Introducción a R y Rstudio

Dra. Stephanie Hereira Pacheco

CICB, UATx

23-02-2024

Introducción a R y Rstudio

- **¿Qué es R y por qué usarlo?**
- **R studio y paneles**
- **Configuración del ambiente de trabajo y Scripts**
- **Instalación y carga de paquetes**
- **Tipos de objetos, tipos de datos y estructuras de datos**
- **Funciones, importación de datos y subconjuntos**
- **Gráficos básicos en R**
- **Tidyverse y ggplot2**

¿Qué es R?

- R es un lenguaje de programación enfocado principalmente a la estadística.
- Fue creado por estadísticos como un ambiente interactivo para el análisis de datos.
- 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 de manera que el software pueda interpretar y ejecutar las instrucciones que le damos.
- 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.

Un poco de historia...

- 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 entre otros.
- Este trabajo, que culminaría en la creación de R, inició en 1992 y no 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 para que pueda ser distribuido de manera gratuita.

¿Por qué usar R?

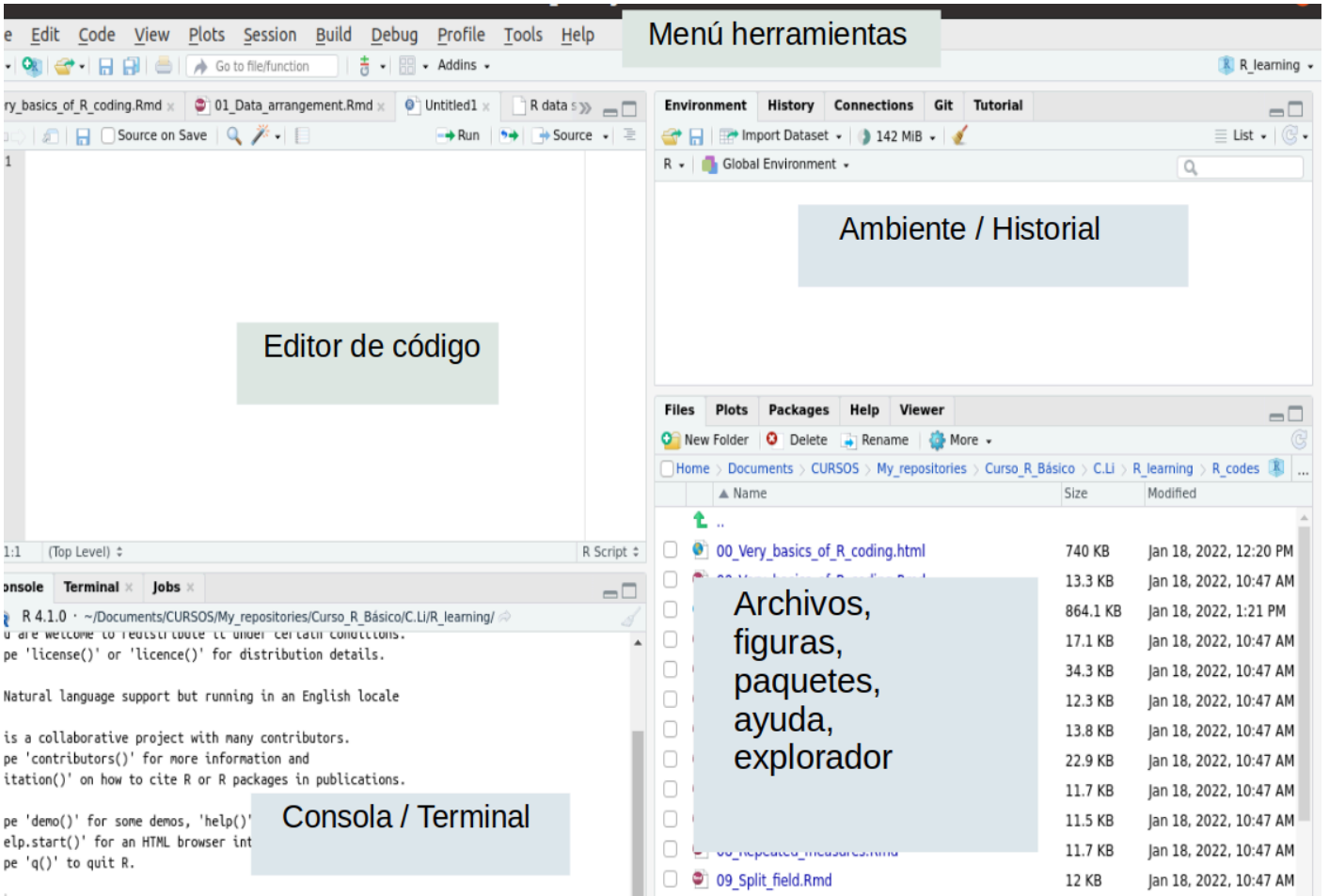
1. R es gratuito y de código abierto.
2. Se ejecuta en todas las plataformas principales: Windows, Mac Os, UNIX/Linux.
3. Los *scripts* y los objetos de datos se pueden compartir sin problemas entre plataformas.
4. Existe una comunidad grande, creciente y activa de usuarios de R y, como resultado, hay numerosos recursos para aprender y hacer preguntas.
5. Es fácil para otras personas contribuir complementos o paquetes que les permiten a los desarrolladores compartir implementaciones de software de nuevas metodologías de ciencia de datos.

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.

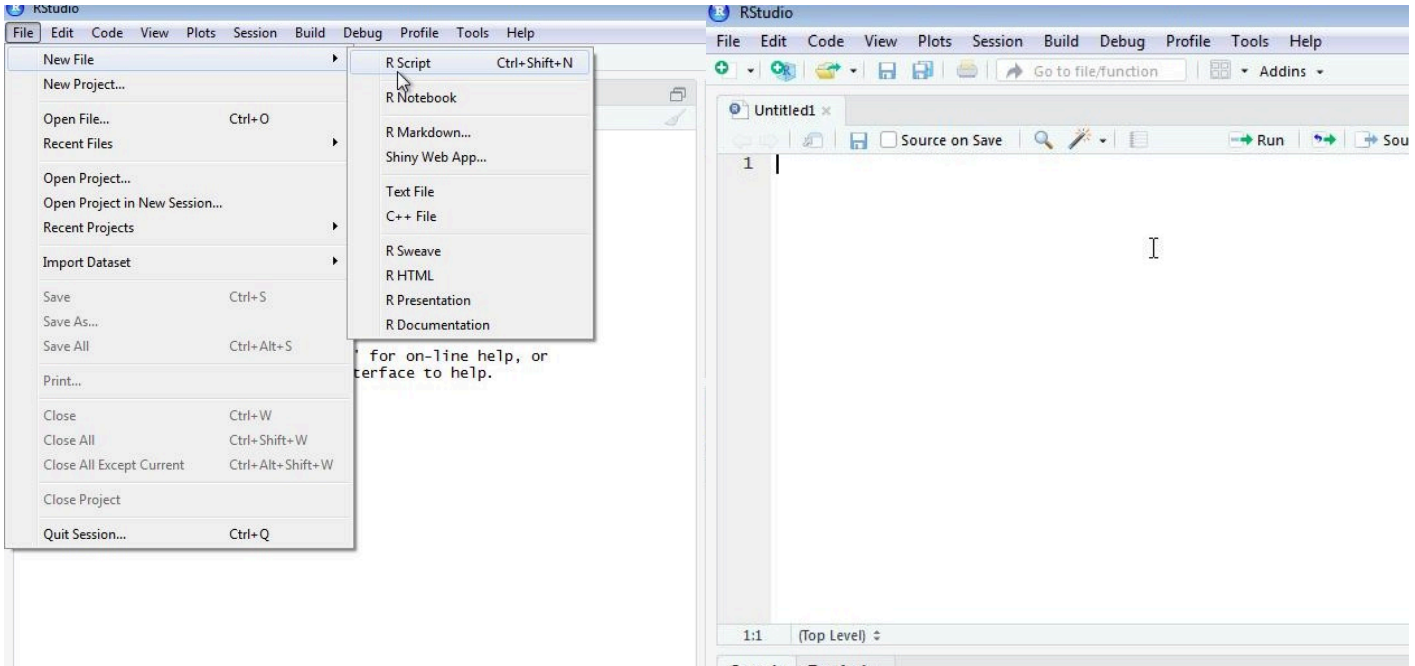
Paneles

Rstudio posee 4 paneles principales:



Scripts

Una de las grandes ventajas de R y Rstudio es que se pueden guardar los diversos códigos e instrucciones, los llamados *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*.



Abriendo un *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.

Por ejemplo.

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

Otra opción es yendo a *File*, entonces *Open File* y luego buscando en las carpetas donde tengas el script.

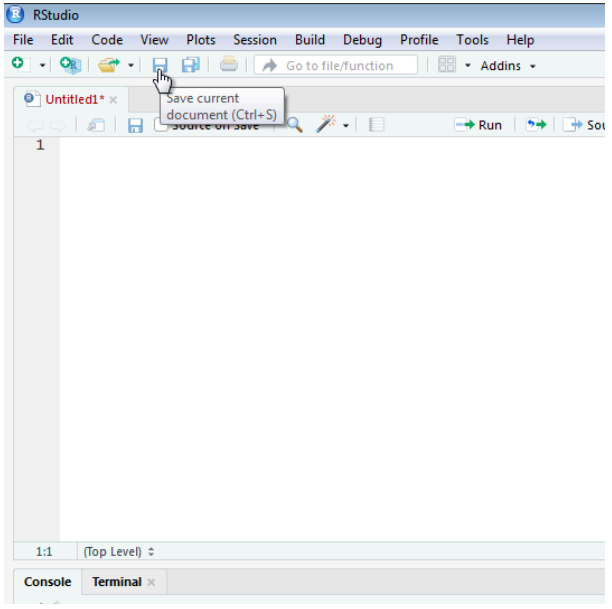
Comentarios en *Scripts*

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. Por ejemplo:\

```
#Este es mi código
data(iris)
#Aquí termina el código
```

Cómo ejecutar comandos mientras editan *scripts*

Empezamos por abrir un nuevo *script*, luego lo guardamos y le dabmos un nombre descriptivo. Ahora podemos editar nuestro primer *script*.



Cómo ejecutar comandos mientras editan *scripts*

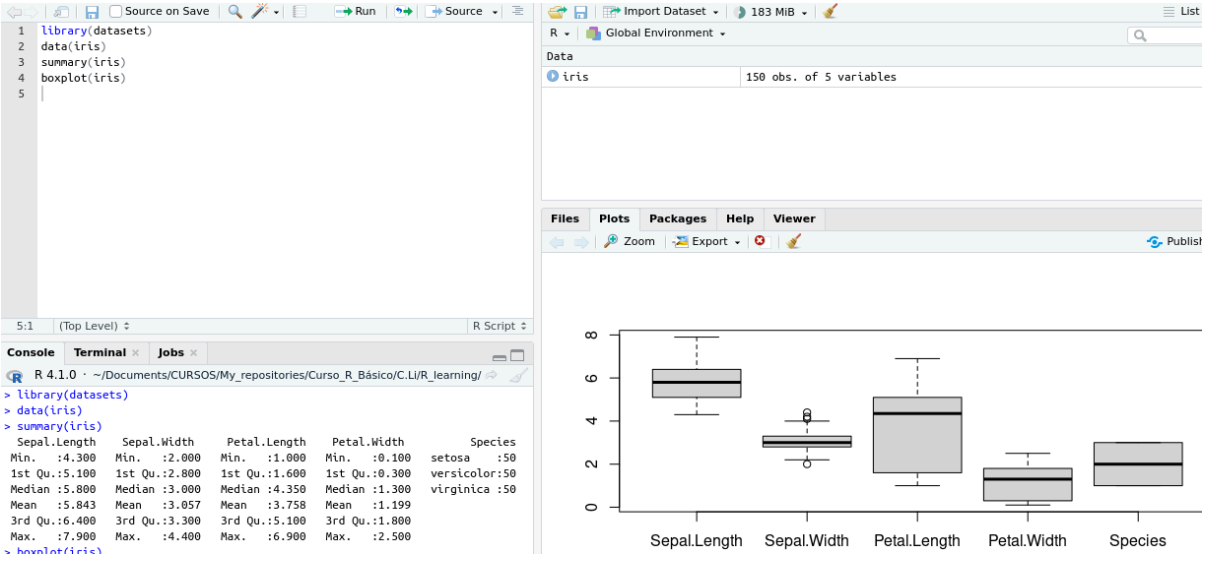
Para hacer esto, escribimos cada línea de código y luego hacemos click en el botón *Run* en la parte derecha superior del panel de edición. Para ejecutar una línea a la pueden usar Control+Enter en Windows y Linux y Command+Return en Mac.

Estas son las líneas del código:

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

Cómo ejecutar comandos mientras editan *scripts*

Y así luce al correrlo:



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.

Directorio de trabajo

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

```
getwd()
```

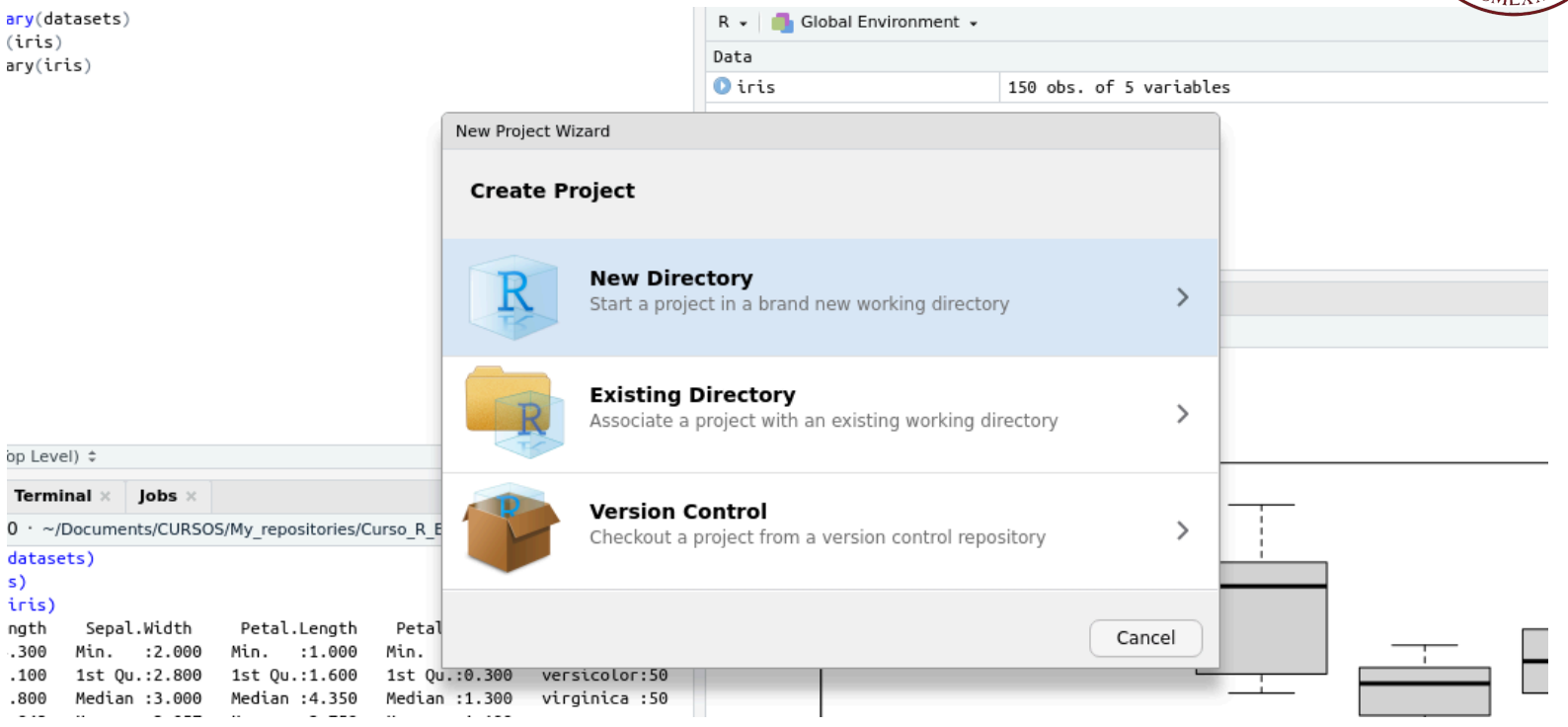
Y para establecer o cambiar este directorio de trabajo, correr el siguiente código:

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

Proyecto

- Un proyecto de R (extensión .Rproj) identifica todos los archivos y contenido asociados.
- Ayuda a organizar tu trabajo y así cada curso, artículo o trabajo diferente en un proyecto por separado.
- Al crear un proyecto todos los ficheros o carpetas quedan vinculados directamente a él.

Proyecto


ary(datasets)
(iris)
ary(iris)
R
Global Environment
Data
iris
150 obs. of 5 variables
New Project Wizard
Create Project
New Directory
Start a project in a brand new working directory
Existing Directory
Associate a project with an existing working directory
Version Control
Checkout a project from a version control repository
Cancel
op Level)
Terminal
Jobs
0 · ~/Documents/CURSOS/My_repositories/Curso_R_E
datasets)
s)
iris)
ngth Sepal.Width Petal.Length Petal
.300 Min. :2.000 Min. :1.000 Min.
.100 1st Qu.:2.800 1st Qu.:1.600 1st Qu.:0.300 versicolor:50
.800 Median :3.000 Median :4.350 Median :1.300 virginica :50

Instalación de paquetes de R

- Dentro de las ventajas de R está que muchos desarrolladores y programadores elaboran constantemente *paquetes* que nos permiten usar en acceso libre con muchas funcionalidades.

- Actualmente hay muchos paquetes disponibles en CRAN que es una red de servidores alrededor del mundo. También hay muchos paquetes desarrollados publicados en GitHub y en Bioconductor.

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

Instalación de paquetes de R

En RStudio pueden navegar a la pestaña *Packages* 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 una vez, 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.

Instalación de paquetes de R

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 pares 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.

Tipos de objetos en R

Aquí unos ejemplos:

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

Visualizando variables asignadas

Podemos poner el nombre o print(). Otra forma de examinar los objetos es buscarlos en el *Enviorment* o ambiente de trabajo y visualizarlos desde allí.

Por ejemplo, si escriben **f**, verán lo siguiente: `Error: object 'f' not found`.

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.
- **NO SE RECOMIENDA** guardar el espacio de trabajo porque si no se trabaja con diferentes proyectos será más difícil darle seguimiento a lo que guardan y ocupará mucho espacio en la memoria de su disco.
- En cambio, **ES RECOMENDABLE** en gran medida realizar un proyecto por trabajo o tarea y así tener un espacio de trabajo para cada uno.

Guardar los espacios de trabajo y exportar objetos de R

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

```
write.table(d, "data.txt", sep = "\t")
write.csv(d, "data.csv")
saveRDS(d, "data.RDS")
```

Funciones

- Hasta aquí ya hemos usado varias funciones tales como las funciones para exportar datos como `write.csv()`.
- Hay otras funciones más sencillas, por ejemplo, la función `log` o `sqrt` con las que poemos obtener el logaritmo o la raíz cuadrada.
- R incluye muchas funciones por *default* o preestablecidas y otras pueden extraerse al cargar los diversos paquetes.

  (Nótese que las funciones en su gran mayoría están definidas en inglés).

Funciones

- La sintaxis de R nos indica que necesitas usar paréntesis para evaluar 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, 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
```

Funciones

- También, mientras escribimos la función si usamos la tecla *tab* nos indica 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
```

Funciones


```
a + a
```

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

```
a - 2
```

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

```
1 * pi
```

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

```
2 / 3
```

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

```
4 ^ a
```

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

Funciones

También es posible declarar una función que no esté definida en R:

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

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

```
mean(x)
```

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

Tipos de datos

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

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
as.factor()	Factor
as.logical()	Lógico

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
- Matrices y arreglos
- Listas
- *Dataframes*

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

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, un arreglo es una matrix pero con n dimensiones, mientras que las matrices tienen solo dos dimensiones.

Matrices


```
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
```

Listas

- Las listas, al igual que los vectores, son estructuras de datos unidimensionales, sólo tienen largo
- 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.

Listas

Un ejemplo de una lista:

```
un_vector <- 1:20
una_matriz <- matrix(1:20, nrow = 2)
una_df     <- data.frame("numeros" = 1:3, "letras" = c("a", "b", "c"))
una_lista <- list("vec" = un_vector,"mat" = una_matriz,"df" = una_df)
una_lista
```

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

Dataframes

- 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.

```
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 ...
```

Dataframes

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


```
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
```

Dataframes


```
names(df)
```

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

```
colnames(df)
```

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

El operador $ y otros accesos

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
```

El operador $ y otros accesos


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

```
notas_estudiantes$nombres
```

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

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

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

Creando subconjuntos o Indexación


```
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
```

Creando subconjuntos


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

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

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

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

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

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

Creando subconjuntos

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

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:

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

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

Creando subconjuntos

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

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

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

Datos de R

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()
```

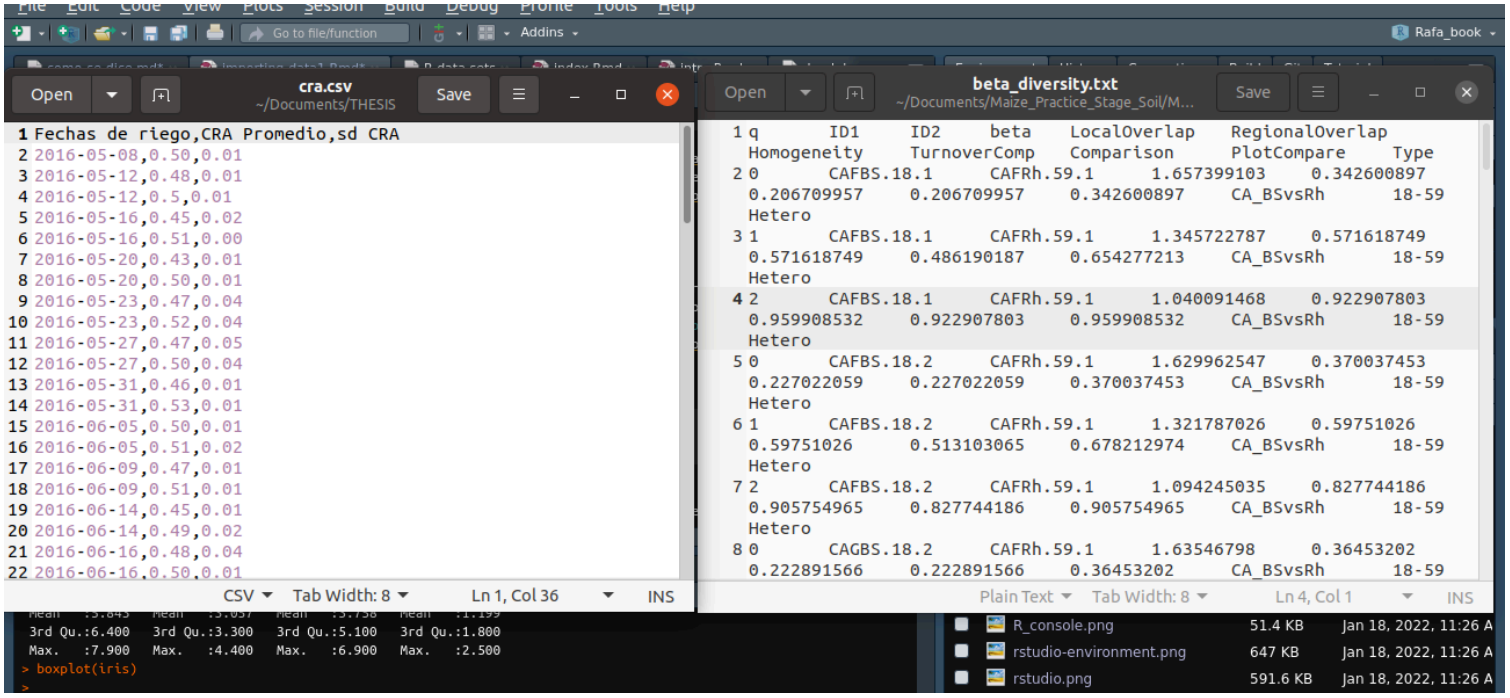
Importando datos

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...

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.

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/
  destfile = "iris.data")
```

```
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 la función `read.csv` o `read.delim` de R base.

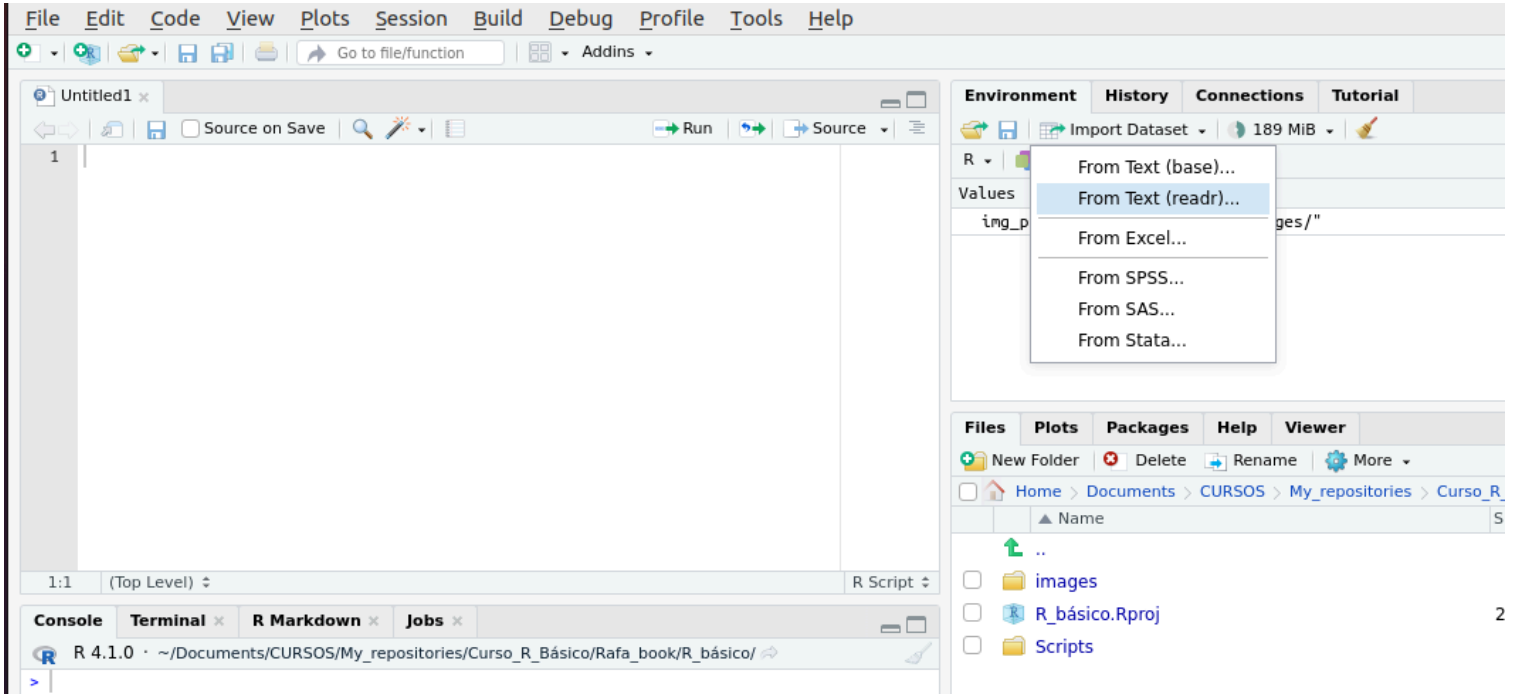
```
iris_data<- read.csv("iris.data", header = F)
iris_data<- read.delim("iris.data",header = F, sep = ",")
```

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

```
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



Los paquetes readr y readxl: readr

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 delimitados por tabulación	tsv o txt
`read_delim`	Archivo de texto general; se debe definir el delimitador	txt, csv o tsv

Los paquetes readr y readxl: readxl

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

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.

Gráficas R base

La función *plot()*

La función *plot()* es usada de manera general para generar gráficos. Esta función es muy especial porque depende del tipo de datos que le demos generará diferentes tipos de gráficos. El argumento principal que pide esta función es *"x"* también podemos poner *"y"*. Y depende de estos el tipo de gráfica que se generará. Diremos:

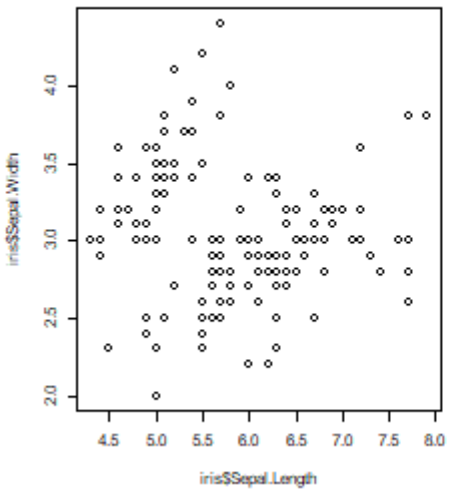
- **Continuo**: Cuando nos referimos a un vector numérico, entero, lógico o complejo.
- **Discreto**: Cuando nos referimos a un vector de factores o cadenas de texto.

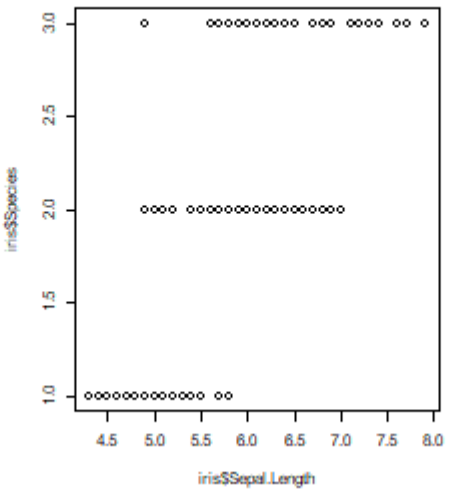
La función *plot()*

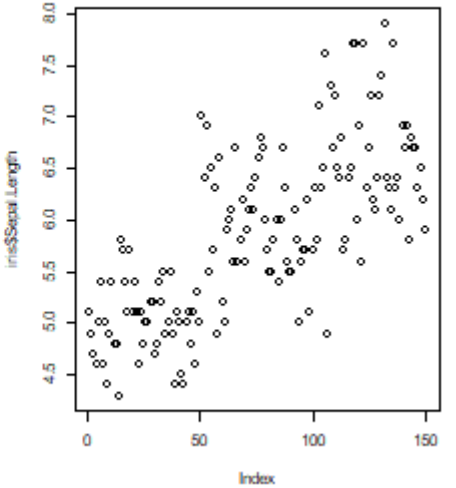
x	y	Tipo Gráfico
Continuo	Continuo	Dispersión /Scatter
Continuo	Discreto	Dispersión y coercionada a numérica
Continuo	Ninguno	Dispersión por número de renglón
Discreto	Continuo	Boxplot/Cajas
Discreto	Discreto	Mosaico
Discreto	Ninguno	Barras

La función *plot()*

```
#Ejemplos
par(mfrow= c(1,3))
plot(x=iris$Sepal.Length, y=iris$Sepal.Width)
plot(x=iris$Sepal.Length, y = iris$Species)
plot(x=iris$Sepal.Length)
```



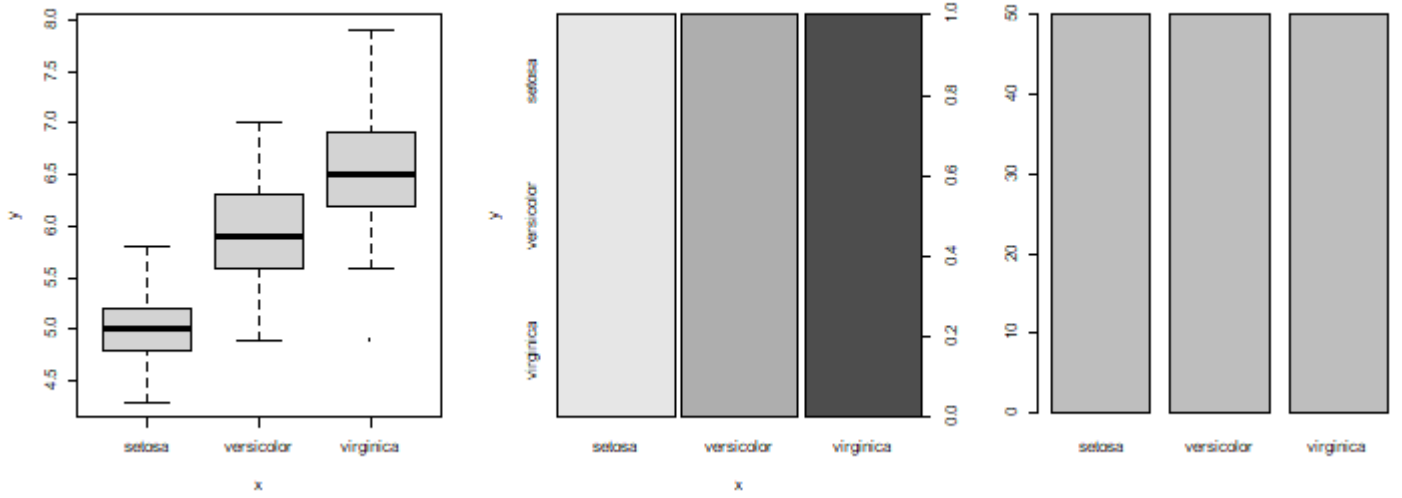




La función *plot()*

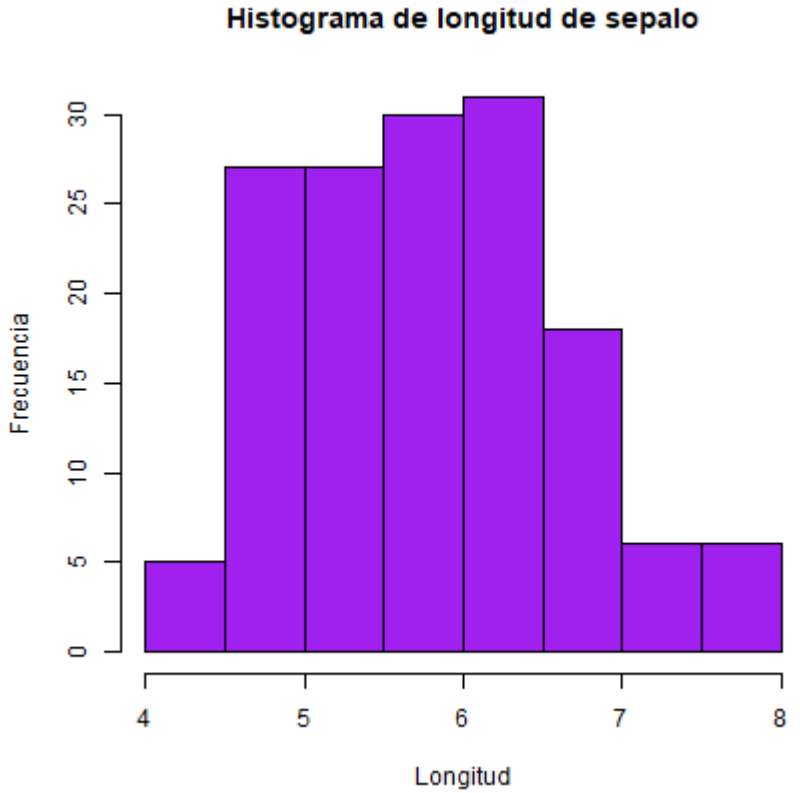
```
par(mfrow= c(1,3))

plot(x = iris$Species, y = iris$Sepal.Length)
plot(x=iris$Species, y=iris$Species)
plot(x=iris$Species)
```



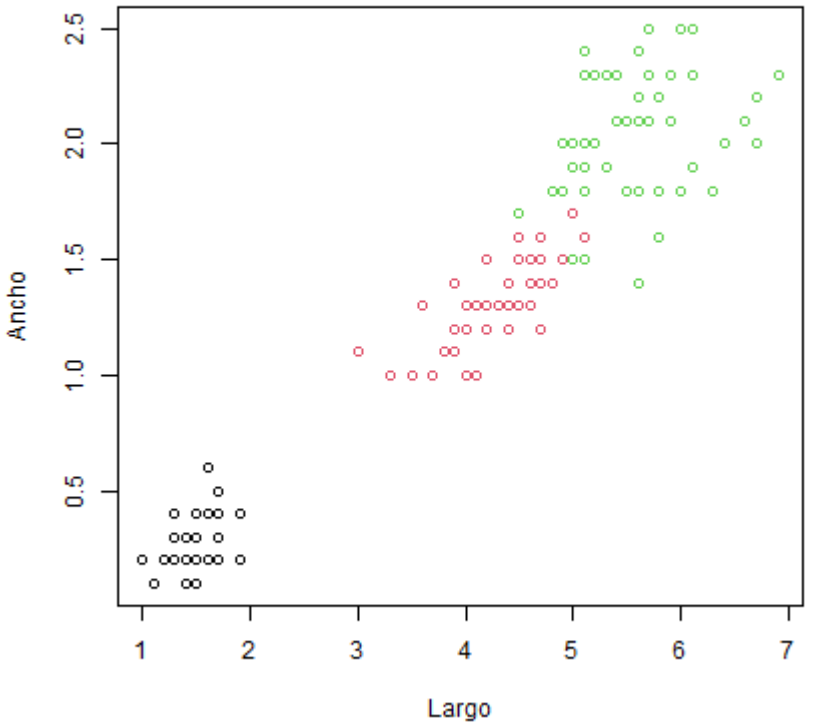
Histogramas

```
hist(x = iris$Sepal.Length, main = "Histograma de longitud de sepalo",
     xlab = "Longitud", ylab = "Frecuencia",
     col = "purple")
```



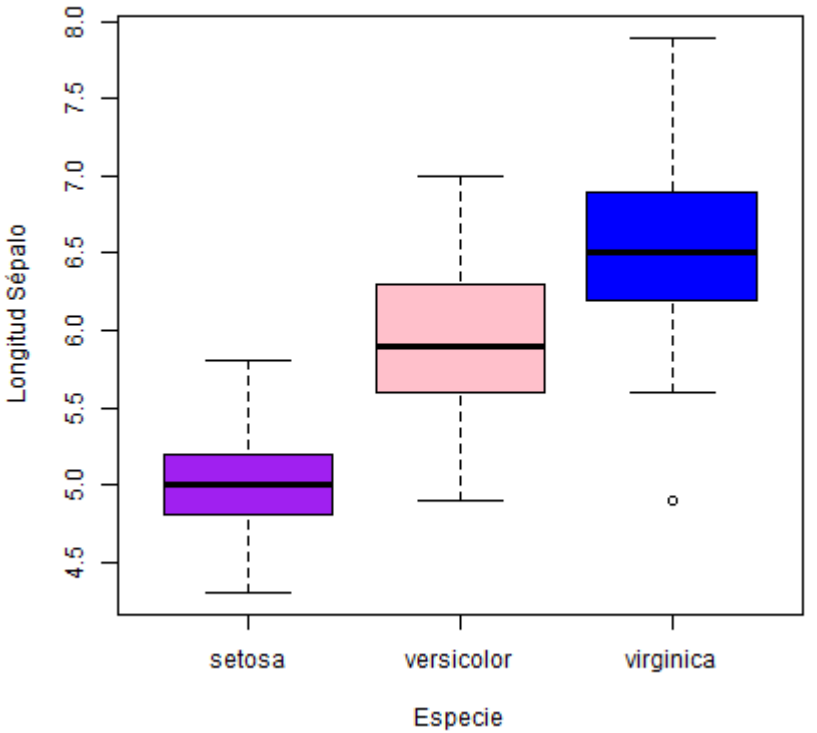
Diagramas de dispersión

```
plot(x = iris$Petal.Length, y = iris$Petal.Width,
     col = iris$Species, xlab = "Largo", ylab = "Ancho")
```



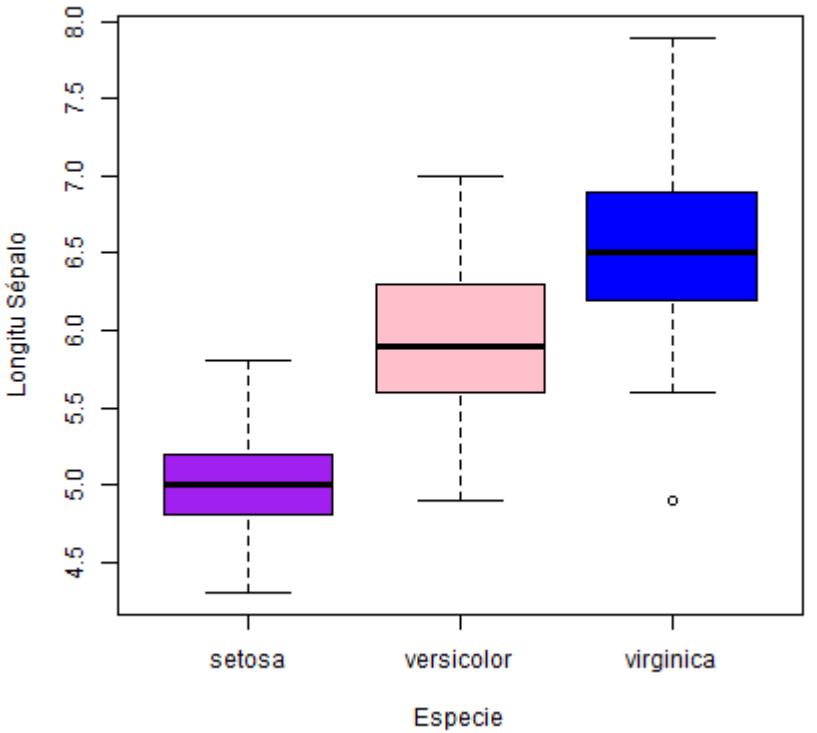
Boxplots o diagramas de cajas

```
plot(x=iris$Species, y = iris$Sepal.Length, xlab = "Especie",
     ylab = "Longitud Sépalo", col = c("purple", "pink", "blue"))
```



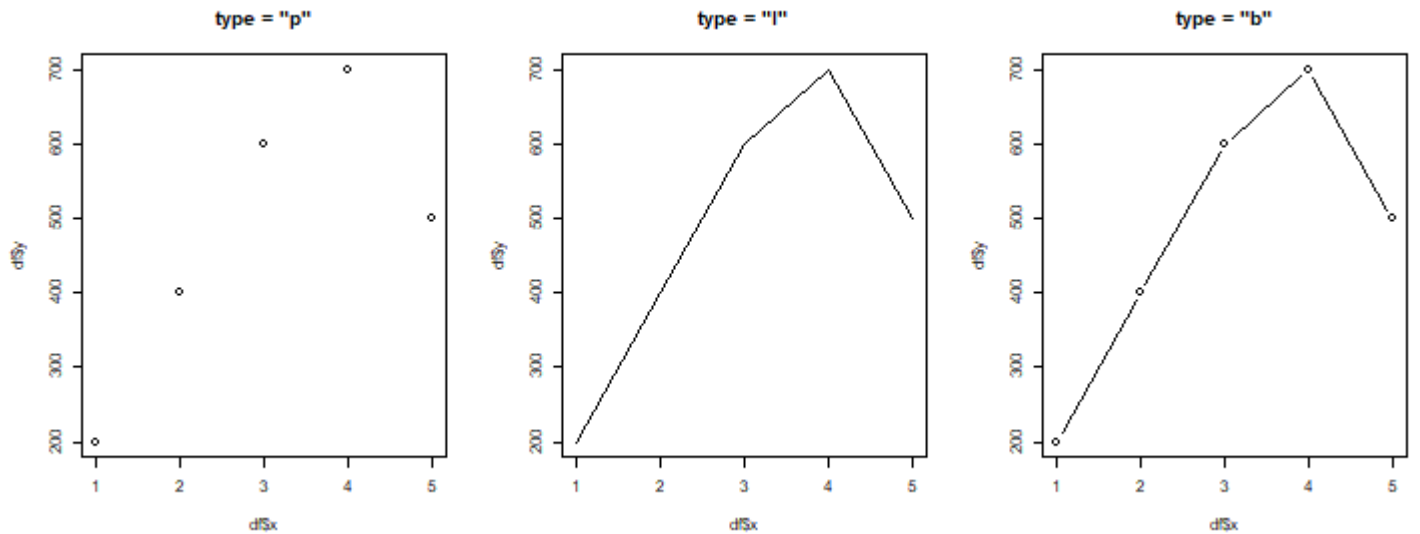
Boxplots o diagramas de cajas

```
boxplot(formula = Sepal.Length ~ Species, data =  iris, xlab = "Especie"
        ylab = "Longitu Sépalo",      col = c("purple", "pink", "blue"))
```



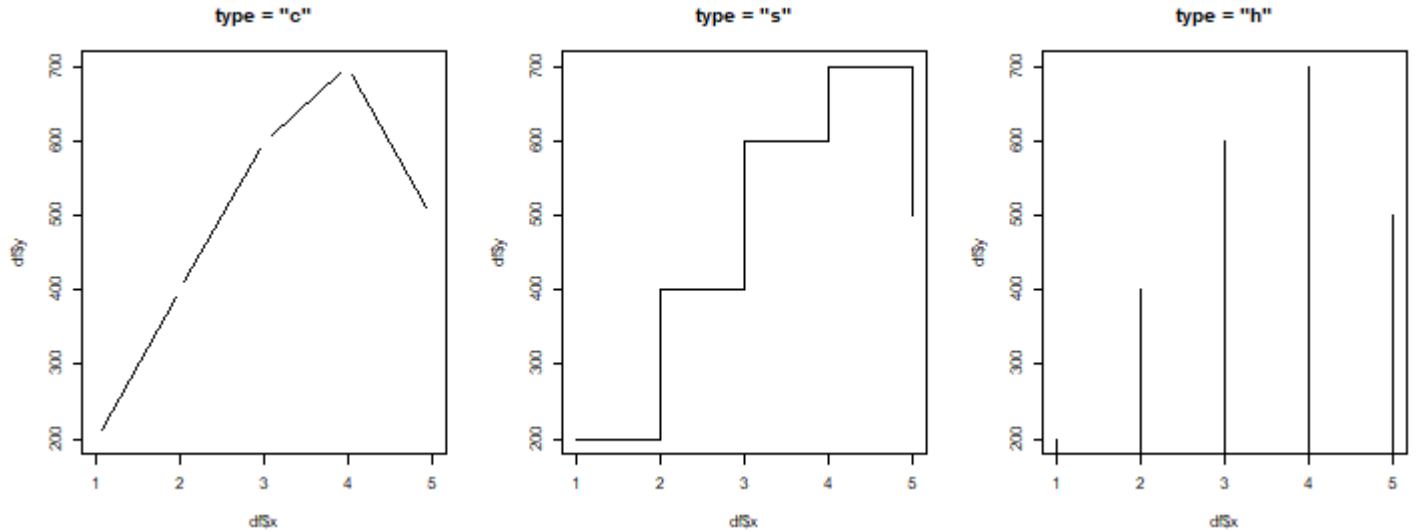
Otros gráficos

```
#dataset de ensayo
df<- data.frame(x= c(1:5),y= c(200, 400, 600, 700, 500))
par(mfrow = c(1, 3))
plot(df$x, df$y, type = "p", main = 'type = "p"')
plot(df$x, df$y, type = "l", main = 'type = "l"')
plot(df$x, df$y, type = "b", main = 'type = "b"')
```



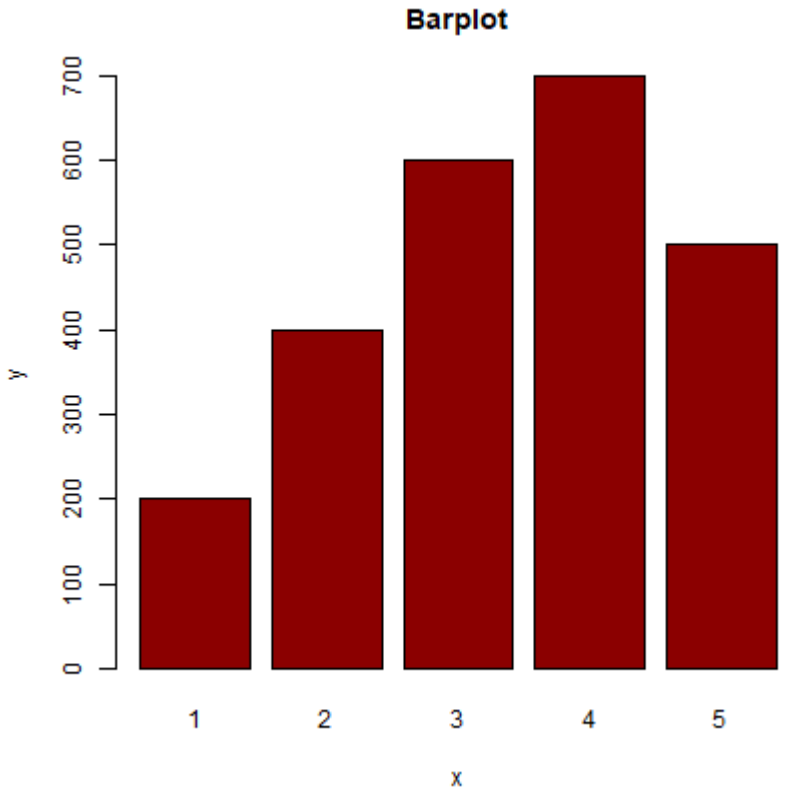
Otros gráficos

```
par(mfrow = c(1, 3))
plot(df$x, df$y, type = "c", main = 'type = "c"')
plot(df$x, df$y, type = "s", main = 'type = "s"')
plot(df$x, df$y, type = "h", main = 'type = "h"')
```



Barplots o gráficas de barras

```
barplot(y ~ x , data = df, main = "Barplot", col = "darkred")
```



Guardando una gráfica

Para guardar o exportar una gráfica debemos:

1. Indicar las instrucciones de cómo exportaremos la imagen
2. Usar *plot()* y graficarla y luego,
3. *dev.off*() para quitarla del panel y exportarla

```
png(filename="gráfica1.png", width=648, height=432)
plot(df$x, df$y, type = "p", main = 'type = "p"')
dev.off()
```

tidyverse


tidyverse
www.rstudio.com


ME ESTÁS DICIENDO
QUE EXISTE UN TIDYVERSO?
imgflip.com

tidyverse

Hasta ahora hemos estado manipulando las tablas o *dataframes* creando subconjuntos mediante la indexación y utlizando otras funciones del Rbase.

Sin embargo, existe todo un universo llamado *tidyverse* que nos perimte hacer todo esto que vimos y más de manera más intuitiva.

tidyverse

Podemos cargar todos los paquetes del *tidyverse* a la vez al instalar y cargar el paquete **tidyverse**:

```
library(tidyverse)
```

Manipulación de *data frames*

El paquete **dplyr** del *tidyverse* ofrece funciones que realizan algunas de las operaciones más comunes y que ya vimos el capítulo anterior con R base.

Las funciones principales de *dplyr* son:

- *select*,
- *mutate*,
- *filter* y
- *summarise*.

Pero antes, revisemos lo que es el *pipe*.

El *pipe*: %>%

El *pipe* es la herramienta que nos permite darle *dplyr* las órdenes, comandos o funciones a realizar. Lo podemos poner con el atajo del teclado **"Ctrl + Shift + M (Windows/linux)"** y **"Cmd + Shift + M (Mac)"**.\

```
#asignando la data a la variable "mi_data"
mi_data<-ToothGrowth

ToothGrowth %>% select(len, dose) %>% glimpse()
```

```
## Rows: 60
## Columns: 2
## $ len  <dbl> 4.2, 11.5, 7.3, 5.8, 6.4, 10.0, 11.2, 11.2, 5.2, 7.0, 16.5, 16
## $ dose <dbl> 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 1.0, 1.0, 1.
```

Datos discretos

Y si queremos renombrar los datos en una columna, entonces:

```
ToothGrowth%>%mutate(dose = case_when(
    dose == 0.5 ~ "D_0.5",
    dose == 1 ~ "D_1",
    dose == 2 ~ "D_2")) %>%  mutate(
      dose = factor(dose, levels = c("D_2", "D_1", "D_0.5"))) %>% head()
```

```
##    len supp  dose
## 1  4.2   VC D_0.5
## 2 11.5   VC D_0.5
## 3  7.3   VC D_0.5
## 4  5.8   VC D_0.5
## 5  6.4   VC D_0.5
## 6 10.0   VC D_0.5
```

Resumiendo los datos


```
ToothGrowth %>% group_by(supp) %>% summarise(promedio = mean(len),
                                             suma = sum(len),
                                             n = n(),
                                             mediana =median(len))
```

```
## # A tibble: 2 × 5
##   supp  promedio  suma     n mediana
##   <fct>    <dbl> <dbl> <int>   <dbl>
## 1 OJ        20.7  620.    30    22.7
## 2 VC        17.0  509.    30    16.5
```

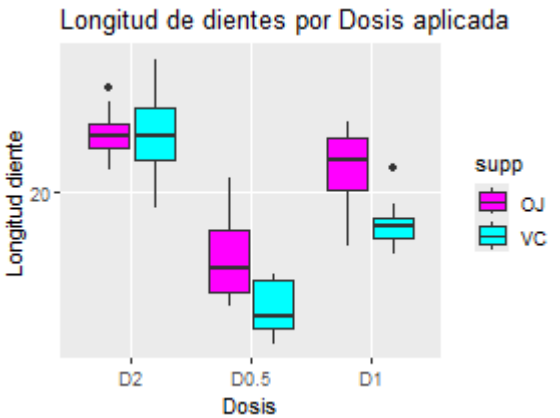
Gráficas con *ggplot2()*

- Las gráficas con *ggplot2()* es quizás de las cosas más poderosas y atractivas de R si lo comparamos con otros lenguajes y/o programas.

- *ggplot2* es generalmente más intuitiva porque usa una gramática de gráficos además de ser visualmente agradable.

Gráficas con *ggplot2()*

```
ToothGrowth %>%  mutate(dose=case_when(
dose==0.5~"D0.5",dose==1~"D1", dose==2~"D2")) %>%
  ggplot(aes(x = dose, y = len, fill=supp)) +
  geom_boxplot()+
  ylab("Longitud diente")+  xlab("Dosis")+
  ggtitle("Longitud de dientes por Dosis aplicada") +
  scale_x_discrete(limits = c("D2", "D0.5", "D1"), position ="bottom" )+
  scale_y_continuous(breaks = c(0, 20,40))+
  scale_fill_manual(values = c("#FF00FF","#00FFFF"))
```



¡Muchas gracias!