Curso básico R studio

Ph D.Stephanie Hereira-Pacheco
Ph D.Selene Gómez Acata

Contents

Comenzando con R y RStudio

¿Qué es R?

R es un lenguaje de programación como C o Java o Python pero enfocado principalmente a la estadística. No fue creado por ingenieros de software para el desarrollo de software, sino por estadísticos como un ambiente interactivo para el análisis de datos. Otros programas como los que mencioné anteriormente sí están más enfocados en el desarrollo de programas aunque también pueden usarse para hacer cálculos estadísticos. Cuando instalamos R en nuestra computadora en realidad lo que estamos instalando es el entorno computacional y para que podamos hacer uso de ese entorno necesitamos conocer la manera de escribir que el software pueda interpretar y ejecutar las instrucciones que le damos. Eso es lo que aprenderemos a hacer en este curso.

Como en otros lenguajes de programación, en R pueden guardar su trabajo como una secuencia de comandos o instrucciones, conocida como un *script*, que se pueden ejecutar fácilmente en cualquier momento, los podemos guardar y nos servirań siempre.

Un poco de historia. . . [1]

R proviene del lenguaje S, creado en los Laboratorios Bell (Estados Unidos). Los mismos que inventaron el transistor, el láser, el sistema operativo Unix y algunas otras cosas más.

Ross Ihaka y Robert Gentleman, de la Universidad de Auckland de Nueva Zelanda, decidieron crear una implementación abierta y gratuita de S. Este trabajo, que culminaría en la creación de R inició en 1992 y no

[1] https://bookdown.org/jboscomendoza/r-principiantes4/un-poco-de-historia.html

fue hasta el 2000 que se obtuvo una versión final estable.

Hoy día, el mantenimiento y desarrollo de R es realizado por el R Development Core Team, un equipo de especialistas en ciencias computacionales y estadística provenientes de diferentes instituciones y lugares alrededor del mundo.

R posee una Licencia Pública General de GNU, esto, para que pueda ser distribuido de manera gratuita, por lo que es software libre y de código abierto. Esta licencia también te permite usar R para los fines que desees, sin limitaciones, no importando si son personales, académicos o comerciales.

¿Por qué usar R?[2]

1. R es gratuito y de código abierto[3].
2. Se ejecuta en todas las plataformas principales: Windows, Mac Os, UNIX/Linux.
3. Los *scripts* y los objetos de datos se pueden compartir sin problemas entre plataformas.
4. Existe una comunidad grande, creciente y activa de usuarios de R y, como resultado, hay numerosos recursos para aprender y hacer preguntas[4] [5].
5. Es fácil para otras personas contribuir con complementos (*add-ons* en inglés) o paquetes que les permiten a los desarrolladores compartir implementaciones de software de nuevas metodologías de ciencia de datos. Esto les da a los usuarios de R acceso temprano a los métodos y herramientas más recientes que se desarrollan para una amplia variedad de disciplinas, incluyendo la ecología, la biología molecular, las ciencias sociales y la geografía, entre otros campos.

RStudio

RStudio será nuestra plataforma para los proyectos usados con el lenguaje R. Nos provee un editor visual e interactivo para crear y editar nuestros *scripts*, además de otras herramientas útiles que iremos viendo con el pasar de los temas.

Paneles

Rstudio posee 4 paneles principales:

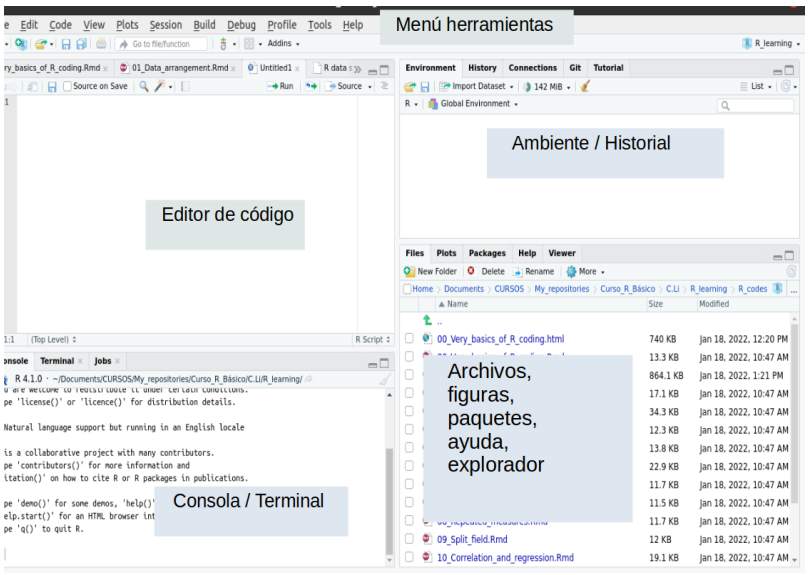
1. El panel izquierdo superior aparece nuestro editor de códigos. Donde ponemos los scripts que vamos a correr.

2. A la derecha, el panel superior incluye pestañas como *Environment* y *History*, que son el ambiente y el historial, aquí podremos observar los objetos que vayamos declarando y data que subamos, además del historial de scripts.

3. En el panel inferior izquierdo nos aparece nuestra consola de R que es donde se corren los códigos.

4. En el panel inferior derecho se muestran cinco pestañas: *File*, *Plots*, *Packages*, *Help* y *Viewer*. Pueden hacer clic en cada pestaña para moverse por las diferentes opciones. Pero a grandes rasgos, en file vemos donde nos encontramos situados y los archivos que hay en nuestro directorio. En plots se visualizan las imágenes que generamos, en packages los paquetes que poseemos instalados y cargados, en Help cuando necesitamos información extra de nuestros paquetes y en Viewer exploramos scripts de rmarkdown.

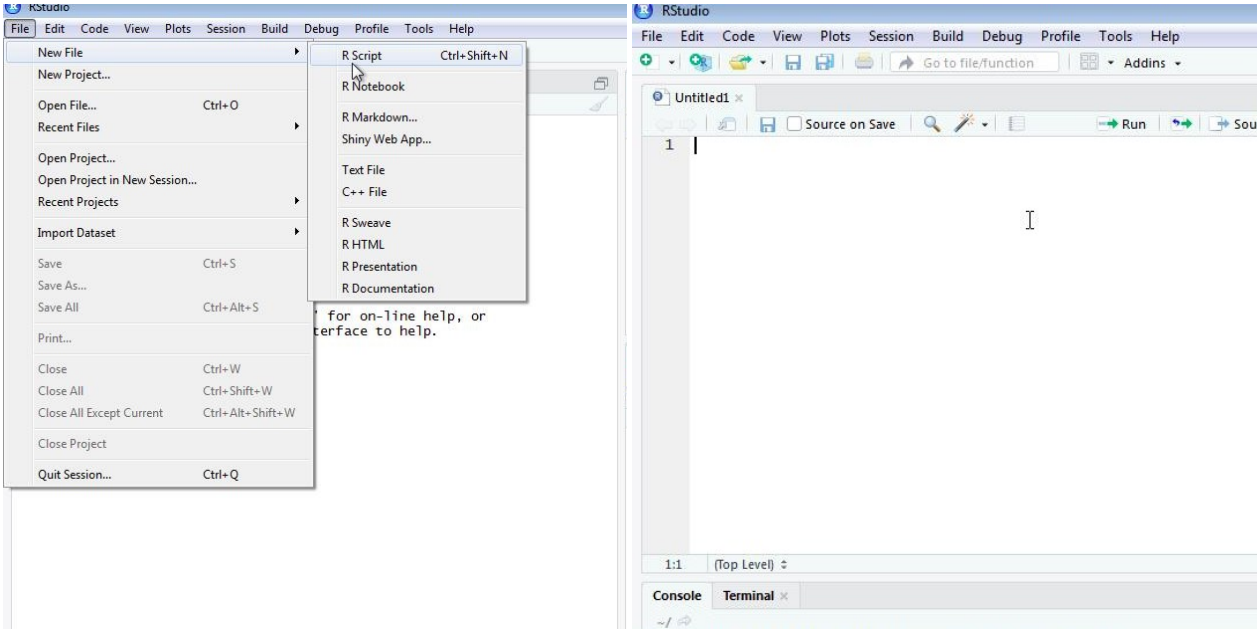
[2] https://rafalab.github.io/dslibro/getting-started.html
[3] https://opensource.org/history
[4] https://stats.stackexchange.com/questions/138/free-resources-for-learning-r
[5] https://www.r-project.org/help.html



Scripts

Una de las grandes ventajas de R y Rstudio es que se pueden guardar los diversos códigos e instrucciones, los famosos conocidos como *scripts*, que entonces se pueden editar y guardar con un editor de texto.

Para iniciar un nuevo *script*, hagan clic en *Archivo*, entonces *Nuevo Archivo* y luego *R Script*. Esto inicia un nuevo panel a la izquierda y es aquí donde pueden comenzar a escribir su *script*.



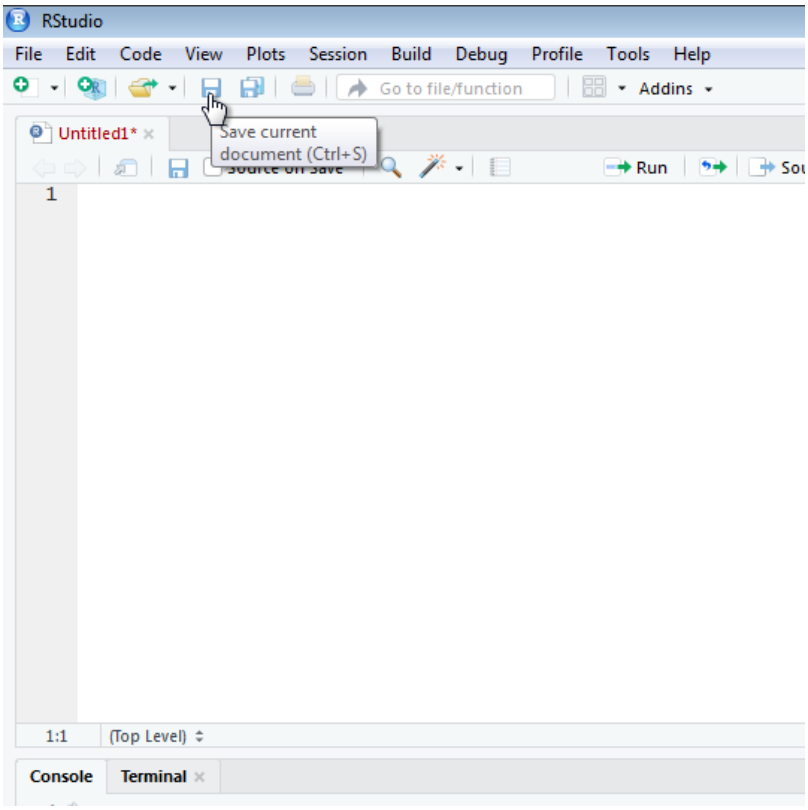
Podemos abrir y ejecutar scripts en R usando la función source(), dandole como argumento la ruta del archivo .R en nuestra computadora, entre comillas o yendo a *File*, entonces *Open File* y luego buscando en las carpetas donde tengas el script.

Por ejemplo.

```
source("C:/mi_script.R")
```

Cómo ejecutar comandos mientras editan *scripts*

Empezamos por abrir un nuevo *script* y luego nombramos el *script*. Podemos hacer esto a través del editor guardando el nuevo *script* actual sin nombre. Al guardar el script por primera vez use un nombre descriptivo, con letras minúsculas, sin espacios, preferiblemente con "_" para separar palabras, evitar guiones "-" para separar las palabras. Llamaremos a este *script*: *mi_script.R*.



Ahora podemos editar nuestro primer *script*.

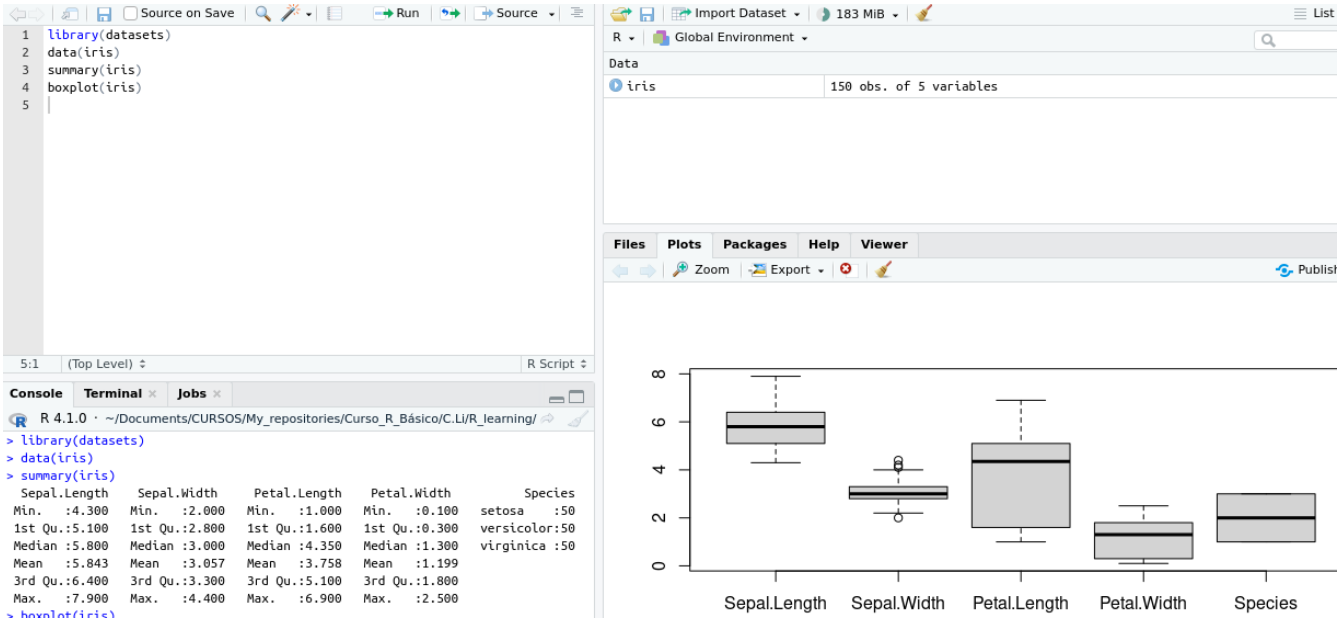
Las primeras líneas de código son títulos o comentarios, para hacerlo siempre debemos poner el símbolo *"#"* para indicar que no es un código y luego cargamos los paquetes y datos que vamos a utilizar. Para esta parte, luego veremos otra sección donde profundizaremos mejor en esto de cargar datos y paquetes.

Ahora podemos continuar escribiendo código. Como ejemplo, trabajaremos con *"iris"* dataset. Cargamos la librería *"datasets"* y cargamos la data, luego veremos el resumen de los datos y por último graficaremos un boxplot. Para hacer esto, escribimos cada línea de código y luego hacemos click en el botón *Run* en la parte derecha superior del panel de edición. Para ejecutar una línea pueden usar Control+Enter en Windows y Linux y Command+Return en Mac.

Estas son las líneas del código:

```
library(datasets)
data(iris)
summary(iris)
boxplot(iris)
```

Y así luce al correrlo:



Tan pronto se corra el código, aparece en la consola y el gráfico aparece en en panel de *"Plots"* , este panel permite hacer click hacia delante o hacia atrás en diferentes gráficos, hacer zoom en el gráfico o guardar los gráficos como archivos.

Directorio de trabajo

El *directorio de trabajo* es el lugar en nuestra computadora en el que se encuentran los archivos con los que estamos trabajando en R. Este es el lugar donde R busca los archivos para importarlos y al que serán exportados o guardados, a menos que se indique otra cosa.

Para saber donde está ubicado tu directorio de trabajo, puedes poner el código:

```
getwd()
```

Y para establecer o cambiar este directorio de trabajo, correr el siguiente código:

```
setwd("/home/steph/Desktop/")
```

Por ejemplo en este caso estoy estableciendo mi directorio de trabajo en la carpeta "Escritorio". También si se quiere conocer el contenido de tu directorio, como archivos o directorios puedes usar los siguientes códigos:

```
list.files()
list.dirs()
```

Sesión

Los objetos y funciones de R son almacenados en la memoria de nuestra computadora. Cuando ejecutamos R, ya estamos creando una instancia del entorno computacional de este lenguaje de programación, cada instancia es una *sesión*.

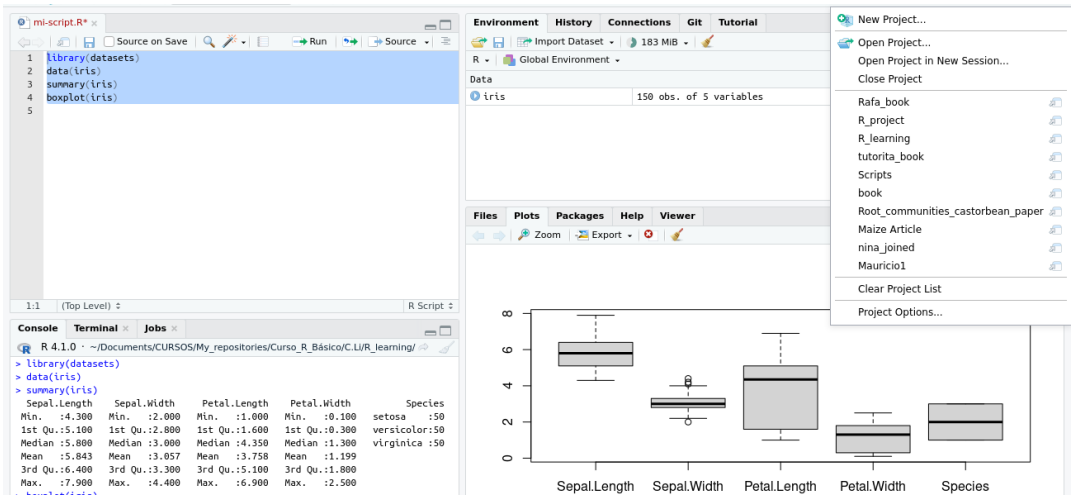
Todos los objetos, tablas, gráficas y funciones creadas en una sesión, permanecen sólo en ella, no son compartidos entre sesiones. Si se quiere guardar toda esta data generada en la sesión antes de cerrar el R hay que indicar que se guarde, sino, aunque se guarden los scripts, los objetos (que algunos pueden ser muy pesados y ocupar memoria) no se guardarán. Lo cual es una opción si no se tiene mucho espacio en el disco.

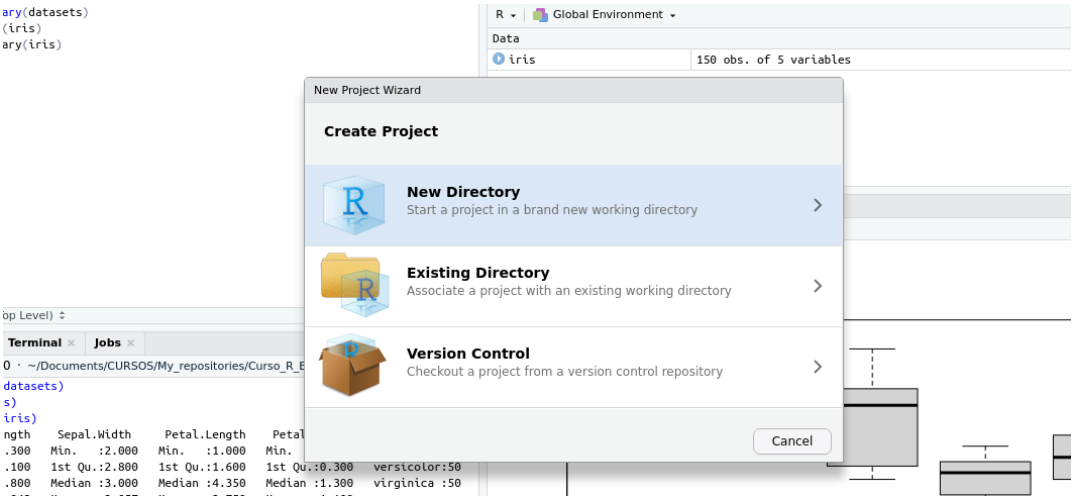
Es posible tener más de una sesión de R activa en la misma computadora. Esto se guarda en un archivo con extensión **.Rdata* en tu directorio de trabajo.

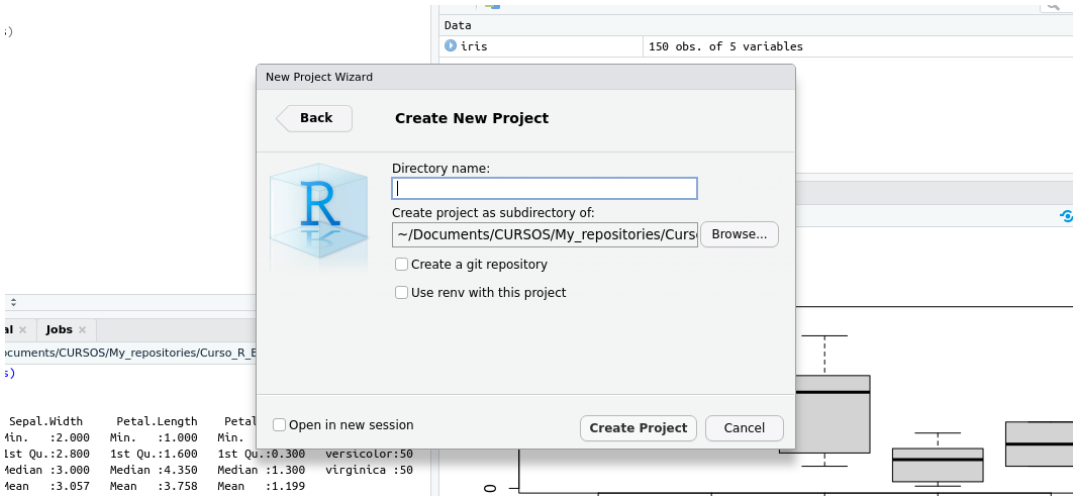
Con la función *ls()* conoceremos una lista con los nombres de todo lo guardado en la sesión. En las siguientes imágenes ilustro como crear un nuevo proyecto:

Proyecto

Un proyecto de R (extensión .Rproj) identifica todos los archivos y contenido asociado con él. Ayuda a organizar tu trabajo y así cada curso, artículo o trabajo diferente puede tener un proyecto diferente. Al crear un proyecto todos los ficheros quedan vinculados directamente a él.







Instalación de paquetes de R

Dentro de las ventajas de R está que muchos desarrolladores y programadores elaboran constantemente complementos, aplicaciones o en palabras propias *paquetes* que nos permiten usar en acceso libre y que tienen muchas funcionalidades. Actualmente hay muchos disponibles en CRAN (Comprehensive R Archive Network) que es una red de servidores alrededor del mundo que almacena versiones actualizadas del código de R y su documentación. También hay muchos paquetes desarrollados publicados en GitHub y en Bioconductor. Por ejemplo para instalar el paquete tidyverse, que es de gran utilidad y que veremos en sesiones posteriores, usamos el código:

```
install.packages("tidyverse")
```

En RStudio pueden navegar a la pestaña *Pakages* y seleccionar *Install*. Luego, escribir el paquete que queremos siempre y cuando esté en CRAN. Para cargar una librería como lo vimos anteriormente se usa la función, `library()`:

```
library(tidyverse)
```

Una vez que se instalan los paquetes, no deben instalarlo de nuevo, sin embargo, cada vez que cerramos sesión, reiniciamos sesión o abrimos un nuevo proyecto o sesión tenemos que volver a cargarlos.

Debemos tener en cuenta que la instalación de **tidyverse** instala varios paquetes. Esto ocurre comúnmente cuando un paquete tiene *dependencias*, es decir usa funciones de otros paquetes. Cuando cargan un paquete usando `library`, también cargan sus dependencias.

Hay paquetes que no se encuentra en CRAN o que si queremos su versión en desarrollo, se necesitan de otros paquetes para ser instalados, por ejemplo, si queremos instalar la versión en desarrollo del paquete *"rmarkdown"*, que se encuentra en github, se utiliza el paquete devtools:

```
devtools::install_github('rstudio/rmarkdown')
```

Los dos puntos *"::"* se utilizan para denotar que llamamos la función de un paquete pero sin llamarla permanentemente en nuestra sesión.

Tipos de objetos en R

La información que manipulamos en R se estructura en forma de objetos y los podemos ver almacenados en el panel del ambiente de trabajo o *Enviroment*. Los objetos pueden ser:

- Números escalares o letras
- Vectores y matrices
- Dataframes, tablas y listas

Más adelante detallaremos este tipo de objetos o datos en R. Aquí unos ejemplos:

```
a <- 1                                                    #escalar
letra <- "a"                                              #caracter o letra
b <- c(1,2,3)                                             #vector
c<- matrix(1:10)                                          #matriz
d<- data.frame(Especie=c("A", "B"), Longitud=c(c(1,2)))   #dataframe o tabla
e<- list(c(1:20), c(1:10))                                #lista
```

Una ventaja de los lenguajes de programación es poder definir variables y escribir expresiones como estas, como se hace en las matemáticas y así almacenar los valores para su uso posterior. Usamos `<-` para asignar valores a las variables. También podemos asignar valores usando `=` en lugar de `<-`, pero recomendamos no usar `=`. Esto, debido a que en R el signo `=` implica igualdad en términos lógicos y no asignación y esto puede evitarnos confusiones en el futuro.

Para ver el valor almacenado en una variable, simplemente le pedimos a R que evalúe `a` y R nos muestra el valor almacenado:

```
a
```

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

Una forma explícita de pedirle a R que nos muestre el valor almacenado en `a` es usar `print` así:

```
print(a)
```

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

Otra forma de examinar los objetos es buscarlos en el *Enviorment* o ambiente de trabajo y visualizarlos desde allí. Deberíamos ver `a`, `b` y las que ya hemos declarado en el ambiente. Si intentamos imprimir o visualizar el valor de un objeto que no está definido en el ambiente se recibirá un mensaje de error. Por ejemplo, si escriben `f`, verán lo siguiente: `Error: object 'f' not found`.

Algunos tips para asignar variables u objetos en R...

- Los nombres de variables tienen que comenzar con una letra, no pueden contener espacios y no deben ser variables predefinidas en R (como funciones o argumentos de funciones). Por ejemplo, no nombren una de sus variables `install.packages` escribiendo algo como: `install.packages <- 2`. Esto reescribe la función o causa confusiones para R.
- Que tus nombres sean descriptivos y/o significativos para lo que se está almacenado, usar solo minúsculas y usar guiones bajos *(_)* como sustituto de espacios, evitar caracteres especiales (- ; . @ ?) .

Guardar los espacios de trabajo y exportar objetos de R

Los objetos evaluados permanecen en el espacio de trabajo hasta que finalicen sus sesiones sin guardar. Pero los espacios de trabajo también se pueden guardar para su uso posterior. De hecho, al salir de R, el programa les pregunta si desean guardar su espacio de trabajo. Si lo guardan, la próxima vez que inicien R, el programa restaurará el espacio de trabajo con todos los objetos en él.

Sin embargo, no se recomienda guardar el espacio de trabajo porque sino se trabaja con diferentes proyectos, será más difícil darle seguimiento de lo que guardan y ocupara mucho espacio en la memoria de su disco. En cambio, se recomienda en gran medida realizar un proyecto por trabajo o tarea y así tener un espacio de trabajo para cada uno. Decidiendo o no si guardar las sesiones y no entrar en confusiones entre las diferentes sesiones.

Ahora bien R ya posee funciones de exporte, tales como:

```
library(readr)
write_tsv(d, "data.tsv")
write.table(d, "data.txt", sep = "\t")
write_csv(d, "data.csv")
```

Estas funciones guardan tus tablas u objetos con formatos de texto. También hay una forma de guardar objetos de R sin declararlos como texto sino que se queden con la identidad de R:

```
saveRDS(d, "data.RDS")
```

Y para abrir o cargar este tipo de objetos con extensión .RDS usamos la siguiente función:

```
readRDS("data.RDS")
```

Funciones[6]

Una vez que definidos los objetos o las variables, si queremos continuar con el análisis de datos generalmente se usan una serie de funciones específicas que se aplican a las variables o datos. R incluye muchas funciones por *default* o preestablecidas y otras pueden extraerse al cargar los diversos paquetes. A lo largo de este curso ya hemos usado varias funciones tales como las funciones para exportar datos como *write_csv()* o para cargar paquetes tales como *library()*, entre otras. Hay otras funciones más sencillas como por ejemplo la función `log` o `sqrt` con las que podemos obtener el logaritmo o la raíz cuadrada. ***(Nótese que las funciones en su gran mayoría están definidas en inglés)***.

Muchas otras funciones vienen establecidas en otros paquetes como fue el caso que vimos de *read_csv()* del paquete *readr*. La sintaxis o el lenguaje de R nos indica que necesitas usar paréntesis para evaluar funciones como hemos visto en los ejemplos anteriores que hemos usado funciones. Algunas funciones como *ls()* no requieren argumentos sino que nos da la información que ocupamos sin evaluar nada dentro de los paréntesis. Sin embargo, en otras funciones sí requieren uno o más argumentos. A continuación se muestra un ejemplo de cómo asignamos un objeto al argumento de la función `log`. Recuerden que anteriormente definimos `a` como 1:

```
log(a)
```

```
## [1] 0
```

Podemos explorar funciones en el panel inferior derecho en la pestaña de *Packages* o *Paquetes* podemos explorar los paquetes y al hacer click en alguno nos despliega las funciones que posee y su documento de ayuda también. Se puede saber cuáles son los argumentos de la función o lo que la función espera o las opciones que tiene, con el comando *help* o también anteponiendo un signo *?* antes de la función. También mientras escribimos la función si usamos la tecla *tab* nos indica también que argumentos espera esta función. Pueden averiguar lo que la función espera y lo que hace revisando unos manuales muy útiles incluidos en R. Pueden obtener ayuda utilizando la función `help` así:

```
help("log")
?log
```

La página de ayuda les mostrará qué argumentos espera la función. Por ejemplo, `log` necesita `x` y `base` para correr. Sin embargo, algunos argumentos son obligatorios y otros son opcionales. Pueden determinar cuáles son opcionales notando en el documento de ayuda cuáles valores predeterminados se asignan con `=`. Definir estos es opcional. Por ejemplo, la base de la función `log` por defecto es `base = exp(1)` que hace `log` el logaritmo natural por defecto.

Para echar un vistazo rápido a los argumentos sin abrir el sistema de ayuda, pueden escribir:

```
args(log)
```

[6] https://rafalab.github.io/dslibro/r-basics.html#

```
## function (x, base = exp(1))
## NULL
```

Pueden cambiar los valores predeterminados simplemente asignando otro objeto:

```
log(x = 8, base = 2)
```

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

El código anterior funciona, pero también podemos ahorrarnos un poco de escritura: si no usan un nombre de argumento, R supone que están ingresando argumentos en el orden en que se muestran en la página de ayuda o por `args`. Entonces, al no usar los nombres, R supone que los argumentos son `x` seguido por `base`:

```
log(8,2)
```

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

Si se usan los nombres de los argumentos, podemos incluirlos en el orden en que queramos:

```
log(base = 2, x = 8)
```

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

Para especificar argumentos, debemos usar `=` y no `<-`. Aquí si aplica la igualdad, en el caso de las funciones.

Hay algunas excepciones a la regla de que las funciones necesitan los paréntesis para ser evaluadas. Entre estas, las más utilizados son los operadores aritméticos y relacionales. Por ejemplo:

```
a + a
```

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

```
a - 2
```

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

```
1 * pi
```

```
## [1] 3.141593
```

```
2 / 3
```

```
## [1] 0.6666667
```

```
4 ^ a
```

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

También es posible declarar una función que a lo mejor no esté definida en R. Por ejemplo, la función *mean()* en Rbase nos da el promedio de un conjunto de datos, pero si quisieramos definirla en caso de no conocerla o hacerlo a nuestra manera, sería algo así:

```
average<- function(x){sum(x)/length(x)}
x<- 1:100
average(x)
```

```
## [1] 50.5
```

```
mean(x)
```

```
## [1] 50.5
```

Al compararlas nos arrojan el mismo valor. Lo importante de este punto es que hemos declarado una nueva función, declarando primero la variable (o variables, dependiendo el caso) y luego las operaciones y asignandolos a un objeto de R:

```
nombre <- function(argumentos) {
  operaciones}
```

Para trabajar con R resulta importante conocer los principales tipos de objetos y sus propiedades básicas.

Tipos de datos

Como vimos en el ejemplo anterior, los objetos en R pueden ser de varios tipos. Por ejemplo, necesitamos distinguir números de los caracteres. La función `class` nos ayuda a determinar qué tipo de objeto tenemos:

```
a <- 5
class(a)
```

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

Tipo	Nombre en inglés/en R	Ejemplo
Numérico	numeric	5.1
Entero	integer	4
Real	double/float	3.4
Cadena de texto, letra	character	"a"
Factor	factor	Bajo
Lógico	logic	TRUE, FALSE
Perdido/Omitido	NA	NA
Vacío	null	NULL

Existen otro tipos de datos más complejos que no están en el alcance del presente curso, por ejemplo: números complejos, fechas, entre otros.

Para tener en cuenta...

- El tipo *character* representa texto y es fácil reconocerlo porque un dato siempre esta rodeado de comillas, simples o dobles. Este es el tipo de datos más flexible de R, pues una cadena de texto puede contener letras, números, espacios, signos de puntuación y símbolos especiales.
- Un factor es un tipo de datos específico a R. Puede ser descrito como un dato numérico representado por una etiqueta. Por último, cada una de las etiquetas o valores que puedes asumir un factor se conoce como **nivel** o **level**. los niveles tienen un orden diferente al orden de aparición en el factor. En R, por defecto, los niveles se ordenan alfabéticamente. **Advertencia**: Los factores pueden causar confusión ya que a veces se comportan como caracteres y otras veces no. Como resultado, estos son una fuente común de errores. A veces las funciones necesitan a fuerza que se ocupe un vector y a veces lo contrario.
- La diferencia entre las dos es que un dato `NULL` aparece sólo cuando R intenta recuperar un dato y no encuentra nada, mientras que `NA` es usado para representar explícitamente datos perdidos, omitidos o que por alguna razón son faltantes. `NA` además puede aparecer como resultado de una operación realizada, pero no tuvo éxito en su ejecución.

Coerción de datos...

En R, los datos pueden ser coercionados, es decir, forzados, para transformarlos de un tipo a otro.

Función de coerción	Tipo
as.integer()	Entero
as.numeric()	Numérico
as.character()	Caracter

Función de coerción	Tipo
as.factor()	Factor
as.logical()	Lógico

Veamos algunos ejemplos:

```
a<- 5
as.character(a)
```

```
## [1] "5"
```

```
as.factor("medio")
```

```
## [1] medio
## Levels: medio
```

Tipos de estructura de los datos

Los datos se estructuran de diferentes formas dependiendo de su propósito, en todo caso, la función *class()* también nos puede dar información sobre los tipos de estructuras de datos.

Vectores

Los vectores son colecciones de uno o más datos del mismo tipo. Por ejemplo, si tenemos un vector con datos numéricos tenemos un vector de tipo numérico. No es posible mezclar datos de tipos diferentes dentro de ellos. Por ejemplo, un vector de colores puede ser:

```
colores<- c("red", "black", "blue")
is.vector(colores)
```

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

```
class(colores)
```

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

Al usar las funcion `is.vector()` corroboramos que efectivamente es un vector al darnos `TRUE` pero al pedirle que nos indique el tipo con la función `class` nos dice que es un "character" es decir que es un vector de una cadena de texto.

Existen algunas operaciones al aplicarlas a un vector, se aplican a cada uno de sus elementos. A este proceso le llamamos **vectorización**. Las operaciones aritméticas y relacionales pueden vectorizarse. Si las aplicamos a un vector, la operación se realizará para cada uno de los elementos que contiene.

```
un_vector <- c(1:10)
un_vector*10
```

```
##  [1]  10  20  30  40  50  60  70  80  90 100
```

```
un_vector+1
```

```
##  [1]  2  3  4  5  6  7  8  9 10 11
```

```
un_vector + un_vector
```

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

Matrices y arreglos

Las matrices y arreglos no son más que vectores multidimensionales, es decir un conjunto de vectores. Al igual que un vector deben contener un sólo tipo de datos. En sentido estricto, una arreglo es una matrix pero con `n` dimensiones, mientras que las matrices tienen solo dos dimensiones. Las matrices y los arreglos suelen ser usados de manera regular en matemáticas y estadística, por ser sencillas y contener solo un tipo de datos (usualmente de tipo numérico). En general, es preferible usar listas en lugar de arrays, una estructura de datos que además tienen ciertas ventajas que se revisará más adelante. En R, podemos usar el símbolo `:` para indicar una secuencia de números que tiene un principio y fin, por ejemplo:

```
vect<- 1:20
```

Este es un vector con números que va desde el 1 al 20. Pero para hacerlo matriz hacemos:

```
matr<- matrix(1:20)
```

O para dividirlo en varias renglones y columnas:

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

```
## [1] 5 4
```

```
matri
```

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

Con la función *dim()* podemos saber cuales son las dimensiones (es decir, largo y ancho) de nuestra matriz. Las operaciones aritméticas también son vectorizadas al aplicarlas a una matriz. La operación es aplicada a cada uno de los elementos de la matriz al igual que los vectores.

```
matri*2
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    2   12   22   32
## [2,]    4   14   24   34
## [3,]    6   16   26   36
## [4,]    8   18   28   38
## [5,]   10   20   30   40
```

Algo más que podemos hacer es tranponer una matriz para rotarla 90°.

```
t(matri)
```

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

Listas

Las listas, al igual que los vectores, son estructuras de datos unidimensionales, sólo tienen largo, pero a diferencia de los vectores cada uno de sus elementos puede ser de diferente tipo o incluso de diferente clase, por lo que son estructuras heterogéneas. Podemos tener listas que contengan escalares, vectores, matrices, data frames u otras listas. Para crear una lista usamos la función list(), que nos pedirá los elementos que

deseamos incluir en nuestra lista. Para esta estructura, no importan las dimensiones o largo de los elementos que queramos incluir en ella. Al igual que con un data frame, tenemos la opción de poner nombre a cada elemento de una lista.

```
un_vector <- 1:20
una_matriz <- matrix(1:4, nrow = 5)
```

```
## Warning in matrix(1:4, nrow = 5): data length [4] is not a sub-multiple or
## multiple of the number of rows [5]
```

```
una_df     <- data.frame("numeros" = 1:3, "letras" = c("a", "b", "c"))

una_lista <- list("vector" = un_vector, "matriz" = una_matriz, "df" = una_df)

una_lista
```

```
## $vector
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
##
## $matriz
##      [,1]
## [1,]    1
## [2,]    2
## [3,]    3
## [4,]    4
## [5,]    1
##
## $df
##   numeros letras
## 1       1      a
## 2       2      b
## 3       3      c
```

No es posible vectorizar operaciones aritméticas usando listas. Al intentarlo nos es devuelto un error.

Finalmente, en caso de que sea necesario utilizar funciones que requieran a fuerza una matrix o un *data frame* también aplica la coerción entre estas dos estructuras de datos:

```
coerción_df<- as.data.frame(una_matriz)
coerción_mat<- as.matrix(una_df)

class(coerción_df); class(coerción_mat)
```

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

```
## [1] "matrix" "array"
```

Esta coerción es muy útil por ejemplo al utilizar la función `t()` que transponer una *data frame*:

```
df_transpuesta<- t(una_df)
class(df_transpuesta)
```

```
## [1] "matrix" "array"
```

Como vemos la función `t()` cambia la estructura de los datos, para evitar esto, coercionamos esta salida.

```
df_transpuesta<- as.data.frame(t(una_df))
class(df_transpuesta)
```

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