R básico para ciencia de datos - Clase 2

Ph D.Stephanie Hereira-Pacheco
CTBC
Universidad Autónoma de Tlaxcala

15 - 02 - 2022

Datos, funciones y operaciones en R - Día 2

- ***Dataframes*: formas de acceso y subconjuntos**
- **Importando datos a R**
- **Funciones básicas en R**

- La forma más común de almacenar un set de datos en R es usando un *dataframe*.
- Los dataframes son estructuras de datos de dos dimensiones que pueden contener datos de diferentes tipos, por lo tanto, son heterogéneas.
- Vamos a trabajar con una data de ejemplo y exploraremos esta *dataframe*, es una data en la que evaluan el efecto de la dosis de vitamina C sobre el crecimiento de los dientes de unos tipos de cerdos. La función `str` es útil para obtener más información sobre la estructura de un objeto:

```
data(ToothGrowth)
str(ToothGrowth)
```

```
## 'data.frame':    60 obs. of  3 variables:
##  $ len : num  4.2 11.5 7.3 5.8 6.4 10 11.2 11.2 5.2 7 ...
##  $ supp: Factor w/ 2 levels "OJ","VC": 2 2 2 2 2 2 2 2 2 2 ...
##  $ dose: num  0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 ...
```

Esto nos dice mucho más sobre el objeto. Vemos que la tabla tiene 60 filas y 3 variables.

Podemos conocer las dimensiones de la tabla con la función `dim`:

```
dim(ToothGrowth)
```

```
## [1] 60  3
```

También, podemos mostrar las primeras seis líneas usando la función `head`:

```
head(ToothGrowth)
```

```
##    len supp dose
## 1  4.2   VC  0.5
## 2 11.5   VC  0.5
## 3  7.3   VC  0.5
## 4  5.8   VC  0.5
## 5  6.4   VC  0.5
## 6 10.0   VC  0.5
```

Dataframes

Para crear un dataframe usamos la función `data.frame()`. Esta función nos pedirá un número de vectores igual al número de columnas que deseemos. Todos los vectores que proporcionemos deben tener el mismo largo. Es decir, un dataframe está compuesto por vectores. Veamos un ejemplo:

```
df <- data.frame(
  "entero" = 1:3,
  "factor" = c("alto", "medio", "bajo"),
  "letras" = as.character(c("a", "b", "c")))
df
```

```
##   entero factor letras
## 1      1   alto      a
## 2      2  medio      b
## 3      3   bajo      c
```

Las funciones `names` y `colnames` nos permiten conocer los nombres de los títulos (headers) o de las columnas.

```
names(df)
```

```
colnames(df)
```

```
## [1] "entero" "factor" "letras"
```

El operador $ y otras formas de acceso

Para tener acceso a las diversas variables o columnas de un *data.frame* utilizamos el operador de acceso $, por ejemplo, si quisieramos tener acceso a la variable 'factor' de la *data.frame* **df** se hace de la siguiente manera:

```
df$factor
```

```
## [1] "alto"  "medio" "bajo"
```

```
class(df$factor)
```

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

```
is.vector(df$factor)
```

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

Cuando usamos el operador $ el tipo de objeto que obtenemos es un vector, en el ejemplo como la columna 'factor'.

Tip: R viene con una muy buena funcionalidad de autocompletar que nos ahorra la molestia de escribir todos los nombres. Escriban `df$` y luego presionen la tecla *tab* en su teclado. Esta funcionalidad y muchas otras características útiles de autocompletar están disponibles en RStudio, esto aplica también para las funciones.

El operador $ y otras formas de acceso

En el caso de las listas también podemos acceder con el operador $, aunque también podemos usar corchetes dobles ([[) así. Por ejemplo declaramos una lista:

```
notas_estudiantes <- list(nombres = c("Ana", "Clara", "Sofy"),
                id_estudiante = c("i1", "i2", "i3"),
                notas = c(10, 9,7))
```

Y queremos extraer los nombres de los estudiantes, entonces hacemos_

```
notas_estudiantes$nombres
```

```
## [1] "Ana"   "Clara" "Sofy"
```

```
notas_estudiantes[["nombres"]]
```

```
## [1] "Ana"   "Clara" "Sofy"
```

Y obtenemos el mismo resultados.

El operador $ y otras formas de acceso

Para el caso de las matrices se puede acceder usando corchetes ([). Si desean la primera fila y la pimera columna, entonces:

```
mat<- matrix(1:10, ncol = 5, nrow = 2)
mat[1,1]
```

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

Para acceder solo a la primera fila y solo a la primera columa usamos las comas, así: s

```
mat[1 ,] #acceder primera fila
```

```
## [1] 1 3 5 7 9
```

```
mat[, 1] #acceder a la primera columna
```

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

```
is.vector(mat[, 1])
```

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

Notese que esto devuelve un vector, no una matriz.

El operador $ y otras formas de acceso

Se pueden crear subconjuntos basados tanto en las filas como en las columnas:

```
mat[1:2 , 2:4]  #en orden de posición es filas primero y luego columnas
```

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

Podemos convertir las matrices en *dataframes* usando la función `as.data.frame`:

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

```
##   V1 V2 V3 V4 V5
## 1  1  3  5  7  9
## 2  2  4  6  8 10
```

Creando subconjuntos o Indexación

En R, podemos obtener subconjuntos de nuestras estructuras de datos, es decir, podemos extraer partes de una estructura de datos (nuestro conjunto).

También podemos usar corchetes individuales ([) para acceder a las filas y las columnas de un *dataframe* y es exactamente igual que lo que se aplicó con las matrices. A esto es lo que llamamos **Subconjuntos** de los *data.frame*. Como las listas de datos que usamos para *notas_estudiantes* tienen las mismas dimensiones entonces podemos coercionarlo a ser una *data.frame*:

```
evaluaciones<- as.data.frame(notas_estudiantes)
```

Y para obtener más de una entrada se puede utilizar un vector de entradas múltiples como índice:

```
evaluaciones[c(1,2)]
```

```
##   nombres id_estudiante
## 1     Ana            i1
## 2   Clara            i2
## 3    Sofy            i3
```

Obtenemos las dos primeras columnas.

Creando subconjuntos.

Las secuencias definidas anteriormente son particularmente útiles si necesitamos acceso, digamos, a los dos primeros elementos:

```
evaluaciones[1:2]
```

```
##   nombres id_estudiante
## 1     Ana            i1
## 2   Clara            i2
## 3    Sofy            i3
```

Ahora bien, si queremos **NO** elegir, por ejemplo, la primera columna o dejarla fuera, entonces usamos el signo '-':

```
evaluaciones[,-1]
```

```
##   id_estudiante notas
## 1            i1    10
## 2            i2     9
## 3            i3     7
```

Creando subconjuntos.

Si los elementos tienen nombres de columna o *headers* también podemos acceder a las entradas utilizando estos nombres:

```
evaluaciones[c("nombres","notas")]
```

```
##   nombres notas
## 1     Ana    10
## 2   Clara     9
## 3    Sofy     7
```

Creando subconjuntos.

Ahora bien, podemos seleccionar datos que tengan características específicas, por ejemplo, todos los valores mayores a cierto número o aquellos que coinciden exactamente con un valor de nuestro interés. Para realizar esta operación haremos uso de índices y operadores lógicos.

Operador	Comparación
$<$	Menor que
$<=$	Menor o igual que
$>$	Mayor que
$>=$	Mayor o igual que
$==$	Exactamente igual que
$! =$	No es igual que
!	No es
$=$	Igual que
&, |	y , ó

Creando subconjuntos.

Por ejemplo en el caso de la tabla de **evaluaciones**, si queremos escoger los valores que sean mayor de 8 en las notas obtenidas:

```
evaluaciones$notas > 8
```

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

Observamos cuales cumplen con la condición si muestran `TRUE`. Ahora para usar este filtro y hacer un subconjunto con el *data.frame*, hacemos:

```
mas_de_8<-evaluaciones[evaluaciones$notas > 8,]
mas_de_8
```

```
##   nombres id_estudiante notas
## 1     Ana            i1    10
## 2   Clara            i2     9
```

Creando subconjuntos.

Si queremos usar más de una condición pero indicando negación:

```
evaluaciones[!(evaluaciones$notas > 8 &
                 evaluaciones$nombres == "Clara"), ]
```

```
##   nombres id_estudiante notas
## 1     Ana            i1    10
## 3    Sofy            i3     7
```

Para escoger un valor que sea exactamente igual a una condición usamos '==':

```
evaluaciones[evaluaciones$nombres == "Sofy",]
```

```
##   nombres id_estudiante notas
## 3    Sofy            i3     7
```

Importando datos

Para el ejemplo que vimos en la clase pasada usamos un dataset que está en el ambiente de R por default, si queremos saber cuales son los datasets que tenemos en nuestro ambiente, podemos usar el comando *data()* y nos desplegará la lista:

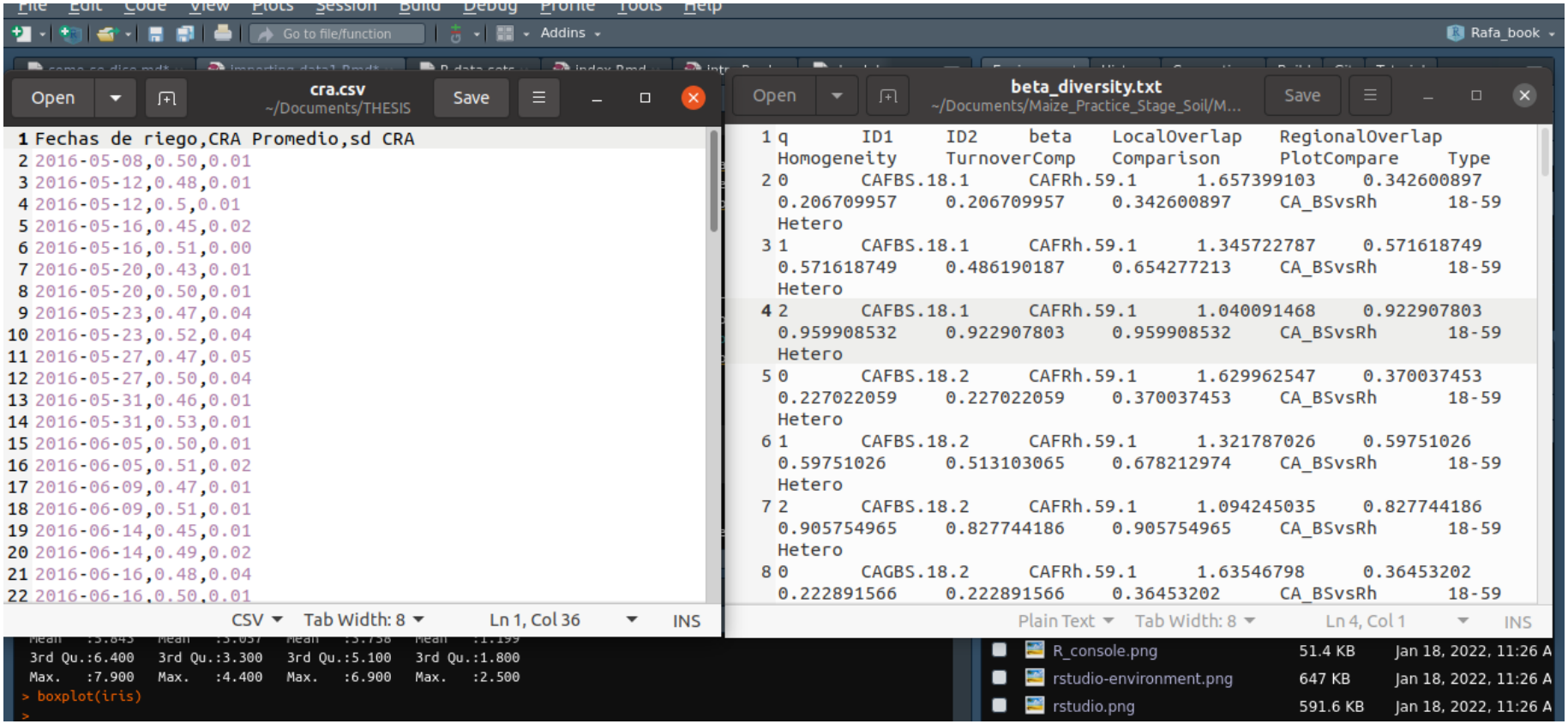
```
data()
```

Si queremos utilizar los datos de nuestro trabajo o usar datos de una base de datos o de un 'dataset' que se encuentre en internet, debemos *Importar* estos datos a nuestra sesión de R. Usualmente tenemos nuestros datos guardados en hojas de cálculo en diferentes formatos con diferentes extensiones, estos son los más populares:

- separados con *coma* o *punto y coma* (`,`,`;`): csv,
- separados con tabulaciones o espacios (*tab,* `\t`) : .txt o .tsv,
- Hojas de cálculo de excel: .xls, son las más usadas.

Importando datos

A continuación muestro una imagen de como se ven un .csv y .txt:



El directorio de trabajo y rutas

Antes de importar nuestros propios archivos, tablas o datos debemos estar seguros en qué directorio nos encontramos, para estar seguros que vamos a importar el archivo deseado a R.

Existen tres opciones para esto:

1. Utilizar **getwd()** y **setwd()**, como lo vimos anteriormente, para establecer y saber en qué directorio nos encontramos y si es el caso, cambiarlo.
2. Poner la ruta completa de nuestro archivo, sin importar donde esté.
3. Utilizar **"Import Dataset"** de nuestro panel de ambiente y ubicar manualmente la ubicación del archivo.

El reto de la primera opción es permitir que las funciones de importación de R sepan dónde buscar el archivo que contiene los datos. La forma más sencilla de hacer esto es tener una copia del archivo en la carpeta donde las funciones de importación buscan por defecto, es decir guardar este archivo en nuestro directorio de trabajo.

Descargando un archivo de la web

Para descargar algún archivo en la web podemos correr el siguiente código:

```
download.file(
  url = "https://archive.ics.uci.edu/ml/machine-learning-databases/
  iris/iris.data",
  destfile = "iris.data")
```

Si observamos nuestros archivos en nuestro directorio de trabajo con el código *list.files()* veremos que se encuentra esta data que hemos descargado.

El código anterior no lee los datos sólo, en este caso, descarga la data. Otra forma de descargarlo y a la vez abrirlo es de la siguiente manera, con el paquete *readr*:

```
iris_dat<-readr::read_csv("https://archive.ics.uci.edu/ml/
                          machine-learning-databases/iris/iris.data")
```

Funciones de importación

Una vez descargado o que se encuentre en nuestro directorio de trabajo, podemos importar los datos con solo una línea de código. Aquí usamos la función `read.csv` o `read.delim` de R base (que viene por default cuando descargamos R).

```
iris_data<- read.csv("iris.data", header = F)
iris_data<- read.delim("iris.data",header = F, sep = ",")
```

Los datos se importan y almacenan en el objeto `iris_dat`.

Los argumentos *header* = *F* y *sep*=*","* son parámetros extra que podemos agregar a la función para indicarle algunas cosas.

Podemos usar la tecla *'tab'* para explorar las demás opciones que podemos utilizar en estas funciones.

También el paquete *readr* tiene otras funciones de importación muy parecidas:

```
library(readr)
iris_data<-read_csv("iris.data",
                    col_names = c("Longitud.sepalo", "Ancho.Sepalo" ,
                    "Longitud.Petalo" ,"Ancho.Petalo" , "Especies"))
```

En esta función usamos el argumento *col_names* para establecer los nombres de las columnas de esta tabla

Funciones de importación

La segunda opción que vimos es utilizar la ruta completa del archivo, por ejemplo:

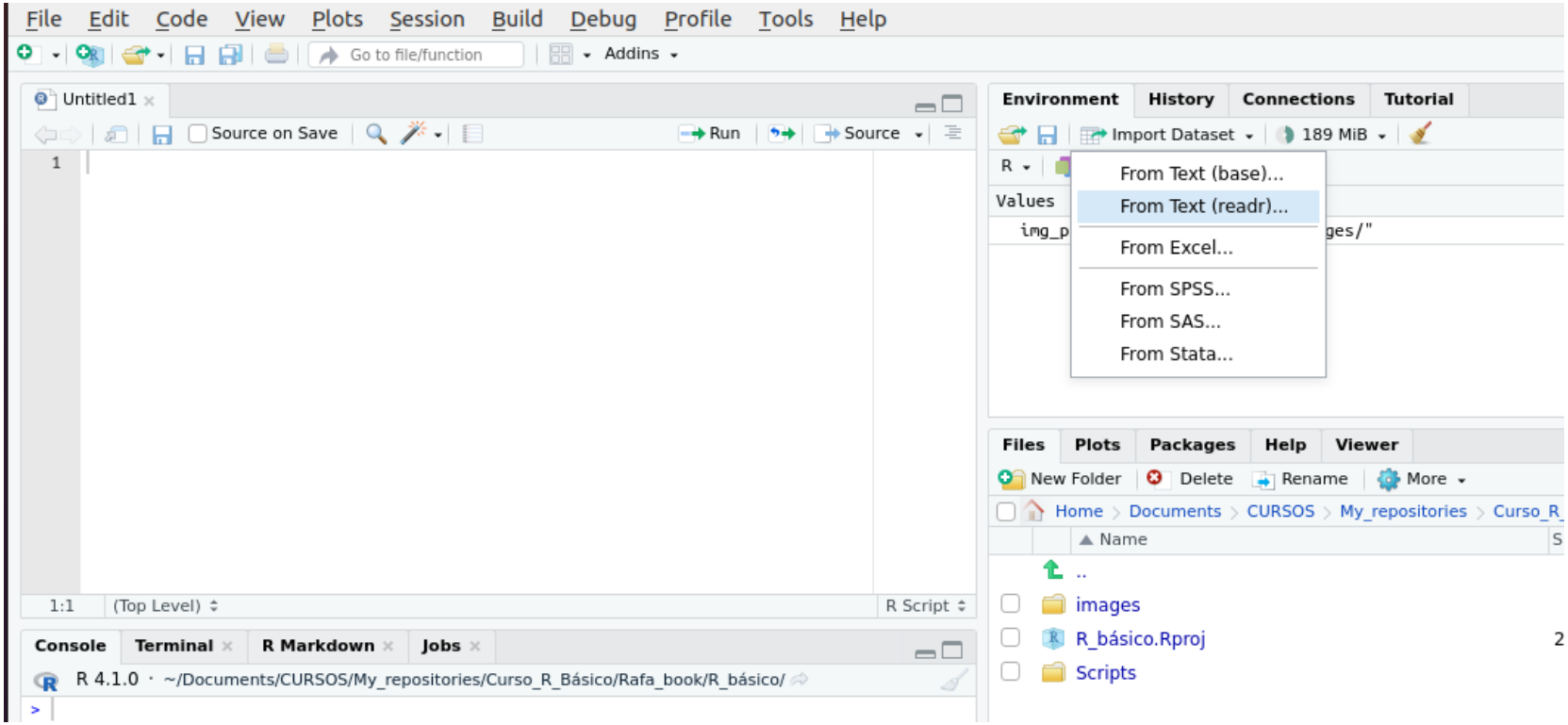
```
data<- read_csv("../Data/penguins_size.csv")
```

```
## Rows: 344 Columns: 7

## -- Column specification --------------------------------------
## Delimiter: ","
## chr (3): species, island, sex
## dbl (4): culmen_length_mm, culmen_depth_mm, flipper_length_mm, body_

##
## i Use `spec()` to retrieve the full column specification for this da
## i Specify the column types or set `show_col_types = FALSE` to quiet
```

En este ejemplo, la data que importamos está ubicada en la carpeta de 'Data' de este proyecto. Y vemos que esta función de *readr* también nos da información del archivo como el tipo y nombre de las variables.

Funciones de importación

La última opción, un poco más fácil para algunos y más interactiva es usar *Import Dataset* del panel de *ambiente*.



En esta opción podemos importar tablas con cualquiera de las funciones que despliega dependiendo del tipo de archivo, yo recomiendo que si es .csv o .txt usar *readr* que es el mismo que usa *tidyverse*, como lo vimos anteriormente. Paquete que veremos más detalladamente junto con *readxl* en la siguiente sección.

El paquete **readr** un paquete de tiyverse, tiene las siguientes funciones para importar archivos con diferentes extensiones:

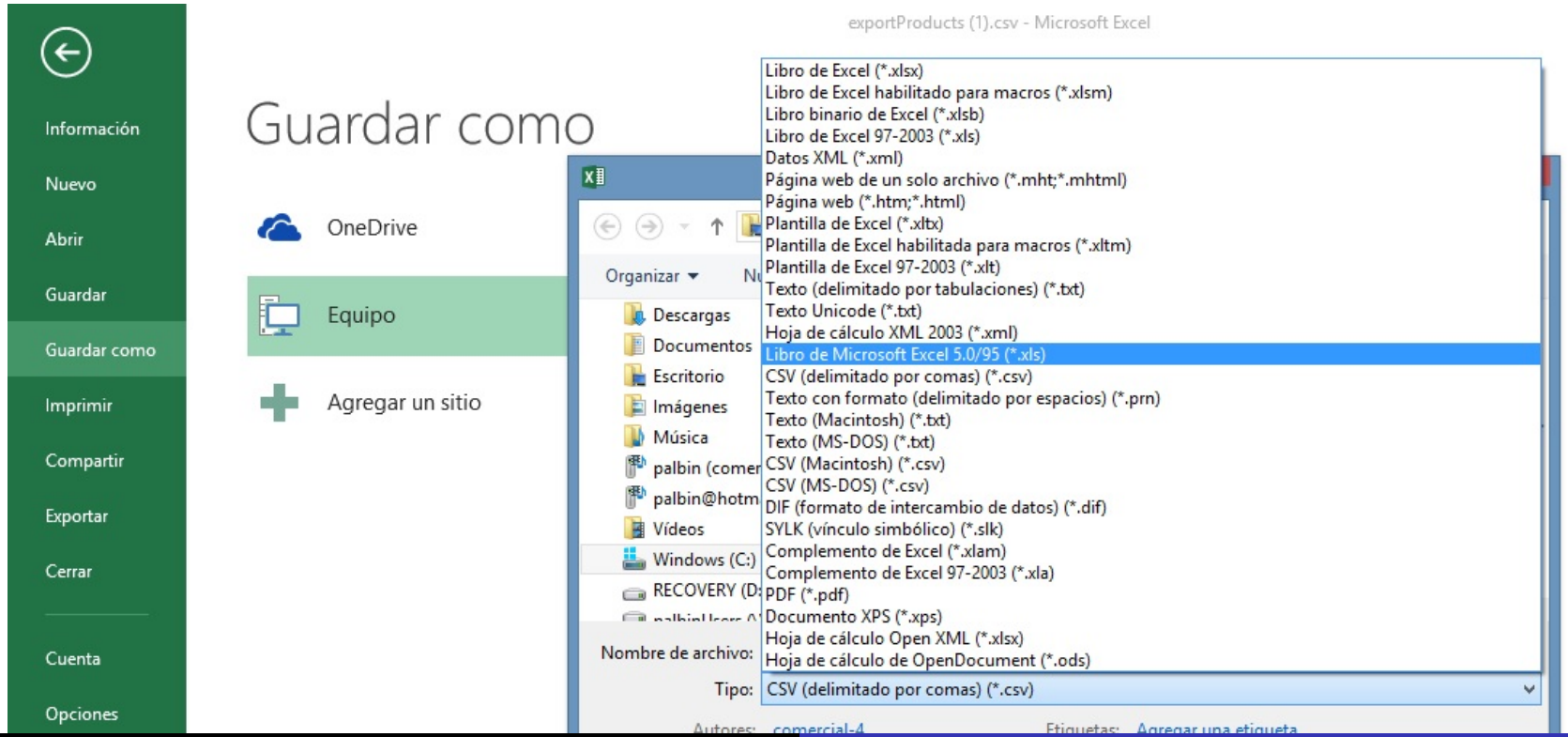
Función	Tipo de archivo	Extensión
read_table	valores separados por espacios en blanco	txt
read_csv	valores separados por comas	csv
read_csv2	valores separados por punto y coma	csv
read_tsv	valores separados delimitados por tab	tsv o txt
read_delim	formato de archivo de texto general, debe definir delimitador	txt, csv o tsv

Este paquete ofrece funciones para leer archivos provenientes de Microsoft Excel:

Función	Formato	Sufijo típico
read_excel	detectar automáticamente el formato	xls, xlsx
read_xls	formato original	xls
read_xlsx	nuevo formato	xlsx

Algunos tips para hojas de cálculo de excel. . .

- Evitar tener diferentes formatos como colores, subrayados, etc.
- Evitar en lo posible celdas vacías y/o poner un dato en cada celda
- Evitar celdas con cálculo o aplicación de fórmulas.
- Tenga en cuenta que nuestros datos que tenemos en hojas de cálculo en excel también podemos guardarlos en otros formatos un poco más fáciles para ser importados como los que ya vimos (.csv y .txt). Esto utilizando la opción *Guardar como* y escogiendo el tipo de formato deseado. En la imagen podemos ver un ejemplo de esto:



Funciones Básicas de R: sort y order

Digamos que queremos clasificar las notas de la mayor a la menor, podemos usar alguna de estas dos funciones:

```
sort(evaluaciones$notas)
```

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

```
order(evaluaciones$notas)
```

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

Funciones Básicas de R: `max` y `which.max`

Si solo estamos interesados en la entrada con el mayor valor, podemos usar `max`:

```
max(evaluaciones$notas)
```

```
## [1] 10
```

y `which.max` nos dice qué valor es el mayor, posicionalmente:

```
which.max(evaluaciones$notas)
```

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

Para el mínimo, podemos usar `min` y `which.min` del mismo modo.

Funciones Básicas de R: `which`

La función `which` nos dice qué entradas de un vector lógico son TRUE. Entonces podemos escribir:

```
ind <- which(evaluaciones$nombres == "Ana")
ind
```

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

```
evaluaciones[ind,]
```

```
##   nombres id_estudiante notas
## 1     Ana            i1    10
```

De esta forma también podemos usarlo para filtrar y hacer subconjuntos.

Funciones Básicas de R: match

La función match nos dice qué índices de un segundo vector coinciden con cada una de las entradas de un primer vector:

```
v1<- c("Uvas", "Peras", "Mandarinas", "Plátanos", "Manzanas")
v2<- c("Uvas","Cerezas", "Mandarinas", "Naranjas", "Manzanas")
match(v1, v2)
```

```
## [1]  1 NA  3 NA  5
```

```
match(c("Peras", "Plátanos"), v1)
```

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

```
ind<-match(c("Peras", "Plátanos"), v1)
v1[ind]
```

```
## [1] "Peras"    "Plátanos"
```

Este filtro puede aplicarse de igual manera a un *dataframe*:

```
ind2<- match(v1, v2)

frutas<- data.frame(persona1=v1,
                    persona2=v2)

frutas[ind,]
```

```
##   persona1 persona2
## 2    Peras  Cerezas
## 4 Plátanos Naranjas
```

```
na.omit(frutas[ind2,])      #na.omit() nos permite quitar los NA's
```

```
##     persona1   persona2
## 1       Uvas       Uvas
## 3 Mandarinas Mandarinas
## 5   Manzanas   Manzanas
```

Funciones Básicas de R: %in%

Si en lugar de un índice queremos un lógico que nos diga si cada elemento de un primer vector está en un segundo vector, podemos usar la función `%in%`. Siguiendo el ejemplo pasado:

```
c("Peras", "Plátanos") %in% frutas$persona1
```

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

Nos dice que los dos elementos que buscamos están presente en el *dataframe*

Avanzado: `match` e `%in%` pueden dar el mismo output usando `which`:

```
match(c("Peras", "Plátanos"), frutas$persona1)
```

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

```
which(frutas$persona1 %in% c("Peras", "Plátanos"))
```

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

La familia de funciones `apply()`

Esta familia de funciones es usada para aplicar una función a cada elemento de una estructura de datos. En particular, es usada para aplicar funciones en matrices, dataframes, arreglos y listas. Para entender más fácilmente el uso de la familia apply, recordemos la vectorización de operaciones. Hay operaciones que si las aplicamos a un vector, son aplicadas a todos sus elementos. La familia apply esta formada por las siguientes funciones:

- **apply()**
- **lapply()**
- **mapply()**
- **sapply()**
- eapply()
- rapply()
- tapply()
- vapply()

Es una familia numerosa y esta variedad de funciones se debe a que varias de ellas tienen aplicaciones sumamente específicas. Las más usadas son las que están en negrita. Repasaremos la función apply pero no nos detendremos mucho porque muchas de estas no están al alcance del presente curso.

apply()

`apply` aplica una función a todos los elementos de una **matriz**.

La estructura de esta función es la siguiente.

```
apply(X, MARGIN, FUN)
```

`apply` tiene tres argumentos:

- `X`: Una matriz o un objeto que pueda coercionarse a una matriz, generalmente, un dataframe.
- `MARGIN`: La dimensión (margen) que agrupará los elementos de la matriz `X`, para aplicarles una función. Son identificadas con números, **1** son filas y **2** son columnnas.
- `FUN`: La función que aplicaremos a la matriz `X` en su dimensión `MARGIN`.

apply()

Si queremos sumar todas las columnas de una matriz, podemos aplicar esta función, para comparar usaremos también la función `ColSums()` que realiza esta misma operación:

```
matriz<- matrix(1:20, ncol = 5, nrow = 4)
matriz
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    5    9   13   17
## [2,]    2    6   10   14   18
## [3,]    3    7   11   15   19
## [4,]    4    8   12   16   20
```

```
apply(X = matriz, MARGIN = 2, FUN = sum)
```

```
## [1] 10 26 42 58 74
```

```
colSums(matriz)
```

```
## [1] 10 26 42 58 74
```

También podemos aplicar múltiples funciones a una matriz:

```
multiples.func <- function(x) {
      c(sum = sum(x), prom = mean(x), max = max(x))}
apply(X = matriz, MARGIN = 2, FUN = multiples.func)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## sum  10.0 26.0 42.0 58.0 74.0
## prom  2.5  6.5 10.5 14.5 18.5
## max   4.0  8.0 12.0 16.0 20.0
```

Estructuras de control

Estas estructuras nos permiten controlar la manera en que se ejecuta nuestro código. Se establecen como condicionales en nuestros código. Por ejemplo, qué condiciones deben cumplirse para realizar una operación o qué debe ocurrir para ejecutar una función.

Las estructuras de control más usadas son:

Estructura de control	Descripción
if, else	Si, de otro modo
while	mientras
for	Para
break	interrumpe
next	siguiente

Conoceremos como se utilizan pero no profundizaremos mucho en ellas.

If, else

If y else se utilizan para crear condiciones, por ejemplo, si cumple esta condición entonces haz esto, de otra manera, haz esto.

Ejemplo:

```
if(10>2) {"Verdadero"
} else {
  "Falso"
}
```

```
## [1] "Verdadero"
```

```
if(10<2) {"Verdadero"
} else {
  "Falso"
}
```

```
## [1] "Falso"
```

If, else

También hay una función que reune estas dos condiciones: `ifelse()` y se usa de igual manera:

```
ifelse((10>2), "Verdadero", "Falso")
```

```
## [1] "Verdadero"
```

Podemos aplicarlo en los *dataframes* usando como ejemplo el dataset anterior:

```
ifelse(evaluaciones$notas>7,  "Aprobado", "Reprobado")
```

```
## [1] "Aprobado"  "Aprobado"  "Reprobado"
```

for

La estructura `for` nos permite ejecutar un bucle (*loop*), realizando una operación para cada elemento de un conjunto de datos.

Ejemplo:

```
un_vector<- 1:10
for(i in un_vector) {
 print(i*2)
}
```

```
## [1] 2
## [1] 4
## [1] 6
## [1] 8
## [1] 10
## [1] 12
## [1] 14
## [1] 16
## [1] 18
## [1] 20
```

Este es un tipo de bucle que ocurre **mientras** una condición es verdadera (TRUE). La operación se realiza hasta que se se llega a cumplir un criterio previamente establecido.

Ejemplo:

```
umbral <- 3
valor <- 0

while(valor < umbral) {
  print("Aún no llegas al umbral")
  valor <- valor + 1
}
```

```
## [1] "Aún no llegas al umbral"
## [1] "Aún no llegas al umbral"
## [1] "Aún no llegas al umbral"
```

Para revisar las demás estructuras, podemos revisar la referencia citada[1]

[1] https://bookdown.org/jboscomendoza/r-principiantes4/estructuras-de-control.html

Extra: Tratando con datos NA

1. Probando qué tenemos NA en nuestros datos:

```
data <- data.frame(x1 = c(NA, 5, 6, 8, 9),
                   x2 = c(2, 4, NA, NA, 1),
                   x3 = c(3,6,7,0,3),
                   x4 = c("Hola", "algo",
                          NA, "Chao", NA))
is.na(data)
is.na(data$x2)
which(is.na(data))
sum(is.na(data))
```

2. Omitir NAs

```
mean(data$x1, na.rm=TRUE)
data[complete.cases(data),]
na.omit(data)
data[!is.na(data$x2),]
```

3. Reemplazar NA's

```
#reemplazar con 0
data[is.na(data)] <- 0
#reemplazar con el promedio o mediana
data$x1[is.na(data$x1)] <- mean(data$x1, na.rm = TRUE)
data$x2[is.na(data$x2)] <- median(data$x2, na.rm = TRUE)
```

Para ver soluciones más complejas podemos buscar los paquetes *Hmisc* (impute), *mice* (mice) y *rpart* (repart).