del gasto semanal

```
var(gastos)
```

```
## [1] 15157.14
```

- Obtener la desviación estándar del gasto semanal

```
sd(gastos)
```

```
## [1] 123.1143
```

- Obtener un resumen de los datos

```
summary(gastos)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   100.0   135.0   250.0   242.9   340.0   400.0
```

Este último comando muestra mucha información de utilidad, el valor mínimo, el primer cuantil, la mediana, el promedio, el tercer cuantil y el valor máximo dentro del vector. Con el cuál, se puede ahorrar tiempo a que si se utiliza de manera separada como se ve en los primeros tres comandos.

#### 2.2.1.4 Problemas comunes

##### 2.2.1.4.1 Mayúsculas y mínusculas

Como ya se mencionó anteriormente, **R** es sensible a las Mayúsculas y Minúsculas, por lo que las funciones no funcionaran si se ponen total o parcialmente en Mayúsculas, para comprender ésto veremos dos ejemplos:

Crear un vector: MAL [X]

```
vector <- C(1,2,3)
```

```
## Error in C(1, 2, 3): object not interpretable as a factor
```

BIEN [O]

```
vector <- c(1,2,3)
```

Sumar los datos de un vector: MAL [X]

```
vector <- c(1,2,3)
SUM(vector)
```

```
## Error in eval(expr, envir, enclos): no se pudo encontrar la función "SUM"
```

BIEN [O]

```
vector <- c(1,2,3)
sum(vector)
```

```
## [1] 6
```

### 2.2.2 Matriz

Para crear una matriz en **R** se utiliza la función

```
matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE, dimnames = NULL)
```

Donde `matrix()` corresponde al nombre de la función y todo lo que está dentro de los paréntesis son los argumentos de dicha función.

| Argumentos | Significado del argumento |
|---|---|
| `data` | Son los datos a ingresar en la matriz. |
| `nrow` | Número deseado de filas. |
| `ncol` | Número deseado de columnas. |
| `byrow` | Valor lógico. Si es falso (valor por defecto), la matriz se llena por orden columna, de otra manera se llenará primero por filas. |
| `dimnames` | Utilizado para darles nombres a las filas y a las columnas, respectivamente. |

Algunos ejemplos de como crear una matriz:

- Una matriz de NA de 5 * 5:

```
m <- matrix(NA, nrow = 5, ncol = 5)
```

- Una matriz desde los datos de un vector:

```
vector <- c("Francisco", 500, "Alberto", 200, "Enrique", 650, "Juan", 300)
m <- matrix(vector, ncol = 2, byrow = T)
m
```

- Una matriz desde los datos de dos o más vectores:

```
Nombre <- c("Francisco", "Alberto", "Enrique", "Juan")
Puntos <- c(500, 200, 650, 300)
m <- matrix(c(Nombre, Puntos), ncol = 2)
m
```

```
##      [,1]        [,2]
## [1,] "Francisco" "500"
## [2,] "Alberto"   "200"
## [3,] "Enrique"   "650"
## [4,] "Juan"      "300"
```

- Una matriz desde los datos de dos o más vectores con cbind:

```
Nombre <- c("Francisco", "Alberto", "Enrique", "Juan")
Puntos <- c(500, 200, 650, 300)
m <- cbind(Nombre, Puntos)
m
```

```
##      Nombre      Puntos
## [1,] "Francisco" "500"
## [2,] "Alberto"   "200"
## [3,] "Enrique"   "650"
## [4,] "Juan"      "300"
```

- Una matriz desde los datos de dos o más vectores con rbind:

```
Cliente1 <- c("Francisco", 500)
Cliente2 <- c("Alberto", 200)
Cliente3 <- c("Enrique",  650)
m <- rbind(Cliente1, Cliente2, Cliente3)
m
```

```
##          [,1]        [,2]
## Cliente1 "Francisco" "500"
## Cliente2 "Alberto"   "200"
## Cliente3 "Enrique"   "650"
```

#### 2.2.2.1 Trabajar con matrices

Continuaremos trabajando con la siguiente matriz, es necesario recordar que se puede llamar de cualquier forma, en este caso se llamará de manera generica, `matriz` :

```
Nombre <- c("Francisco", "Alberto", "Enrique", "Juan", "Francisco", "Alberto", "Enrique", "Juan")
Puntos <- c(500, 200, 650, 300, 300, 350, 600, 400)
matriz <- cbind(Nombre, Puntos)
matriz
```

```
##      Nombre      Puntos
## [1,] "Francisco" "500"
## [2,] "Alberto"   "200"
## [3,] "Enrique"   "650"
## [4,] "Juan"      "300"
## [5,] "Francisco" "300"
## [6,] "Alberto"   "350"
## [7,] "Enrique"   "600"
## [8,] "Juan"      "400"
```

- Nombrar las columnas de la matriz

```
colnames(matriz) <- c("Cliente", "Puntos")
head(matriz)
```

```
##      Cliente     Puntos
## [1,] "Francisco" "500"
## [2,] "Alberto"   "200"
## [3,] "Enrique"   "650"
## [4,] "Juan"      "300"
## [5,] "Francisco" "300"
## [6,] "Alberto"   "350"
```

- Nombrar las filas de la matriz

```
rownames(matriz) <- c("Compra1", "Compra2", "Compra3", "Compra4", "Compra5", "Compra6", "Compra7", "Comp
matriz
```

```
##         Cliente     Puntos
## Compra1 "Francisco" "500"
## Compra2 "Alberto"   "200"
## Compra3 "Enrique"   "650"
## Compra4 "Juan"      "300"
## Compra5 "Francisco" "300"
## Compra6 "Alberto"   "350"
## Compra7 "Enrique"   "600"
## Compra8 "Juan"      "400"
```

- Nombrar las columnas y las filas de la matriz

```
dimnames(matriz) <- list(c("C1", "C2", "C3", "C4", "C5", "C6", "C7", "C8"),c("Cliente", "Puntos"))
matriz
```

```
##    Cliente     Puntos
## C1 "Francisco" "500"
## C2 "Alberto"   "200"
## C3 "Enrique"   "650"
## C4 "Juan"      "300"
## C5 "Francisco" "300"
## C6 "Alberto"   "350"
## C7 "Enrique"   "600"
## C8 "Juan"      "400"
```

- Obtener los primeros valores de una matriz:

```
head(matriz,3)
```

```
##    Cliente     Puntos
## C1 "Francisco" "500"
```

```
## C2 "Alberto"   "200"
## C3 "Enrique"   "650"
```

- Obtener los últimos valores de una matriz:

```
tail(matriz,3)
```

```
##    Cliente   Puntos
## C6 "Alberto" "350"
## C7 "Enrique" "600"
## C8 "Juan"    "400"
```

- Obtener los nombres de los clientes:

```
matriz[,1]
```

```
##          C1          C2          C3          C4          C5          C6
## "Francisco"   "Alberto"   "Enrique"      "Juan" "Francisco"   "Alberto"
##          C7          C8
##   "Enrique"      "Juan"
```

- Obtener los nombres ordenados:

```
sort(matriz[,1])
```

```
##          C2          C6          C3          C7          C1          C5
##   "Alberto"   "Alberto"   "Enrique"   "Enrique" "Francisco" "Francisco"
##          C4          C8
##      "Juan"      "Juan"
```

- Obtener los nombres de los clientes sin repetir:

```
unique(matriz[,1])
```

```
## [1] "Francisco" "Alberto"   "Enrique"   "Juan"
```

- Obtener los nombres de los clientes sin repetir y ordenados:

```
sort(unique(matriz[,1]))
```

```
## [1] "Alberto"   "Enrique"   "Francisco" "Juan"
```

- Obtener al primer cliente y sus puntos:

```
matriz[1,]
```

```
##     Cliente      Puntos
## "Francisco"       "500"
```

- Obtener la suma de los puntos (ver problemas comunes):

```
sum(as.numeric(matriz[,2]))
```

```
## [1] 3300
```

#### 2.2.2.2 Problemas comunes

Estos son algunos de los problemas más comunes al momento de intentar trabajar con una matriz:

##### 2.2.2.2.1 No puedo hacer la suma de números.

Es ocasiones **R** tiende a asignar nuestros datos de otra forma que no es la que nosotros esperamos, en este caso tenemos el siguiente error:

```
sum(matriz[,2])
```

```
## Error in sum(matriz[, 2]): 'type' (character) de argumento no válido
```

El error que retorna **R** es que el tipo character no es un argumento válido, esto significa que los datos que nosotros ingresamos son del tipo caracter, para comprobar esto, usaremos la función `type()`:

```
typeof(matriz[,2])
```

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

```
2
```

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

#### 2.2.2.2.2 Los nombres en `dimnames` no concuerdan con las dimensiones de la matriz.

#### 2.2.2.2.3 Los datos no se ordenan o se desordenan.

#### 2.2.2.2.4 Los datos no concuerdan con las dimensiones de la matriz

Supongamos que tenemos un vector en secuencia del 1 hasta el 31 y queremos representar solo el mes de mayo del 2017

```
vector <- 1:31
matrix(vector, nrow = 5, ncol=7, byrow=T)
```

```
## Warning in matrix(vector, nrow = 5, ncol = 7, byrow = T): la longitud de
## los datos [31] no es un submúltiplo o múltiplo del número de filas [5] en
## la matriz
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
## [1,]    1    2    3    4    5    6    7
## [2,]    8    9   10   11   12   13   14
## [3,]   15   16   17   18   19   20   21
## [4,]   22   23   24   25   26   27   28
## [5,]   29   30   31    1    2    3    4
```

Vemos que R nos ha devuelto un error que nos indica que el vector de datos no es un submúltiplo o múltiplo del número de filas y columnas de la matriz, en otras palabras, la matriz es de 5 renglones por 7 columnas, es decir, tendrá un espacio para ingresar 35 valores y nosotros estamos ingresando solo 31, por lo que R intentará completar la matriz repitiendo el vector hasta completar los 35 valores. Si nosotros no queremos que suceda esto debemos de completar la cantidad de datos, existen varias maneras de solucionar este problema, este es uno de varias de ellas:

```
vector <- 1:31
matrix(c(vector,NA,NA,NA,NA), nrow = 5, ncol=7, byrow=T)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
## [1,]    1    2    3    4    5    6    7
## [2,]    8    9   10   11   12   13   14
## [3,]   15   16   17   18   19   20   21
## [4,]   22   23   24   25   26   27   28
## [5,]   29   30   31   NA   NA   NA   NA
```

Lo que sucede aquí es que al vector le agregamos 4 valores `NA` que significa que no están disponibles y ahora tenemos un vector de longitud 35, la cual entra exactamente en la matriz de 5*7. ### Data Frame

Los dataframes, son una clase parecida a las matrices, sin embargo, poseen determinadas características que las hace más útiles en ciertas ocasiones.

Al igual que los vectores y matrices, existen varias formas de crear data frames, una forma sencilla de crearlos es especificar los datos de las columnas como en el ejemplo siguiente:

```
data.frame(w = 1, x = 1:5, y = LETTERS[1:5], z=runif(5))
```

```
##   w x y         z
## 1 1 1 A 0.86994842
## 2 1 2 B 0.75218814
## 3 1 3 C 0.73615902
## 4 1 4 D 0.06252613
## 5 1 5 E 0.13403621
```

#### 2.2.2.3 Crear un Data frame

#### 2.2.2.4 Trabajar con Data frames

#### 2.2.2.5 Ordenar los datos de un Data frame

#### 2.2.2.6 Extraer datos de un data frame

#### 2.2.2.7 Tratar los datos de un data frame(apply)

#### 2.2.2.8 Problemas comunes

### 2.2.3 Lista

#### 2.2.3.1 Crear una lista

#### 2.2.3.2 Trabajar con listas

#### 2.2.3.3 Ordenar los datos de una lista

#### 2.2.3.4 Extraer datos de una lista

#### 2.2.3.5 Tratar los datos de una lista

#### 2.2.3.6 Problemas comunes

## 2.3 Manejo de datos

### 2.3.1 Salvar datos

En algún momento, tendremos la necesidad de guardar o leer algo en un formato determinado, sea `.csv` o `.Rdata`, veamos unos ejemplos que podrán ser de utilidad.

#### 2.3.1.1 CSV (Comma-separated values)

Un formato muy utilizado en **R** a parte de los `.RData` (que son archivos especiales de **R**), para exportar los datos en dicho formato:

```
df = data.frame(runif(10), runif(10), runif(10))
names(df) = c("dato1", "dato2", "dato3")

write.table(df, file = "dataframe1.csv", sep = ",",
col.names = NA, qmethod = "double")
```

En caso de que no se quisiera exportar el nombre de las filas, basta con modificar un poco el código, como en el siguiente ejemplo:

```
write.table(df, file = "dataframe2.csv", sep = ",",
row.names = FALSE, qmethod = "double")
```

#### 2.3.1.2 .RData

Si tienes objetos que te gustaría guardar como tal para luego procesarlos o simplemente se te hace más sencillo comprenderlo sobre un `.csv`, los comandos serán los siguientes:

```
foo = "bar"
save(foo, file="nombre.RData")
```

### 2.3.2 Leer datos

#### 2.3.2.1 CSV

```
read.table("dataframe1.csv", header = TRUE, sep = ",",
row.names = 1)
```

Y si no se quieren importar los nombres de las filas optaremos por el siguiente comando:

```
read.table("dataframe2.csv", header = TRUE, sep = ",")
```

Aunque, la forma más sencilla de importar un `.csv` es a través del comando `read.csv(...)`, cuya implementación es la siguiente:

```
read.csv(file, header = TRUE, sep = ",", dec = ".", ...)
```

Dónde:

| Argumentos | Significado o uso |
|---|---|
| `file` | Ruta al archivo, en caso de estar en el mismo directorio de trabajo poner solo el nombre del archivo, de otro modo ingresar la ruta completa |
| `header` | Valor lógico para determinar si el archivo incluye encabezados en la primera linea. |

| Argumentos | Significado o uso |
|---|---|
| `sep` | Este campo sirve para especificar el carácter de separación. |
| `dec` | El carácter usado para los puntos decimales |
| `...` | Ver la documentación para argumentos extras |

Un ejemplo seria el siguiente:

```
csv <- read.csv("dataframe1.csv")
```

#### 2.3.2.2 .RData

para volver a cargar los datos:

```
load("nombre.RData")
```

Para llamar el objeto, basta con volver a introducir su variable, que, aunque no haya sido creada antes de usar la función `load()`, esta se encarga de crear el objeto y asignarle el valor que decidimos guardar en un principio, ver 2.3.1.2.

```
foo
```

```
## [1] "bar"
```

##### 2.3.2.2.1 Problemas comunes con .RData

###### 2.3.2.2.1.1 Error al cargar los datos

**R** dice que no los datos que trato de cargar no existen pero yo veo que sí existen en mi carpeta.

###### 2.3.2.2.1.2 ¿Donde están los datos?

###### 2.3.2.2.1.3 ¿Cuales son los datos que cargué?

# Chapter 3

# Introducción a RStudio

# Chapter 4

# Introducción a R-Markdown

# Chapter 5

# Apendice A