

Parte II - A



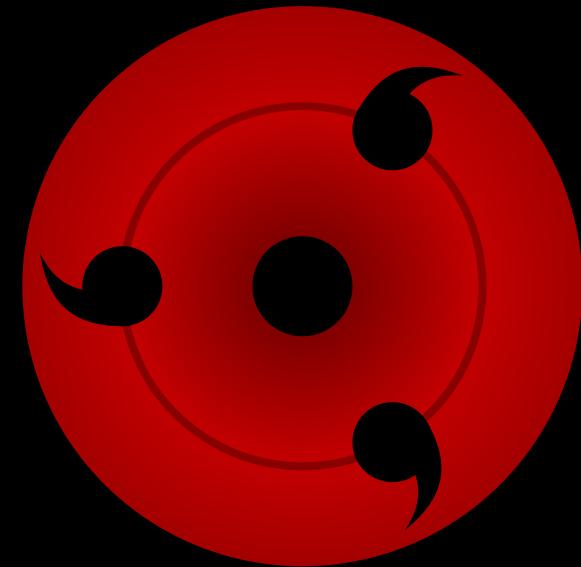
con Branext

A.Otiniano (co-author: J.Andrade)

AoZ Geoanalytics And Multidisciplinary Sensing, Universal Accessibility
and Machine Learning Group, National University of Engineering

2023/01/01 (updated: 2024-05-23)

Image credit: [Wikimedia Commons](#)



Data Science in Mining Geology and Machine Learning

/ʃa:.riŋ.gan/

Empecemos!!

Al infinito y más allá!



R

R

Es un **potente lenguaje orientado a objetos y destinado al análisis estadístico y la representación de datos desarrollado** por Ross Ihaka y Robert Gentleman en 1993. Es además un software libre. En la cual ingenieros, científicos, economistas entre otras ramas trabaja en el análisis de datos. Se puede considerar a R como un software estadístico del 2024 y a su vez un lenguaje de programación.

¿Porqué usar R?

Es libre - multiplataforma - Analiza cualquier tipo de objeto.

Es potente, **sumamente potente**.

Su capacidad gráfica es difícilmente superada por algún paquete estadístico.

Es compatible con casi todos los formatos de datos (.csv, .xls, .xlsx, .sav, .sas?, entre otros).

Es ampliable. Lenguaje de script: puedes automatizar o realizar tus propias rutinas.

Hay miles de técnicas estadísticas implementadas, y cada día más.

Rstudio

Rstudio

Es una herramienta **IDE (Integrated Development Environment)** fundada por J.J Allaire, es libre y gratuita con los siguientes beneficios:

- Trabaja con R de forma interactiva.
- Organiza el código y mantiene en múltiples proyectos.
- Actualizaciones y mantenimiento de paquetes.
- Crear y compartir informes de forma sencilla.
- Compartir código y colaborar con otros usuarios (*implementar técnicas*).
- Permite *automatizar procesos* crear **Dashboards, documentos interactivos (html - web based), aplicaciones interactivas, programas, paquetes, APIs, Modelos, entre muchos otros.**

Rstudio no realiza operaciones estadísticas solo facilita realizarlas. Instalar **Rstudio** IDE desde [Rstudio IDE](#)

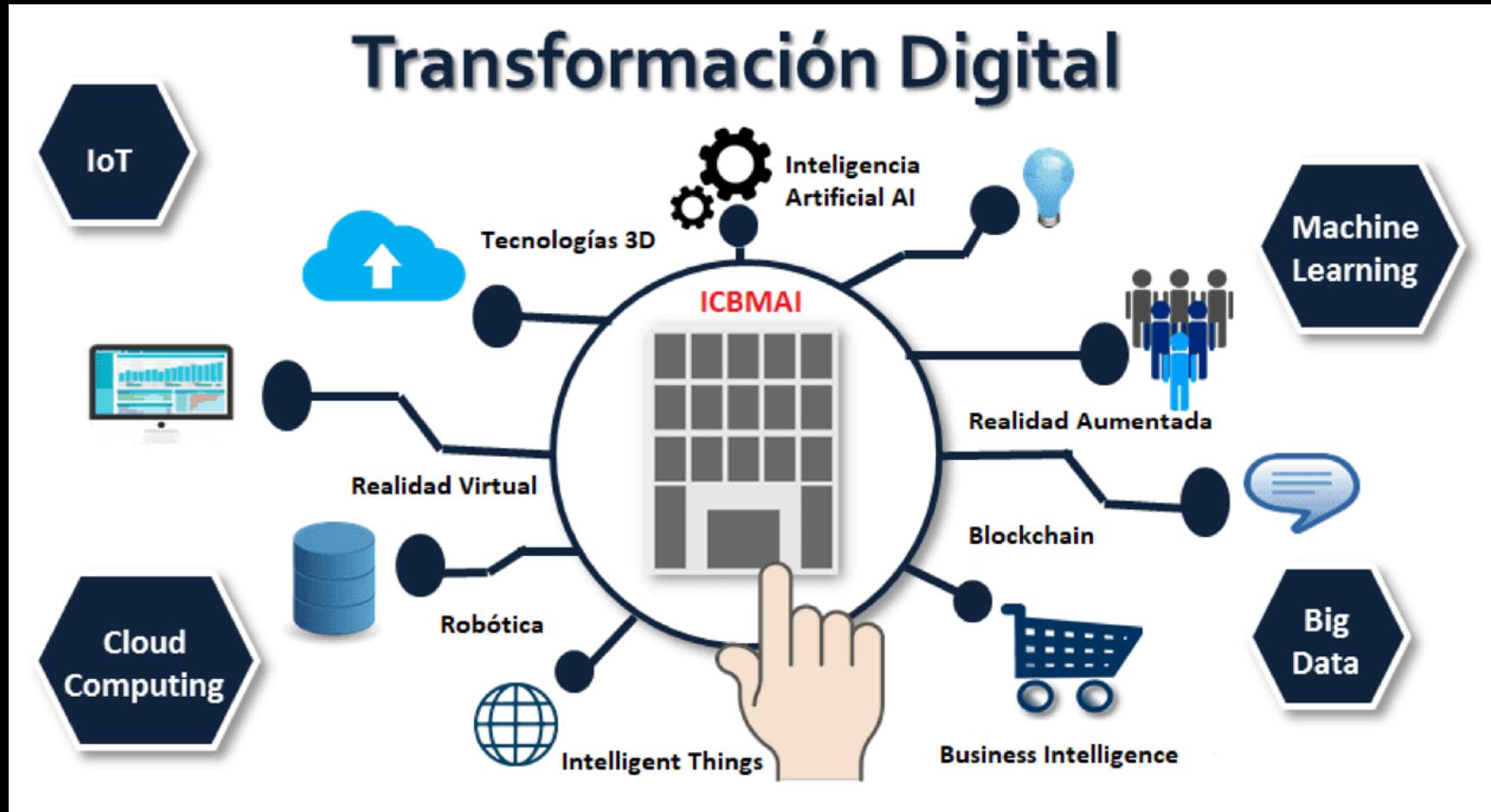
- Crear un nuevo documento Rmarkdown desde el menú **File -> New File -> R Markdown -> OK**¹
- Click en el botón **Knit** para compilar;

[1] Mirar [#1](#) si existen problemas para descargar Rstudio.

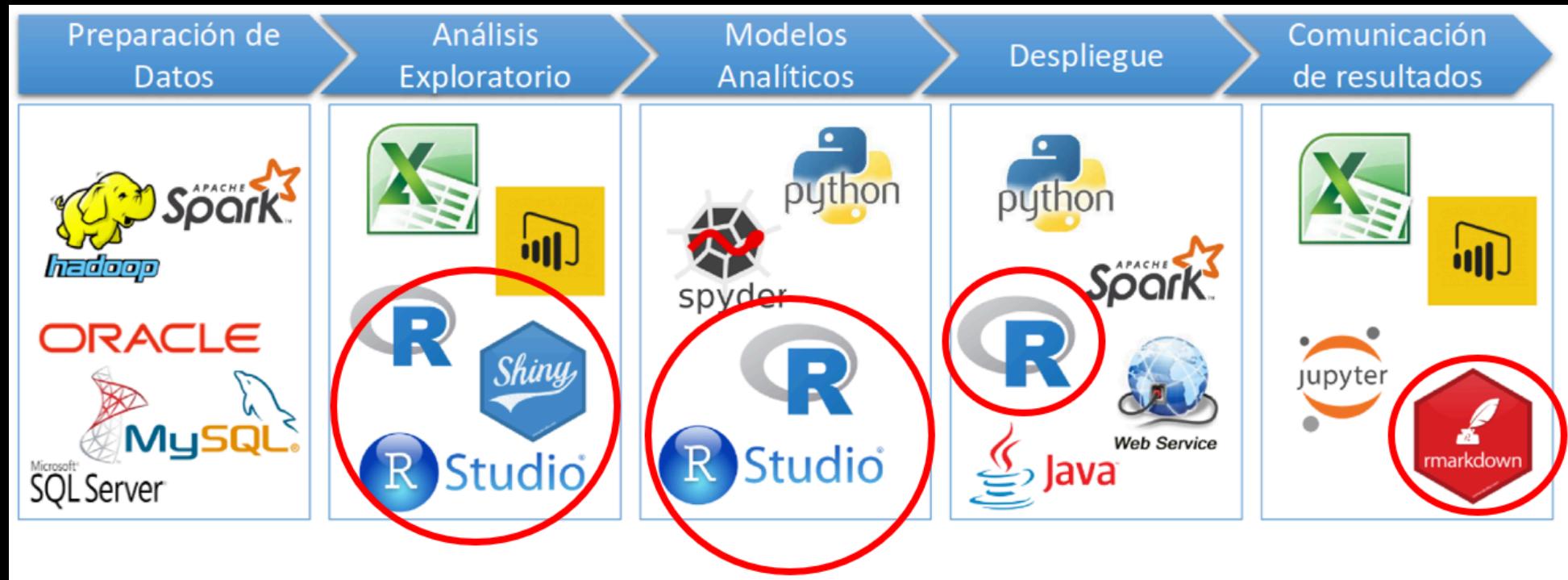
Características de Rstudio

- Integración - Ejecución - Resaltado.
- Ayuda - Autocompletado.
- Teclas de atajo.
- Navegadores de Objetos.
- Gestión del historial de comandos.
- Navegación del Código.
- Importación y visualización de datos.
- Integración de gráficos.
- Gestión de proyectos.
- Control de versiones.
- Generación de documentos.

R Y RSTUDIO - TRANSFORMACIÓN DIGITAL

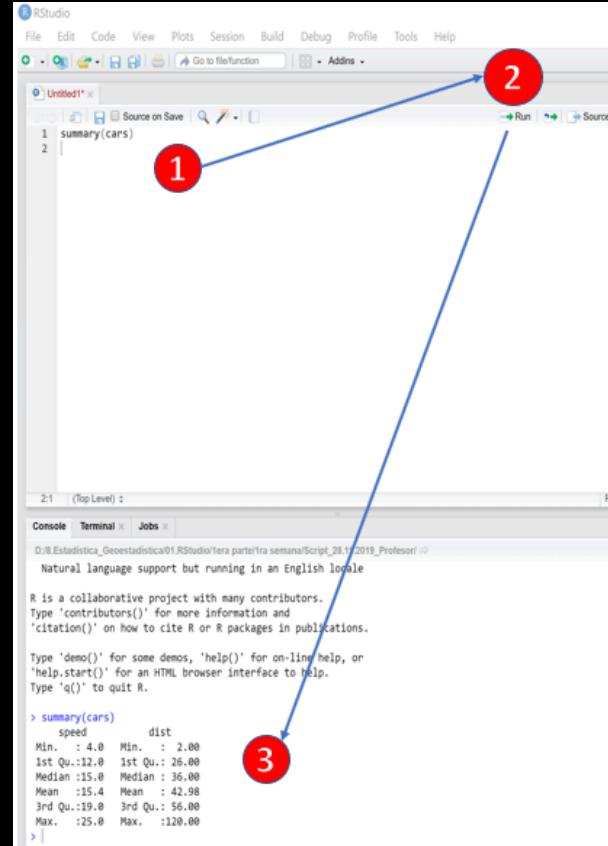


R Y RSTUDIO BIG DATA Y MACHINE LEARNING



Entorno de Rstudio

- Abrir el IDE Rstudio **File** -> **New File** -> **R Script** -> **Click**.⁰
- Escribir **summary(cars)**.¹
- Click en **Run**.²
- Mirar la Salida;³



```
summary(cars)
```

```
2:1 (Top Level) :
```

```
Console Terminal Jobs
```

```
D:\R\Estadistica_Geostadistica\01.RStudio\tercera parte\tra semana\Script_18.1.2019_Profesor
```

```
Natural language support but running in an English locale
```

```
R is a collaborative project with many contributors.
```

```
Type 'contributors()' for more information and
```

```
'citation()' on how to cite R or R packages in publications.
```

```
Type 'demo()' for some demos, 'help()' for on-line help, or
```

```
'help.start()' for an HTML browser interface to help.
```

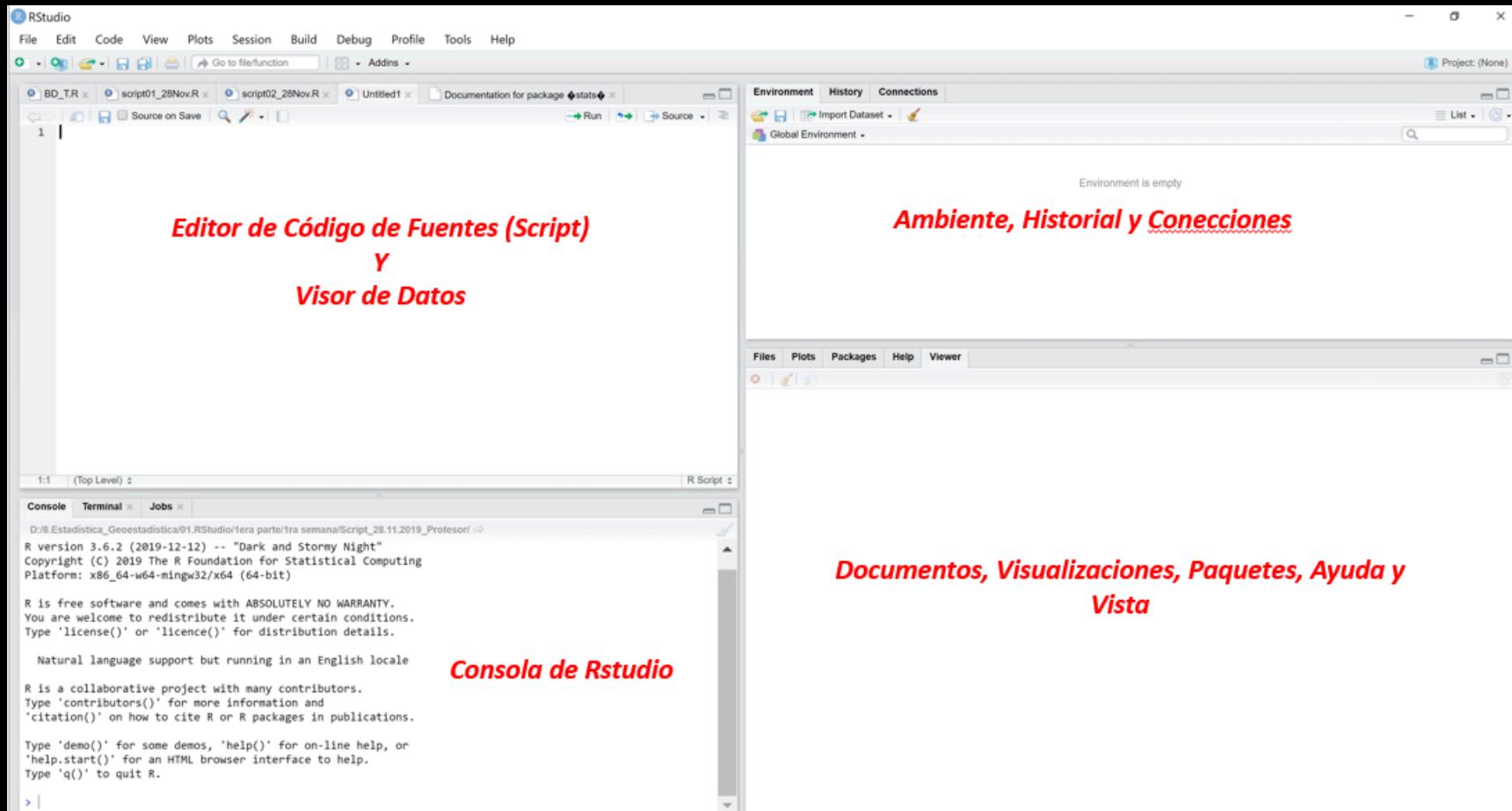
```
Type 'q()' to quit R.
```

```
> summary(cars)
```

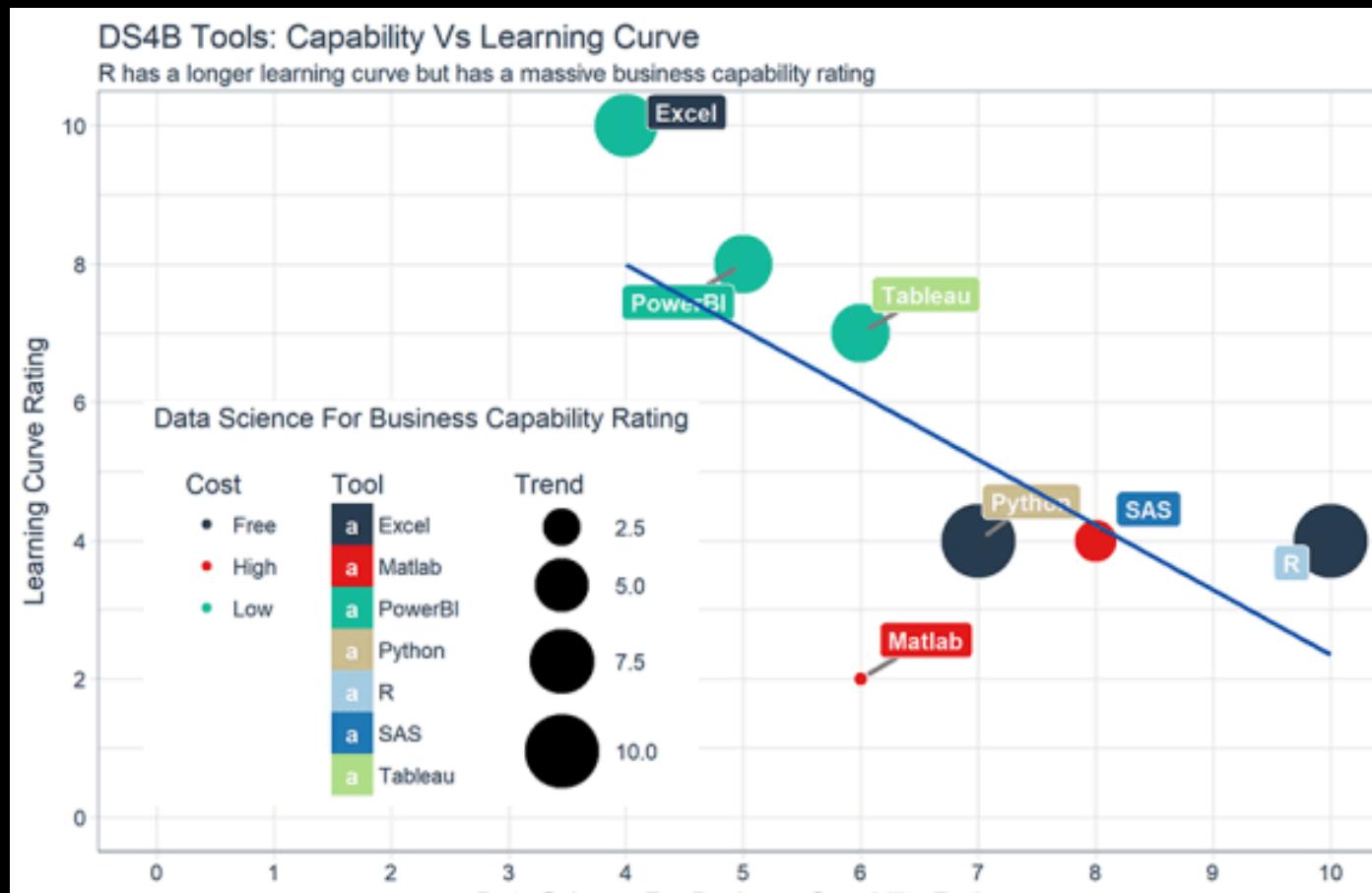
speed	dist
Min. : 4.0	Min. : 2.00
1st Qu.:12.0	1st Qu.: 26.00
Median :19.0	Median : 36.00
Mean :19.4	Mean : 42.98
3rd Qu.:25.0	3rd Qu.: 56.00
Max. :28.0	Max. :120.00

Entorno Rstudio

Entorno de Rstudio Paneles



Curva de Aprendizaje

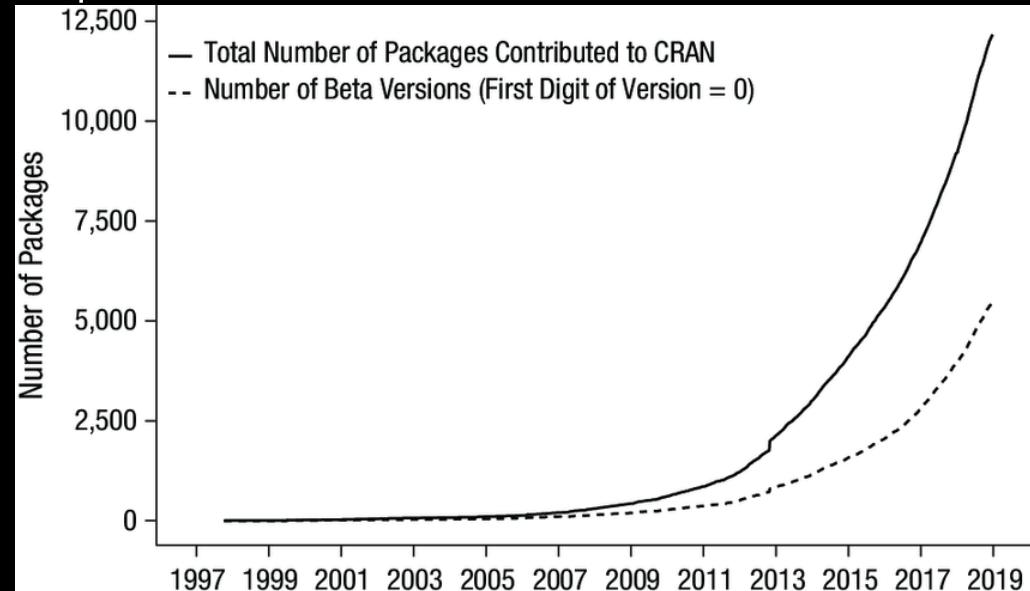


Capacidad Vs. Curva de Aprendizaje.

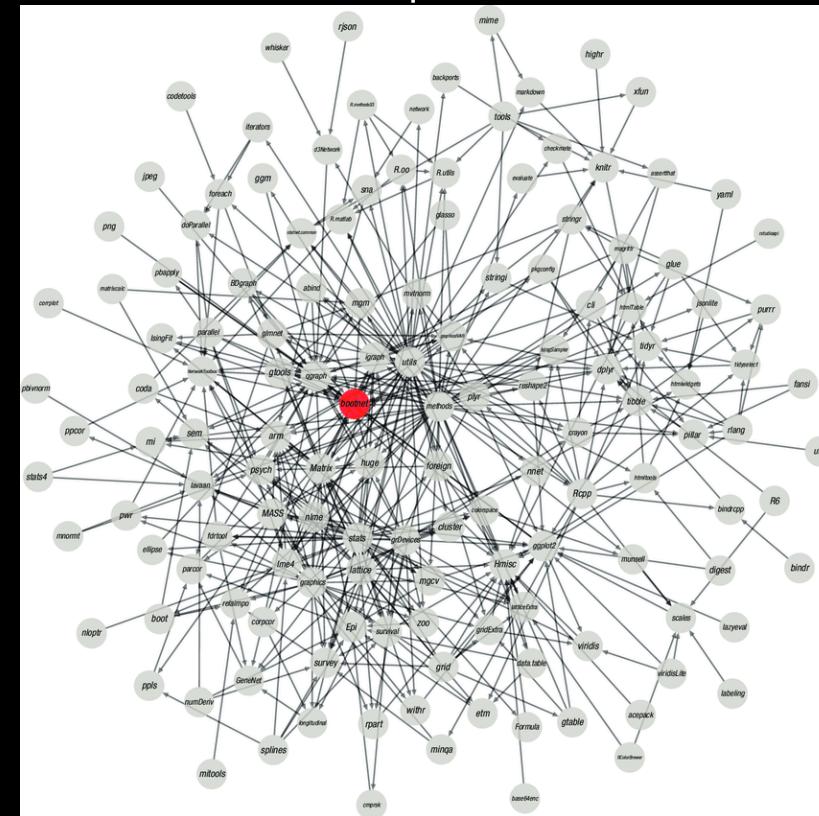
Paquetes y Librerías

Paquetes y Librerías por Defecto

Paquetes desde 1997 hasta el 2019



Red Neuronal de Paquetes



Creando un Proyecto para Trabajar:

Para crear un **proyecto** para trabajar necesitamos realizar los siguientes pasos:

- Abrir Rstudio **File** -> **New Project** -> **New Directory** -> **New Project**;¹
- En el directorio **Directory name** -> **Browse (Ubicar)** -> **Create Project (Click)**;²

```
#Directorio y Archivos Contenidos
dir()
#Funciones de Ayuda de Funciones:
help.start()
help(sum)
?sum
help.search("sum")
```

[*] Es importante tener como fuentes de ayuda [Stackoverflow](#), [R help](#), además del buscador de google como básicos, más adelante se buscarán soluciones a preguntas más complejas.

Un momento voy a tomar mi muestra de agua!



Mi primer Script

R como calculadora científica

R como calculadora científica es muy útil en la realización de cálculos matemáticos desde los *simples* hasta los **complejos**.

Para ejecutar ubicarnos al inicio de la línea y **Ctrl + Enter**.

```
2+2
```

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

```
sin(log(1+sin(pi/4)))
```

```
## [1] 0.509669
```

```
rnorm(10, mean=3, sd=2.4)
```

```
## [1] -1.8893356  0.1631999  2.7727605  5.6026777  4.9088013  4.1661785
## [7]  2.8506159  0.9010834  3.2673862  1.9779561
```

- Las áreas de interés de usuarios R, ver [Cran Task View](#) o también en [CRAN ORG](#)

Tipos de Variable y Operadores en R

Existe una secuencia en las características de los objetos en R: modos y atributos. En la cual **objetos -> compuesto de elementos -> tipo de elementos más simples: variables**

- Tipos de Data en R:
 1. *Escalares*
 2. *Vectores* (numeric, character, logical).
 3. *Matrices*.
 4. *Dataframe*.
 5. *Listas*.
- Básicos:^{*}
 1. **Numeric**: Número real tales como 4.5, 9, -2, notación científica 3.12e-47.
 2. **Logical**: **TRUE - FALSE** que es un *booleano*.
 3. **Character**: Valor dentro de "" o son textos (*string*).
 4. **Complejos**: Números de la forma $a + bi$.

[*] Existen otras clasificaciones.

Ejemplos de Tipo de Variables

```
x <- 28  
class(x)
```

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

```
y <- "R es fantastico"  
class(y)
```

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

```
z <- TRUE  
class(z)
```

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

```
#Guardar variables creadas:  
#<< save.image("mydata.RData") # Guarda variables del environment.  
#<< load("mydata.RData") #Carga las variables guardadas.
```

Operadores Básicos en Rstudio:

Operadores Aritméticos Básicos

Operador	Descripción
+	Adición
-	Sustracción
*	Multiplicación
/	División
[^] or ^{**}	Potenciación

Operadores Lógicos

Operador	Descripción
<	Menor que
<=	Menor igual que
>	Mayor que
>=	Mayor igual que
==	Exactamente Igual
!=	No igual a (Diferente)
X	No X
X	Y
X&Y	X y Y
X Y	X o Y
Is TRUE(x)	Prueba si x es TRUE

Definición de Vectores en R

Vectores en Rstudio

Los vectores son un **arreglo unidimensional** el cual es un objeto en R, los números son interpretados como vectores de una componente. La forma de crear un vector es con la función `c()`.

```
#Generando variable
vec_num <- c(1, 10, 49)
vec_num
```

```
## [1] 1 10 49
```

```
vec_chr <- c("a", "b", "c")
vec_chr
```

```
## [1] "a" "b" "c"
```

```
vec_bool <- c(TRUE, FALSE, TRUE)
vec_bool
```

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

```
vector_01 <- rnorm(12, mean=4, sd=2)
random_01 <- c(rnorm(5,10,5),runif(15,14,20))
range(random_01)
```

```
## [1] 5.399497 19.967134
```

```
range(random_01)[1]
```

```
## [1] 5.399497
```

```
range(random_01) == range(random_01)[1]
```

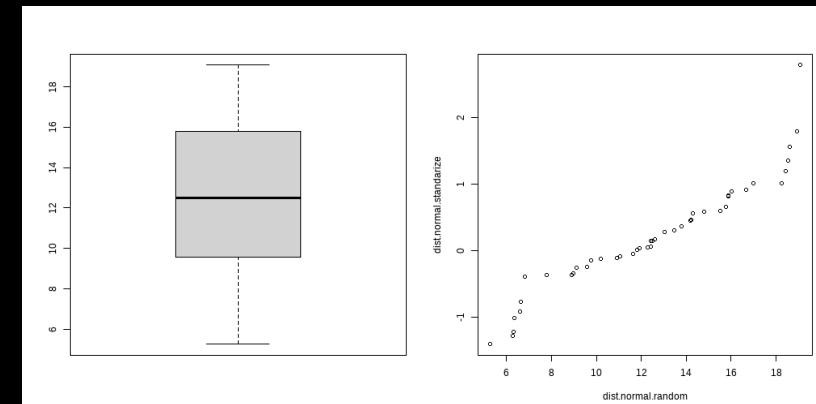
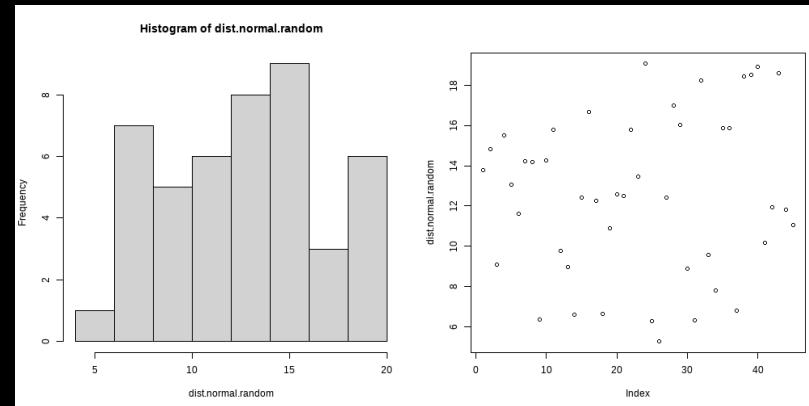
```
## [1] TRUE FALSE
```

```
random_02 <- rexp(10,0.5) #rexp(n,rate) genera
#de la función exponencial.
```

Jugando con Vectores I

```
y <- 42
Espero_sea_buena_la_clase <- c(20,20)
espero_sea_buena_la_clase <- c(20,20)
Variable.1 <- c(1,2,3,4,5)
variable.1 <- c(1,2,3,4,5)
dist.normal.random <- rnorm(n = 45,
                           mean = 12,
                           sd = 5)
dist.normal.standarize <- rnorm(n = 45,
                                 mean = 0,
                                 sd = 1)
```

```
hist(dist.normal.random); plot(dist.normal.random)
```



La diferencia entre las líneas está basado en diferencias entre *mayúsculas* y *minúsculas* lo cual diferencia la creación de las variables. Las funciones generadas con `rnorm()` son distribuciones `r(random)` y `norm(normal)` la primera es normal no estandar y la segunda normal estandar.

Jugando con Vectores II

Para acceder a un elemento del vector usamos `[]`.

```
weight <- c(33,45,78,77,45,89,48,75)
length(weight) # Dimensión del vector
```

```
## [1] 8
```

```
weight[4] #Ingresar al elemento 4.
```

```
## [1] 77
```

```
vector <- 1:20 #Creando sin función c()
altura <- c(1.70, 1.45, 1.54, 1.72, 1.85 ,1.45
IMC <- round(weight/(altura^2),1)
IMC
```

```
## [1] 11.4 21.4 32.9 26.0 13.1 42.3 15.0 23.1
```

Para hacer una formulación lógica debemos considerar la siguiente estructura:
`variable_estudio[(sentencia_logica)]`

```
vector <- c(1,5,2,6,4,5,74)
vector > 5
```

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

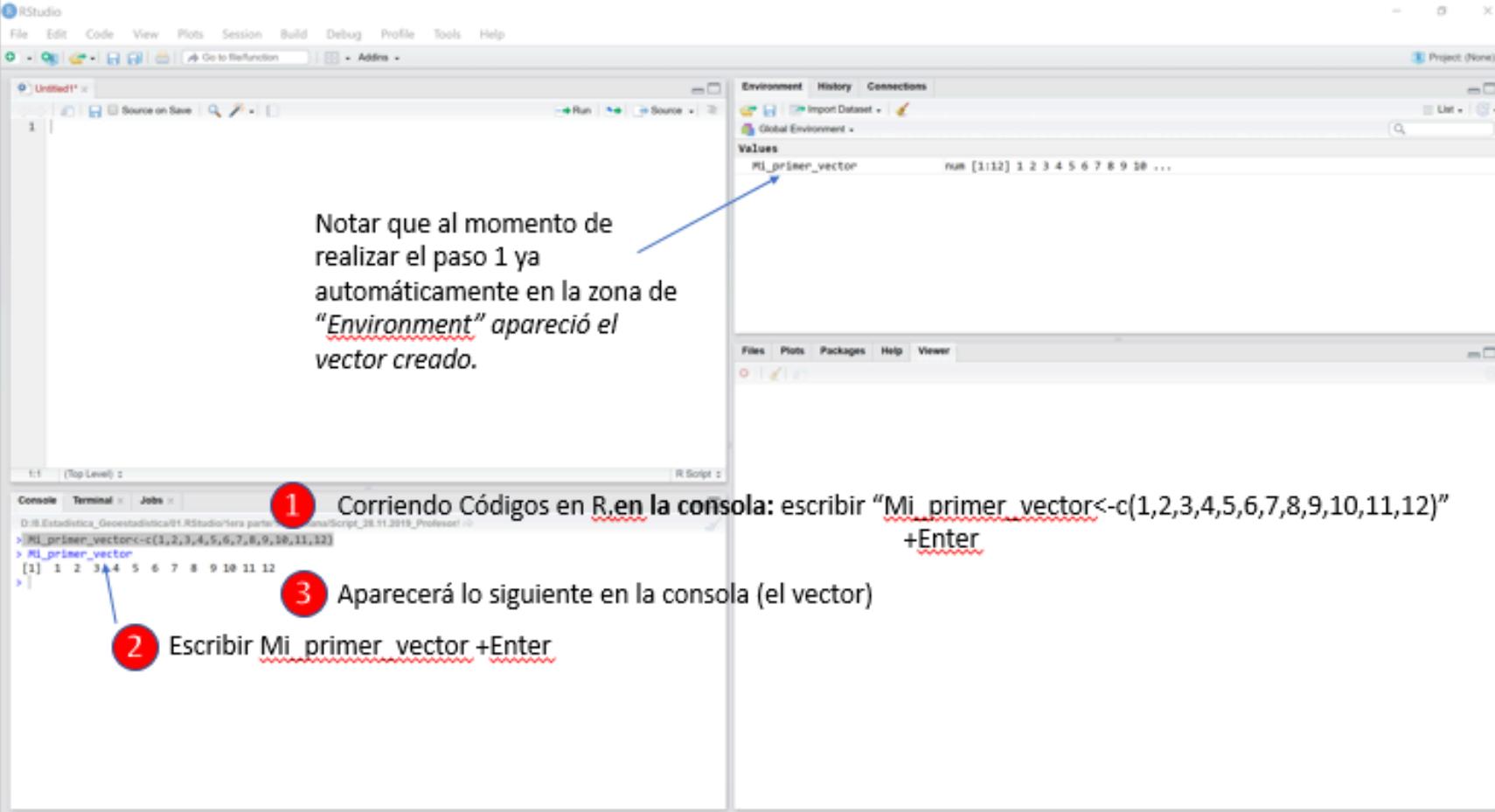
```
vector[(vector > 5)]
```

```
## [1] 6 74
```

```
vector[(vector > 1)&(vector < 7)]
```

```
## [1] 5 2 6 4 5
```

Consideraciones del entorno R en el Script



Notar que al momento de realizar el paso 1 ya automáticamente en la zona de “Environment” apareció el vector creado.

1 Corriendo Códigos en R en la consola: escribir “`Mi_primer_vector<-c(1,2,3,4,5,6,7,8,9,10,11,12)`” +Enter

2 Escribir `Mi_primer_vector` +Enter

3 Aparecerá lo siguiente en la consola (el vector)

```
Mi_primer_vector<-c(1,2,3,4,5,6,7,8,9,10,11,12)
> Mi_primer_vector
[1] 1 2 3 4 5 6 7 8 9 10 11 12
```

Consideraciones del entorno R en el Script

Tratar de no usar tildes, ni la ñ, entre dos palabras o más no dejar el espacio libre, ejemplo “*Mi priñer vector*” en cambio usar “*Mi_priner_vector*” debido a que se puede trabajar mejor en el software, además cabe mencionar que debemos **ser lo más simple al momento de realizar los códigos**, es decir no usar nombres extensos sino representativos, por ejemplo *Resultados Económicos del año 2018*, en cambio usar *RE_2018*.

Tener en claro que el paso más importante es la codificación completa del proyecto que vamos a trabajar para no tener problemas al momento de crear variables y relacionarlas. Además de las *esctructuras de las carpetas*.

Siendo un entorno dinámico tiene un enfoque mientras menos mejor, esto está en función de reducir largos procesos de códigos un simple ejemplo es escribir `vector <- c(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15)` en cambio se puede usar **`vector<-c(1:15)`** buscando siempre la manera más simple del código, tener en cuenta que siempre se puede sino se crea.

Funciones Básicas

Las funciones básicas son :

```
#Session
dir()
sessionInfo()
date()
Sys.time()
help()
?
#Objetos:
str()
colnames()
rownames()
dim() #or length()
View()
```

```
# Estadísticas
range()
summary()
sum()
min()
max()
mean()
median()
sd()
scale()
quantile()
var()
cov()
cor()
```

Datos Especiales

- NA (NOT AVAILABLE).

Generalmente nuestra base de datos posee siempre datos que **no son completamente conocidos o son vacíos** por ejemplo: en Excel una celda vacía la cual se reconocerá como un *dato no existente*, posiblemente se pueda usar una función para omitir estos datos o **tratar esos datos faltantes de forma especial** (por ejemplo: reemplazarlo por la media aritmética, la mediana, medias cortadas, método ROS, entre otros siempre y cuando sea la variable faltante numérica), al aplicar cualquier función de las básicas enseñadas el código corre según:

`na.rm = FALSE` (que indica que no se eliminan los NA) que se usa cuando existe al menos un dato faltante, si nosotros cambiamos la opción por default de la función y aplicamos `na.rm =TRUE` (indicamos que sólo trabaje con datos válidos).

```
weight<-c(33,45,78,77,45,89,48,75,NA)  #Para ejecutar aplicamos (Ctrl+Enter)
mean(weight)                            #Calculamos la media
```

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

```
mean(weight , na.rm =TRUE)            #Media sin considerar not avalaible (NA)
```

```
## [1] 61.25
```

```
x <- NA          #para ejecutar aplicamos (Ctrl+Enter)
x+1             #calculamos lo pedido (¿Qué pasó?)
```

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

```
y <- c(x,3,5,x)      #calculamos media aritmética de y.
mean(y)
```

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

- Inf (Infinito) y NaN (Not a number)

Al realizar cálculos en Rstudio en determinadas ocasiones puede resultar un valor *infinito positivo (+Inf)* o *infinito negativo (-Inf)* adicionalmente R permite hacer cálculos aproximados con este número, aunque existen condiciones en las cuales las expresiones son expresadas como *NaN's (not a number)*.

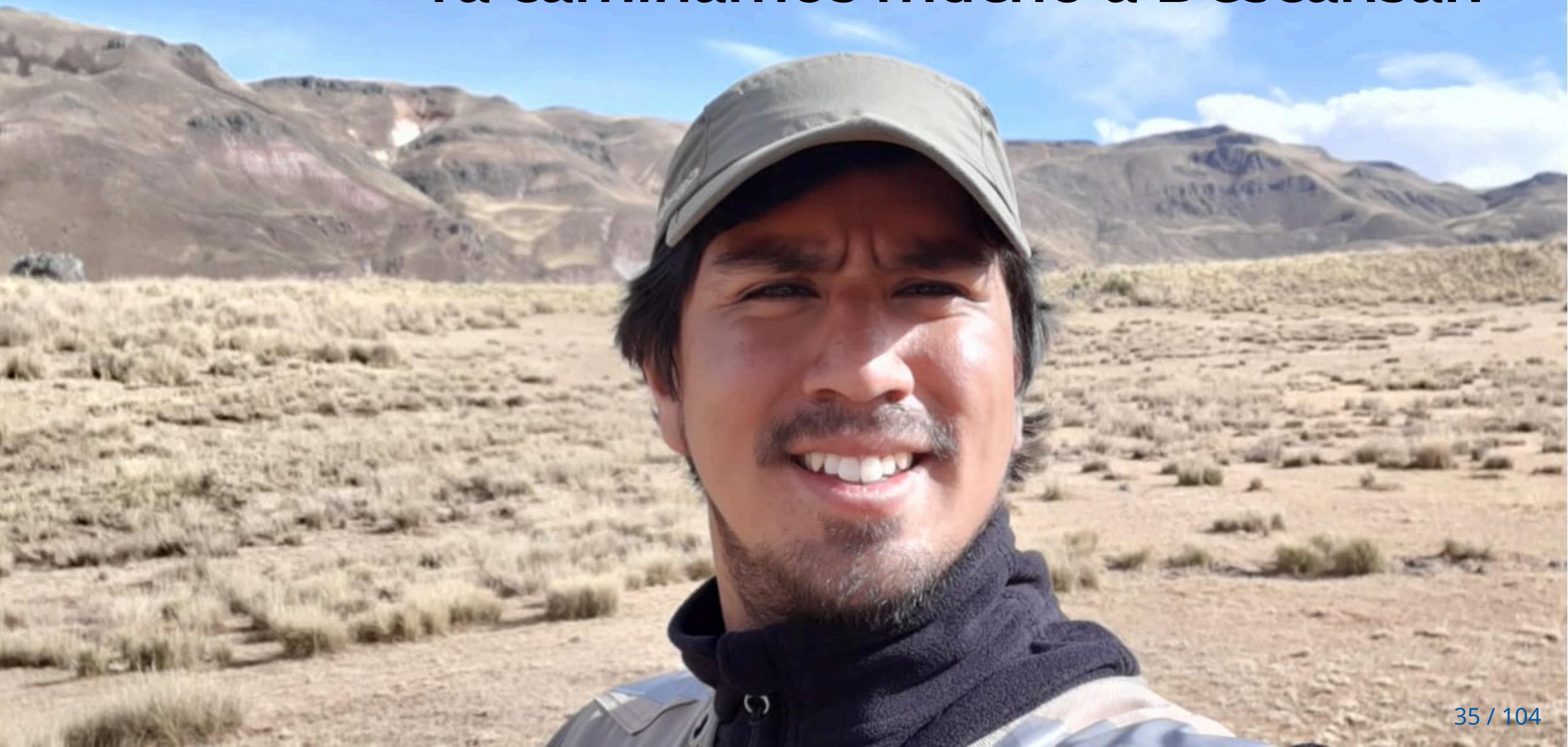
Se debe tener en cuenta que los números complejos el R los trabaja con normalidad, sin embargo se debe citar estos números, por ejemplo: $22 + 3i$, $i + 45$, i^2 , $33 + 154i$, $0 + 2i$.

#Ejecutemos los siguientes códigos en R:

```
log(10/0)+log(0/10)
sqrt(-24)
sqrt(-24+2i)
y<-23/0
exp(-y)
exp(y)-exp(y)
```

!R STUDIO O R REALIZA APROXIMACIONES DE LOS VALORES CON TENDENCIA DE LÍMITES. VARIANZA(NO!)
CuasiVarianza(SI!!).

Ya caminamos mucho a Descansar!



MATRICES

Concepto de Matrices

Las matrices son un *arreglo bidimensional* el cual es un objeto en R, los números son interpretados como vectores de dos componentes (un vector define el número de columnas y el otro el número de filas) los elementos deben ser del *mismo tipo*. La forma de operar es con la función `matrix(data, nrow, ncol, byrow =FALSE)`.

Argumentos Básicos:

data: Colección de elementos que R arreglará. nrow: Número de filas ncol: Número de columnas byrow: Las filas son llenadas de izquierda a derecha con el default.

- Debemos considerar las funciones `is.matrix()` y `as.matrix()` que comprueban o fuerzan el carácter de matriz de un objeto respectivamente. Generalmente con el atributo `dimnames()` damos nombre a las filas y columnas de una matriz.
- Además R tiende a realizar todo sencillo, es decir seleccionar una columna, el objeto resultante es un vector y no una matriz, en el caso se deseé mantener las dimensiones añadir `drop = F`.
- Tenemos que considerar el reciclado que sucede al momento de añadir el vector a una matriz, este vector se extiende repitiendo sus elementos hasta alcanzar el tamaño de la matriz.

Ejemplo de Matrices

Crearemos una matriz y analizaremos

```
matrix_01 <- matrix(1:12, byrow= TRUE, nrow=6)      #Para ejecutar aplicamos (Ctrl+Enter)  
matrix_01                                         #Visualizar la matriz (¿Qué vemos?)
```

```
##      [,1] [,2]  
## [1,]    1    2  
## [2,]    3    4  
## [3,]    5    6  
## [4,]    7    8  
## [5,]    9   10  
## [6,]   11   12
```

```
dim(matrix)                                         #Ver la dimensión de la matriz
```

```
## NULL
```

Crear una matriz con cinco filas que contienen los números del 1 al 10 y sea por columnas. Ver matriz y calcular la dimensión. Buscar otra forma de hacerlo diferente a la primera. Discutir.

```
y <- c(1:8)                                #Para ejecutar aplicamos (Ctrl+Enter)
dim(y) <- c(2,4)                            #Asignamos la dimensión de la matriz
dimnames(y) <- list(c("F1","F2"), c("C1","C2","C3","C4")) #Colocamos nombres a la matriz
covid_19 <- matrix(1:12, ncol=3, dimnames =list( c("Desobedece mucho",
                                                 "Desobedece", "Obedece",
                                                 "Es fiel"), letters[1:3]))
covid_19
```

```
##          a b c
## Desobedece mucho 1 5 9
## Desobedece      2 6 10
## Obedece        3 7 11
## Es fiel        4 8 12
```

#Creamos la matriz con filas de comportamiento

Tenemos opciones de entrar a los elementos o grupos de elementos de la matriz (también se puede hacer con el vector) usando `[]` (corchetes).

Por ejemplo:

- `matrix[n,m]` selecciona los elementos de la fila n y la columna m .
- `matrix[,m]` selecciona todos los elementos de la columna m .
- `matrix[n,]` selecciona todos los elementos de la fila n .

Además si realizamos lo siguiente: `matrix[1:3,2:3]` accedemos a la data de las *filas 1,2,3 y de las columnas 2 y 3*. También se pueden elegir los nombres como un vector.

```
covid_19 [1,1]
```

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

```
covid_19 [ , c(2,3)]
```

```
##          b   c
## Desobedece mucho 5  9
## Desobedece      6 10
## Obedece        7 11
## Es fiel        8 12
```

```
covid_19 [ , c(-1,-3), drop=F]
```

```
##          b
## Desobedece mucho 5
## Desobedece      6
## Obedece        7
## Es fiel        8
```

```
covid_20 <- matrix(rep(c(T,F),6),4,3)
covid_19[covid_20]
```

```
## [1] 1 3 5 7 9 11
```

```
m <- matrix(1:8, 2, 4)
diag(m) # diag(m) extrae la diagonal de la ma
```

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

```
m[,3]
```

```
## [1] 5 6
```

```
m[,3, drop=FALSE]
```

```
## [,1]
## [1,] 5
## [2,] 6
```

```
m[1, 1:2]
```

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

```
w <- matrix(1:8, 4, 2)
w[3, ]
```

```
## [1] 3 7
```

```
ncol(m);ncol(w);nrow(m);nrow(w) # ncol(m) número
```

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

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

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

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

```
m
```

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

```
t(m)
```

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

```

#Creamos la matriz Economia
Economia<-matrix(1:10, byrow=TRUE,nrow=5)
View(Economia)
# cbind() y rbind(): La función cbind() concatena columnas
# A la matriz Economia concatenaremos con el vector Geología
Geologia<-c(1:5)
Mina <- cbind(Economia,Geologia)
dim(Mina)

```

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

```

#Creamos la matriz denominada mama
mama <- matrix(13:24, byrow=FALSE, ncol=3)
#Creamos la matriz ama
ama <- matrix(1:12,byrow=FALSE,ncol=3)
#Unimos las matrices mama y ama:
mama_ama <- cbind(mama,ama)
#Probar Mina y mama, ¿cómo se haría?
Mina_mama <- rbind(Mina,mama)

```

```

#Continuando con la matriz m.
cbind(1,m) ; cbind(m,1) ; rbind(1,m); rbind(m,1)

```

```

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

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

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

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

```

```

#Háganlo ustedes con w.
# cbind(1:3,1:6) #Probemos con esto.
# apply(w,1,sum) #¿Que será?

```

Otras Funciones Basicas

sort() - seq() - rep()

Secuencia : seq()

Función `seq(from=1, to=1, by=((to-from)/(length.out-1)), lengout.out=NULL, along.with=NULL, ...)`

Genera secuencias regulares en los cuales los argumentos son:

`From`, `to`: el comienzo y el final (máximo) valores de la secuencia. De *length* 1 a menos solo *from* suministra como un no nombrado argumento.

`by`: número incremento de la secuencia.

`length-out`: deseada longitud de la secuencia. Un número nonegativo, el cual para *seq* y *seq.int* será redondeado a un fraccional.

`along.with`: tomar la longitud desde la longitud de este argumento.

```
seq(0, 1, length.out = 11)
```

```
## [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

```
seq(stats:::rnorm(20)) # effectively 'along'
```

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

```
seq(1, 9, by = 2)      # matches 'end'
```

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

```
seq(1, 9, by = pi)      # stays below 'end'
```

```
## [1] 1.000000 4.141593 7.283185
```

```
seq(1, 6, by = 3)
```

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

```
seq(1.575, 5.125, by = 0.05)
```

```
## [1] 1.575 1.625 1.675 1.725 1.775 1.825 1.875 1.9
```

```
## [13] 2.175 2.225 2.275 2.325 2.375 2.425 2.475 2.5
```

```
## [25] 2.775 2.825 2.875 2.925 2.975 3.025 3.075 3.1
```

```
## [37] 3.375 3.425 3.475 3.525 3.575 3.625 3.675 3.7
```

```
## [49] 3.975 4.025 4.075 4.125 4.175 4.225 4.275 4.3
```

```
## [61] 4.575 4.625 4.675 4.725 4.775 4.825 4.875 4.9
```

```
seq(17) # same as 1:17, or even better seq_len
```

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

```
seq(2, 4, by = 0.02)
```

```
## [1] 2.00 2.02 2.04 2.06 2.08 2.10 2.12 2.14 2.16
```

```
## [16] 2.30 2.32 2.34 2.36 2.38 2.40 2.42 2.44 2.46
```

```
## [31] 2.60 2.62 2.64 2.66 2.68 2.70 2.72 2.74 2.76
```

```
## [46] 2.90 2.92 2.94 2.96 2.98 3.00 3.02 3.04 3.06
```

```
## [61] 3.20 3.22 3.24 3.26 3.28 3.30 3.32 3.34 3.36
```

```
## [76] 3.50 3.52 3.54 3.56 3.58 3.60 3.62 3.64 3.66
```

```
## [91] 3.80 3.82 3.84 3.86 3.88 3.90 3.92 3.94 3.96
```

```
seq(7, 14, length.out = 15)
```

```
## [1] 7.0 7.5 8.0 8.5 9.0 9.5 10.0 10.5 11.0
```

Repetición: rep()

Función rep(x, ...). Replica los valores en x, los argumentos son:

x = un vector(de cualquier modo incluyendo en una lista) o un factor or (para *rep* solo). ... = otros argumentos para ser pasados para o desde otros métodos, por ejemplo:

```
times = un vector de valor entero dado el número de veces para repetir cada elemento o repetir el ve
length.out = Entero no negativo. El deseado de la longitud de la salida del vector. Otros *inputs* o
each = Entero no negativo. Cada elemento de x es repetido cada vez.. Otros *Inputs* deberán ser forz
```

```
rep(1:4, 2) ; rep(1:4, each = 2) ; rep(1:4, c(2,2,2,2))
```

```
## [1] 1 2 3 4 1 2 3 4
## [1] 1 1 2 2 3 3 4 4
## [1] 1 1 2 2 3 3 4 4
```

```
rep(1:4, c(2,1,2,1))
```

```
## [1] 1 1 2 3 3 4
```

```
rep(1:4, each = 2, len = 4)
```

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

```
rep(1:4, each = 2, len = 10)
```

```
## [1] 1 1 2 2 3 3 4 4 1 1
```

```
rep(1:4, each = 2, times = 3)
```

```
## [1] 1 1 2 2 3 3 4 4 1 1 2 2 3 3 4 4 1 1 2 2 3 3 4
```

```
rep(1, 40*(1-.8))
```

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

```
rep(1, 40*(1-.8)+1e-7)
```

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

```
fred <- list(happy = 1:10, name = "squash")  
rep(fred, 2)
```

```
## $happy
```

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

```
## $name
```

```
## [1] "squash"
```

```
##
```

```
## $happy
```

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

```
##
```

```
## $name
```

```
## [1] "squash"
```

```
x <- .leap.seconds[1:3]  
rep(x, 2)
```

```
## [1] "1972-07-01 GMT" "1973-01-01 GMT" "1974-01-01  
## [5] "1973-01-01 GMT" "1974-01-01 GMT"
```

```
rep(as.POSIXlt(x), rep(2, 3))
```

```
## [1] "1972-07-01 GMT" "1972-07-01 GMT" "1973-01-01  
## [5] "1974-01-01 GMT" "1974-01-01 GMT"
```

```
x <- factor(LETTERS[1:4]); names(x) <- letters[1:4]      ## factor nombrado
rep(x, 2)
```

```
## a b c d a b c d
## A B C D A B C D
## Levels: A B C D
```

```
rep(x, each = 2)
```

```
## a a b b c c d d
## A A B B C C D D
## Levels: A B C D
```

```
rep.int(x, 2) # no names
```

```
## a b c d a b c d
## A B C D A B C D
## Levels: A B C D
```

```
rep_len(x, 10)
```

```
## a b c d a b c d a b
## A B C D A B C D A B
## Levels: A B C D
```

Ordenar: sort()

Función `sort(x, decreasing=FALSE, ...)`. Replica los valores en `x`, en los cuales los argumentos son:

`x`: para *sort* un objeto de R (vector) con una clase o un número, complejo, carácter o lógico, o un factor.

`decreasing`: lógico. Deberá el *sort* incrementarse o disminuir? Para el método `radix`, esto puede ser un vector de igual para el número de argumentos dentro.. para otros métodos debe ser `length` uno.

`...`: Argumentos para pasara o desde métodos o (or defecto métodos u objetos sin ninguna clase) para `sort.int`.

```
sort(c(1,2,4,5,7,45,78,7,8,4,7,5,4,7,4,4,44))
```

```
## [1] 1 2 4 4 4 4 5 5 7 7 7 8 44 45 78
```

```
sort(rev(c(1,4,7,8,5,2,3,6,9)))
```

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

```
rep(c(sort(c(1,2,3)), 2))
```

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

Sort y Rep:

```
sort(rep(1,4,by=0.5))
```

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

```
sort(rep(seq(1,25,by=3),3))
```

```
## [1] 1 1 1 4 4 4 7 7 7 10 10 10 13 13 13 16 16 16 19 19 19 22 22 22 25
## [26] 25 25
```

```
rep(c(2,3,5), each=5)
```

```
## [1] 2 2 2 2 2 3 3 3 3 3 5 5 5 5 5
```

Dataframe

Dataframe

Un dataframe (objeto) es una lista de vectores los cuales tienen igual longitud. Una matriz a diferencia de los dataframe contiene solo un tipo de data, mientras que los dataframe aceptan todo tipo de data (*numerica, factor, character, etc.*)

Creando un Dataframe: Podemos crear un dataframe convirtiendo las variables a,b,c,d,e,... con la función `data.frame()`. Podemos cambiar el nombre de las columnas con `name()` y especificar de manera simple el nombre de las variables.

`data.frame(df, stringAsFactors = TRUE)`

`df`: Puede ser una matriz convertida como un dataframe o una colección de variables a unir. `stringAsFactors`: Convertir un *string* a *factor* por defecto.

Consultar funciones en `help->as.data.frame, is.data.frame, data.frame` y similares.

```

a <- c(10,20,30,40)                      #Creamos una variable a
b <- c("libro","lapicero","notas","portapincel") #Creamos una variable b
c <- c(TRUE, FALSE, TRUE, FALSE)           #Creamos una variable c
d <- c(7.4,4.5,8,22)                      #Creamos una variable d
#Uniremos todas las variables para crear el dataframe
df <- data.frame(a,b,c,d)                 #Creando el dataframe
#Como nos damos cuenta el dataframe no tiene nombres, crearemos estos:
names(df) <- c("ID", "Items", "Stock", "Precio") #Asignamos los nombres del dataframe.
str(df)                                     #Verificamos la estructura del dataframe

```

```

## 'data.frame': 4 obs. of 4 variables:
## $ ID    : num 10 20 30 40
## $ Items : chr "libro" "lapicero" "notas" "portapincel"
## $ Stock : logi TRUE FALSE TRUE FALSE
## $ Precio: num 7.4 4.5 8 22

```

```

#edit(data.frame(df)) #Editar online
#Trabajen en el siguiente ejemplo:
x <- c(2,7,8,9,2,4)
y <- matrix(x, ncol=3)
colnames(y) <- c("Y1", "Y2", "Y3") ; rownames(y) <- c("CasoA", "CasoB")
w <- as.data.frame(y)

```

```
#Comparemos lo siguiente (¿Cuál es la diferencia?)  
str(y)
```

```
##  num [1:2, 1:3] 2 7 8 9 2 4  
##  - attr(*, "dimnames")=List of 2  
##    ..$ : chr [1:2] "CasoA" "CasoB"  
##    ..$ : chr [1:3] "Y1" "Y2" "Y3"
```

```
str(w)
```

```
## 'data.frame': 2 obs. of 3 variables:  
## $ Y1: num 2 7  
## $ Y2: num 8 9  
## $ Y3: num 2 4
```

Moverse en el dataframe: Es posible moverse dentro de los valores del *dataframe* o en los rangos que se desea, similar a la matriz. Debemos seleccionar las filas y columnas y retornar estos valores dentro de corchetes por el nombre *dataframe*, es decir **df[A,B]** donde A representa las filas y B las columnas.

```
#Probemos con lo siguiente  
df[1,2]
```

#Analicemos que está pasando!!

```
## [1] "libro"
```

#Hagamos lo siguiente

```
df[1:2, ] #Analizar
```

```
##   ID   Items Stock Precio
## 1 10    libro  TRUE    7.4
## 2 20  lapicero FALSE    4.5
```

```
df [ , 1] #Analizar
```

```
## [1] 10 20 30 40
```

```
df[1:3,3:4] #Analizar
```

```
##   Stock Precio
## 1  TRUE    7.4
## 2 FALSE    4.5
## 3  TRUE    8.0
```

#Ahora esta forma

```
df[ , c("ID","Items")]
```

```
##   ID   Items
## 1 10    libro
## 2 20  lapicero
## 3 30      notas
## 4 40 portapincel
```

Agregar Columnas al Dataframe: Podemos agregar columnas solo necesitamos adicionar el simbol dolar.

```
#Creamos un nuevo vector
w$Y4<-c("si","no")
w
```

```
##      Y1 Y2 Y3 Y4
## CasoA  2  8  2 si
## CasoB  7  9  4 no
```

```
cantidad <- c(10,35,40,45)
# A gregamos cantidad al df dataframe
df$cantidad <- cantidad #¿Qué pasa?
df
```

```
##   ID      Items Stock Precio cantidad
## 1 10      libro  TRUE  7.4      10
## 2 20    lapicero FALSE  4.5      35
## 3 30      notas  TRUE  8.0      40
## 4 40 portapincel FALSE 22.0      45
```

Seleccionar una columna del Dataframe:

Algunas veces necesitamos alamacenar una columna de un dataframe para usarla en el futuro o mejorar operaciones en la columna. Podemos usar el símbolo `$` para seleccionar una columna del dataframe.

```
df$ID
```

```
## [1] 10 20 30 40
```

```
df$Y1
```

```
## NULL
```

Subset de Data

A veces necesitamos trabajar con ciertos datos de nuestro *dataframe* no con todo, por eso es útil realizar un **subset** de nuestra data para filtrar siguiend ciertas condiciones. Para ello usamos la funcipon `subset()`.

$subset(x, condition)$

Argumentos:

`x`: Dataframe usada para llevar a cabo el subset.

`condition`: Definir la declaracion condicional.

```
subset(df, subset = Precio>5)
```

```
##   ID      Items Stock  Precio cantidad
## 1 10     libro  TRUE   7.4      10
## 3 30     notas  TRUE   8.0      40
## 4 40 portapincel FALSE  22.0      45
```

```
subset(df, subset = Stock==TRUE)
```

```
##   ID      Items Stock  Precio cantidad
## 1 10     libro  TRUE   7.4      10
## 3 30     notas  TRUE   8.0      40
```

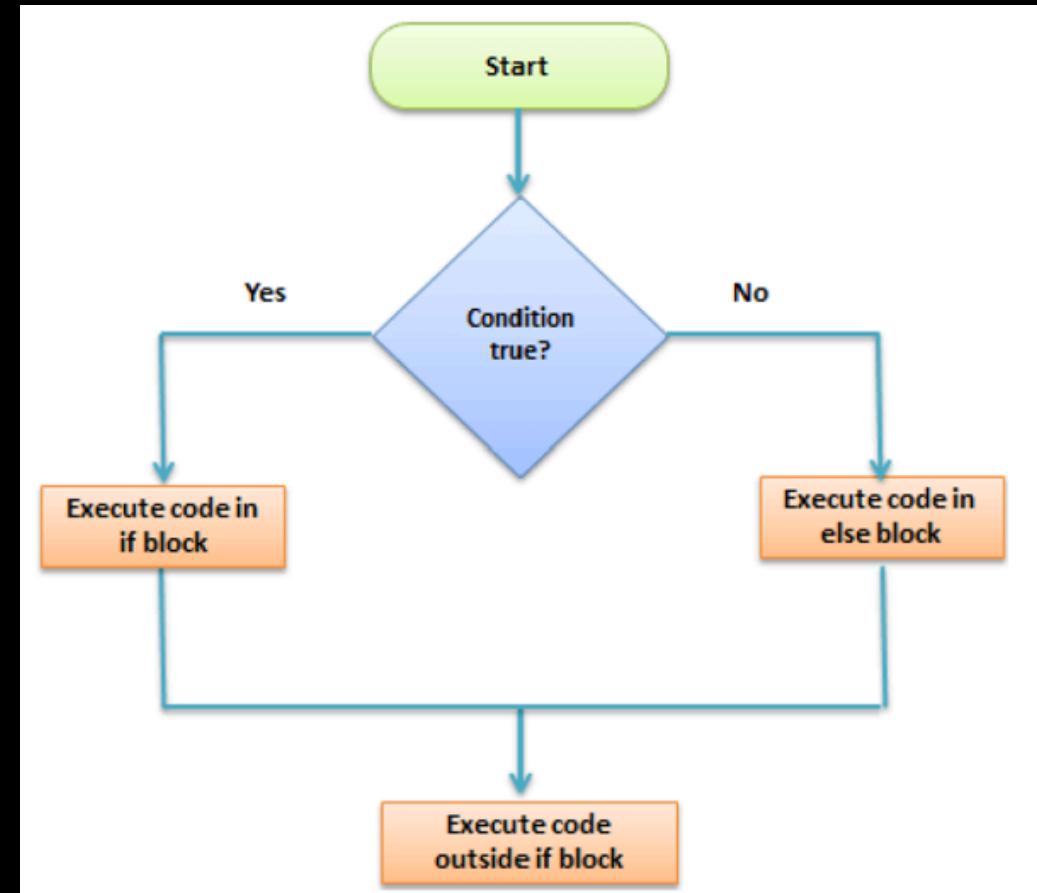
No entraremos más en detalle puesto que más adelante conoceremos el asombroso funcionamiento del `Tidyverse`

Estructuras de Decisión y Repetición

IF, ELSE, ELSE IF

Un if-else statement es una gran herramienta para el desarrollo de traer una salida al inicio con una condición. En R, la sintaxis es:

```
#La sentencia tiene la siguiente estructura:  
if (condition) {  
  Expr1 (bloque de código)  
}  
  
if (condition) {  
  Expr1 (bloque de código)  
} else {  
  Expr2 (continua con el resto del código)  
}  
  
if (condition) {  
  Expr1 (bloque de código)  
} else if (condition2) {  
  Expr2 (otro bloque de código)  
} else {  
  Expr3(otro bloque de código)  
}
```



#Realizaremos los siguientes ejemplos (seguir la correcta estructura):

```
cantidad <-50
if (cantidad >30) {
  print ("Tú vendiste mucho!")
} else {
  print ("No fue suficiente hoy")
}
```

```
## [1] "Tú vendiste mucho!"
```

```
cantidad <-80
if (cantidad<50) {
  print("No fue suficiente hoy")
} else if (cantidad>50 & cantidad<=60) {
  print ("Promedio de venta")
} else {
  print("¡Fue un dia maravilloso!")
}
```

```
## [1] "¡Fue un dia maravilloso!"
```

#Realizaremos los siguientes ejemplos (seguir la correcta estructura):

```
x <- c("yo", "engaño", "con el curso")
if ("Con el curso" %in% x) {
  print("Con el curso no me va bien")
} else if ("con el curso" %in% x) {
  print ("Con el curso me va bien")
} else {
  print("Yo fui engañado con este curso")
}
```

```
## [1] "Con el curso me va bien"
```

```
p <- runif (10,0,20)
Vec_Booleano <- p>=14
NumAprob1 <- length(p)-5
NumAprob2 <- length(Vec_Booleano[Vec_Booleano==TRUE])
which(Vec_Booleano)
```

```
## [1] 2 6 8
```

```
if (NumAprob1>0){
  p[which(Vec_Booleano)]
}
```

```
## [1] 17.57340 17.10012 17.98365
```

```
#Consultar el help de R:  
help("print")
```

```
## starting httpd help server ... done
```

```
help("%in%")
```

#Realizaremos los siguientes ejemplos (seguir la correcta estructura):

```
p <- runif(100000,0,30)  
Vec_Booleano2 <- p>=15  
NumAprob1 <-sum(Vec_Booleano2)  
NumAprob2 <-length(Vec_Booleano2[Vec_Booleano2==TRUE])  
system.time(sum(Vec_Booleano2))
```

```
##      user  system elapsed  
##        0        0        0
```

```
system.time(length(Vec_Booleano2[Vec_Booleano2==TRUE]))
```

```
##      user  system elapsed  
##        0        0        0
```

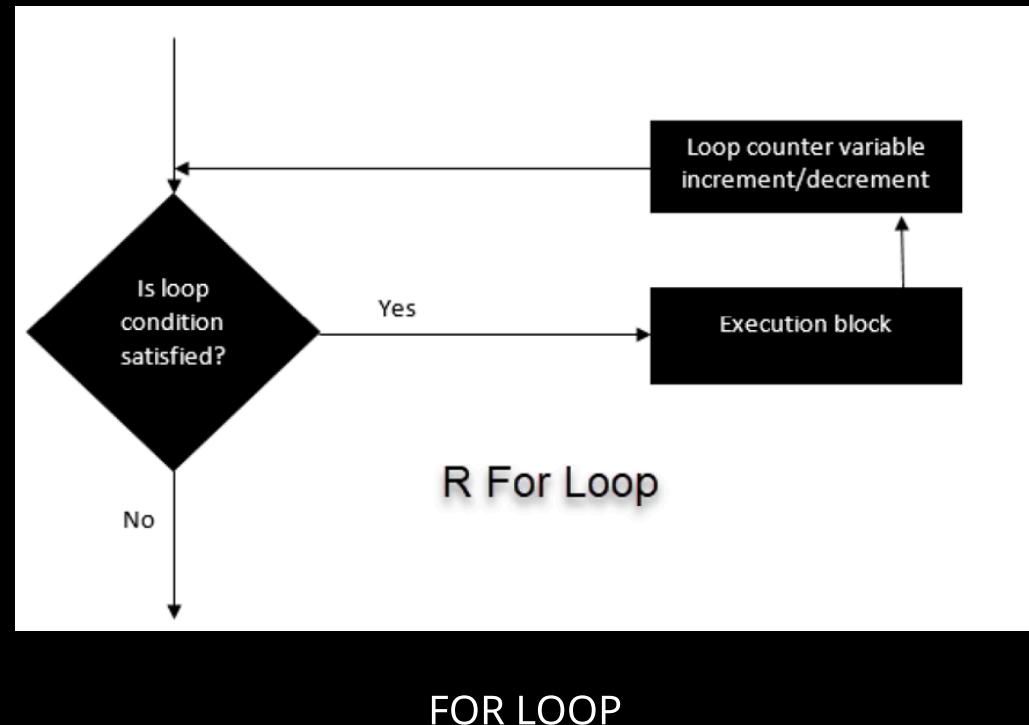
```
#El Ejemplo final:.
#Score <- as.integer(readline("Ingresa tu nota : "))
Score <- 20
if(Score >= 19){
print("Excelente!!")
print("Llevarás el curso tratamiento estadístico GRATIS!!")
} else if (Score >= 16){
print("Felicidades!!")
print("Llevarás el curso tratamiento estadístico pagando 5 soles!!")
} else if (Score>=14){
print("Felicidades!!")
print("Llevarás el curso tratamiento estadístico pagando 50 soles!!")
} else {
print("No has sido elegido ☹!!")
print("Lo sentimos paga el precio completo 5000 soles")
}
```

```
## [1] "Excelente!!"
## [1] "Llevarás el curso tratamiento estadístico GRATIS!!"
```

FOR LOOP

Un **for loop** es muy útil cuando necesitamos itinerar sobre una lista de elementos o un rango de números. Loops pueden usarse para itinerar sobre una lista, dataframe, vector, matriz o cualquier otro objeto.

```
for ( i in vector) {  
  Expr1 (bloque de código)  
}  
i: <variable>  
Vector: <objeto iterable>  
  
#Ejemplo sencillo:  
for ( i in 1:5) {  
  print(i)  
}  
y <- c(1:5)  
for ( x in 1:5) {  
  w<-y**x  
}
```



```
fruta<-c("Manzana","Naranja","Melocotón","Plátano")
for (i in fruta){
print(i)
}
```

```
## [1] "Manzana"
## [1] "Naranja"
## [1] "Melocotón"
## [1] "Plátano"
```

```
fruit <-list(Caja = c("Manzana","Naranja","Melocotón","Plátano"))
for (p in fruit){
print(p)
}
```

```
## [1] "Manzana"    "Naranja"    "Melocotón"  "Plátano"
## [1] 10 12 15
## [1] FALSE
```

```
lista <-c()
for (i in seq(1,4,by=1)){
lista[[i]]<-i*i
}
print(lista)
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 4
##
## [[3]]
## [1] 9
##
## [[4]]
## [1] 16
```

El uso de *for loop* es muy valioso cuando se llevan tareas de **machine learning**. Después que tenemos un modelo de entrenamiento, necesitamos regularizar el modelo para el ajuste. Esta regularización es una tarea muy tediosa porque necesitamos encontrar el valor para el cual la pérdida de la función sea mínima. Para ayudarnos a detectar estos valores, podemos hacer uso de *for loop* para itinerar sobre un rango de valores y definir el mejor candidato.

#Otros ejemplos:

```
matriz<-matrix(data=seq(9,20,by=1), nrow=6, ncol=2)
for (r in 1:nrow(matriz)) {
  for (c in 1:ncol(matriz)) {
    print(paste("Row",r,"and column",c, "have values of", matriz[r,c]))}}
```

```
## [1] "Row 1 and column 1 have values of 9"
## [1] "Row 1 and column 2 have values of 15"
## [1] "Row 2 and column 1 have values of 10"
## [1] "Row 2 and column 2 have values of 16"
## [1] "Row 3 and column 1 have values of 11"
## [1] "Row 3 and column 2 have values of 17"
## [1] "Row 4 and column 1 have values of 12"
## [1] "Row 4 and column 2 have values of 18"
## [1] "Row 5 and column 1 have values of 13"
## [1] "Row 5 and column 2 have values of 19"
## [1] "Row 6 and column 1 have values of 14"
## [1] "Row 6 and column 2 have values of 20"
```

```
print (paste ("The year is", 2010))
```

```
## [1] "The year is 2010"
```

```
print (paste ("The year is", 2011))
```

```
## [1] "The year is 2011"
```

```
print (paste ("The year is", 2012))
```

```
## [1] "The year is 2012"
```

```
print (paste ("The year is", 2013))
```

```
## [1] "The year is 2013"
```

```
print (paste ("The year is", 2014))
```

```
## [1] "The year is 2014"
```

```
print (paste ("The year is", 2015))
```

```
## [1] "The year is 2015"
```

```
for (year in c(2010,2011,2012,2013,2014,2015)) {  
  print (paste ("The year is", year))  
}
```

```
## [1] "The year is 2010"  
## [1] "The year is 2011"  
## [1] "The year is 2012"  
## [1] "The year is 2013"  
## [1] "The year is 2014"  
## [1] "The year is 2015"
```

```
for (year in 2010:2015){  
  print (paste ("The year is", year))  
}
```

```
## [1] "The year is 2010"  
## [1] "The year is 2011"  
## [1] "The year is 2012"  
## [1] "The year is 2013"  
## [1] "The year is 2014"  
## [1] "The year is 2015"
```

El uso de for loop es ayuda al principio de lenguaje de programación : **DRY (Do not Repeat Yourself)** con lo cual el código se hace más eficiente. ¿Cómo leemos las líneas de for loop?

Consultar el help de R:

```
help("paste")
```

```
#Otros ejemplos:  
x <- c("a", "b", "c", "d")  
seq_along(x)
```

Consultar el help de R:

```
## [1] 1 2 3 4
```

```
help("seq_along")
```

```
for (i in seq_along(x)){  
  print(x[i])  
}
```

```
## [1] "a"  
## [1] "b"  
## [1] "c"  
## [1] "d"
```

```
x <- matrix(1:6, 2, 3)  
for (i in seq_len(nrow(x))){  
  for (j in seq_len(ncol(x))){  
    print (x[i,j])  
  }  
}
```

```
## [1] 1  
## [1] 3  
## [1] 5  
## [1] 2  
## [1] 4  
## [1] 6
```

WHILE

Un **loop** que es para mantener corriendo el código desde que la condición es satisfecha, en otras palabras, si es verdadera la condición se entra al cuerpo del bucle.

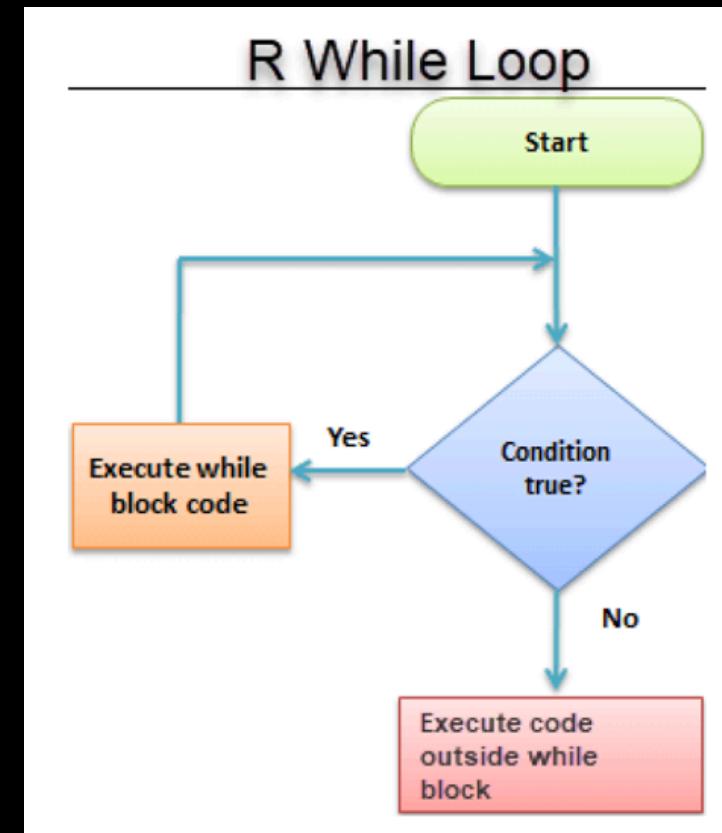
#La sentencia tiene la siguiente estructura:

```
1 while ( <condicion> ) {  
2   Expr1 (bloque de código)  
3 }
```

#Ejemplo sencillo:

```
1 count<-0  
2 while (count<10){  
3   print(count)  
4   count<-count+1  
5 }
```

```
1 Comenzar<-1  
2 while (Comenzar<=10){  
3   cat("Este es un loop número", Comenzar)  
4   Comenzar<-Comenzar+1  
5   print(Comenzar)  
6 }
```



```
set.seed(123)
stock <- 50
precio <- 50
loop <- 1
while(precio > 45){
  precio<-stock+sample(-10:10, 1)
  loop <- loop + 1
  print(loop)
}
```

Consultar el help de R:

```
help("set.seed()")
help("cat()")
```

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

```
cat("it took",loop,"loop before we short the Price")
```

```
## it took 5 loop before we short the Price. The lowest Price is 42
```

SWITCH

Este comando permite **ejecutar un bloque de código distinto** en función del valor de una variable.

#La sentencia tiene la siguiente estructura:

```
1 switch ( <Expr> ,  
2 <valor_1>{  
3 # código  
4 },  
5 ...  
6 <valor_n>{  
7 #codigo  
8 }  
9 ) #Termina switch
```

#Ejemplo sencillo:

```
x <- 1:10  
type <- "mean"  
switch(type,  
mean = mean(x),  
median = median(x),  
sd = sd(x)  
)  
## [1] 5.5
```

```
x <- as.integer(2)
z <- switch(x,1,2,3,4,5)
z
```

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

```
x <- 3.5
z <- switch(x,1,2,3,4,5)
z
```

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

```
y <- rnorm(5)
x <- "sd"
z <- switch(x,"mean"= mean(y),"median"= median(y),"variance"= var(y),"sd"=sd(y))
z
```

```
## [1] 1.048981
```

```
x <- "median"
z <- switch(x,"mean"= mean(y), "median"= median(y),"variance"= var(y), "sd"=sd(y))
z
```

```
## [1] -0.108966
```

REPEAT

La estructura **repeat** ejecuta un bucle infinitamente. En general no es utilizada para realizar análisis, sino cuando se realiza programación. La única forma de terminar con el bucle es llamando dentro de esta a la función **break**.

#La sentencia tiene la siguiente estructura:

```
1 repeat(<condición>){  
2   #código  
3   ...  
4 }
```

#Ejemplo sencillo:

```
count<-0  
repeat{  
  print(count)  
  count <- count+1  
  if(count==2) break  
}
```

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

Next/Break

La función `next` es utilizada para terminar un ciclo del bucle en ejecución y pasar al siguiente. Por ejemplo, si estamos dentro de un *bucle FOR* lo que sucedería al momento de ejecutar la función `next` es que se salta directo al siguiente elemento sobre el que está iterando. Por otro lado, la función `break` es usada para detener el bucle y salir de él inmediatamente.

```
#Imprimir en la pantalla los números del 2,4,6 y 8
for( i in 1:10){
  if( i %%2 == 0) next
  print (i)
}
```

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

```
#Pondremos el ejemplo para imprimir los números impares entre 1 y 10, pero que los pares no se vean
for( i in 1:10){
  if (!i %% 2){
    next
  }
  print (i)
}
```

```
# Contar los números pares de un vector
x <- c(2,5,3,9,8,11,6)
count <- 0
for (val in x){
  if (val %% 2 ==0)
    count = count + 1
}
print(count)
```

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

```
#Imprimir nuestra famosa tabla de multiplicar
num = as.integer(readline("Ingrese un numero: "))
```

```
## Ingrese un numero:
```

```
for (i in 1:10){
  print(paste(num, "x", i, "=", num*i))
}
```

```
## [1] "NA x 1 = NA"
## [1] "NA x 2 = NA"
## [1] "NA x 3 = NA"
## [1] "NA x 4 = NA"
## [1] "NA x 5 = NA"
## [1] "NA x 6 = NA"
## [1] "NA x 7 = NA"
```

```
# Calculemos el factorial de un número entero
num = as.integer(readline("Ingrese un numero entero: "))
factorial = 1
if(num<0){
  print("El numero ingresado debe ser positivo")
} else if (num==0){
print("Por definición, el factorial de 1 es 0")
} else {
for (i in 1:num){
factorial = factorial * i
}
print(paste("El factorial de", num, "es", factorial))
}
```

```
# Verificador de números primos
num = as.integer(readline("Ingrese un numero : "))
flag=0
if(num>1){
  flag=1
  for(i in 2:(num-1)){
    if((num%%i) == 0){
      flag=0
      break
    }}}
if(num ==2) flag =1
if(flag ==1){
  print(paste(num, "es un numero primo"))
} else {
  print(paste(num, "No es un numero primo"))
}
# Probar con el número: 3,5,7,23,2305843009213693951
```



Volvemos en 3 limones!

Factores - Listas en R

Factores R

Factores son **variables en R** las cuales toman un número límite de diferentes valores; tales variables usualmente referidas como una *variable categórica*.

En los dataset, podemos distinguir dos tipos de variables categóricas: **categórica** y **continua**.

- En variables categóricas, los valores están limitados y usualmente en un grupo finito en particular.
- En variables continuas, sin embargo, pueden tomar cualquier valor, desde un entero a decimal. Por ejemplo. Podemos tener ingresos, precios de venta, etc.

Variables Categóricas: R almacena variables categóricas en un factor. Veamos el código siguiente para convertir un a variable categórica en un factor variable. Caracteres no son soportados en algoritmos de **machine learning**, y la única forma es convertirlos a string o en integer.

`factor(x = character(), levels, labels = levels, ordered = is.ordered(x))`

Argumentos:

`x` = Un vector de data. Necesita ser un string o integer, no decimal. `levels` = Un vector de posibles valores tomados por x. Este argumento es opcional. El valor por defecto es la única lista de ítems del vector. `labels` = Agregar una etiqueta a la data x. Por ejemplo, 1 puede tomar la etiqueta "hombre" mientras 0 la etiqueta "mujer". `ordered` = Determina si los niveles deberán ser ordenados.

```
vec_genero<-c("Male","Female","Female","Male","Male")      # Creamos un vector genero.  
class (vec_genero)                                     # Vemos la clase del vector.
```

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

```
factor_vec_genero<-factor(vec_genero)  
class(factor_vec_genero)
```

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

Es importante transformar un string en un factor cuando llevamos a cabo una tarea de machine learning. Una variable *categórica* puede ser dividida en una variable **nominal** categórica y variable **ordinal** categórica.

Variable Categórica Nominal: posee muchos valores pero el orden no importa. Por ejemplo, hombre y mujer categóricas variables no tienen un orden.

```
vector_color <- c("azul","rojo","verde","blanco","negro","amarillo") # Creamos un vector color.  
factor_color <- factor(vector_color)                                # Convertimos el vector a factor.  
factor_color #Revisar la salida (output) !
```

```
## [1] azul      rojo      verde     blanco    negro     amarillo  
## Levels: amarillo azul blanco negro rojo verde
```

Variable Categórica Ordinal: poseen un orden natural. Nosotros podemos especificar el orden , desde el menor a el mayor con order = TRUE y del mayor al menor con order = FALSE.

```
vec_dia <- c("evening", "morning", "afternoon", "midday", "midnight", "evening") # Creamos un vector
factor_dia <- c(vec_dia, order = TRUE, levels = c("morning", "midday", "afternoon", "evening", "midnight"))
factor_dia # Imprimir o correr la variable.
```

```
## "evening" "morning" "afternoon" "midday" "midnight" "evening"
## order      levels1      levels2      levels3      levels4      levels5
## "TRUE"      "morning"     "midday"     "afternoon"    "evening"     "midnight"
```

```
# Observar la salida (output) !
# R ordena los niveles desde "morning" a "midnight" como específicos en los niveles entre paréntesis.
```

Variables Continuas: La clase variable continua son valores por defecto en R. Ellas son almacenadas como numeric o integer. Podemos ver desde el dataset siguiente mtcars (dataset construido en R). Este reúne informacion de diferentes tipos de carros. Nosotros podemos importar usando mtcars y revisando la clase de la variable mpg, mile per gallon. Esto nos retorna un valor numérico, indicándonos una variable continua.

```
dataset <- mtcars
class(dataset)
```

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

Listas en R

Una lista es una gran herramienta para *almacenar variedad de tipos de objetos* en el orden esperado. Nosotros podemos incluir matrices, vectores, dataframe o listas. Imaginemos una lista como una maleta grande en la cual queremos poner muchos diferentes ítems. Cuando necesitamos usar un ítem, abrimos la maleta y lo usamos. Una lista es similar, podemos almacenar una colección de objetos y usarlos luego cuando sean necesarios.

Empezaremos creando una lista con la siguiente función:

list(element₁, ...)

Argumentos:

element_1 = almacenar cualquier tipo de objeto en R. ... = poner tantos objetos como se necesite. Cada objeto necesita estar separado por una coma.

Crearemos diferentes objetos, un vector, una matriz y un dataframe.

```
vec <- 1:5                      # Creamos un vector genero.  
mat <- matrix(1:10, ncol=5)       # Creamos una matriz.  
df  <- EuStockMarkets[1:10, ]     # Seleccionamos las 10 filas del dataset construido en R EuStockMarkets  
mi_lista <- list(vec, mat, df)    # Creamos la lista con los tres objetos.
```

Seleccionar Elementos de la Lista: Después de construir nuestra lista, podemos acceder muy fácil. Necesitamos usar el `[[index]]` seleccionar un elemento desde en una lista. El valor interno del doble corchete representa la posición del ítem en una lista que deseamos extraer. Por ejemplo, pondremos el número 2 dentro de los corchetes, R nos devolverá el segundo elemento listado.

```
mi_lista[[2]] # Elegimos el elemento 2 de la lista
```

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

```
#Probemos con el elemento 1 y 2.
```

Paquetería de Funciones

Definición de Funciones

Una función, en un ambiente de programación, es un **conjunto de instrucciones**. Un programador o practicante de programador construye una función evitando repetir la misma tarea, o reducir complejidad. Una función debe ser : (1) *escrita con para llevar a cabo una tarea específica*, (2) *puede o no puede incluir argumentos*, (3) *contener un cuerpo* y (4) *puede o no retornar más valores*. Un general aprovechamiento de las funciones es para usar argumentos como entradas (inputs), alimentar parte del cuerpo y finalmente retornar una salida (output). La escritura de la función es la siguiente:

#La sentencia tiene la siguiente estructura:

```
1 function (arglist) {  
    #Function body  
}
```

#Para escribir la función necesitamos nombre, argumentos y un cuerpo, ejemplo:

```
1 function.name <- function(arguments) {  
    computations on the arguments  
    some other code  
}
```

```
#Funciones de un argumento (Crearemos una función que se llame eleva al cuadrado)
square_function<-function(n)
{
  # compute the square fo integer "n"
  n^2
}
#Llamaremos a la función y asignaremos valore de 4
square_function(4)
```

```
## [1] 16
```

#Algo importante a considera es que nosotros no explicamos el tipo de variable por ende el programa

```
rm(square_function)
```

#Funciones en R definición:

```
es.par<-function(numero){
  residuo<-numero%%2
  if(residuo==0)
    return(TRUE)
    return(FALSE)
}
```

#Evaluamos:

```
es.par(14)
```

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

Tipos de Funciones: En general hemos revisado en lo que va del curso funciones Generales, Matemáticas y Estadísticas propiamente dichas. **Funciones Generales:** Como ejemplos tenemos las funciones familiarizadas de cbind(), rbind(), range(), sort(), order(), diff(), length() functions. **Funciones Matemáticas:** Como parte de estas funciones tenemos valor absoluto (abs(x)), logaritmo de x en base y (log(x, base=y)), exponencial de x (exp(x)), raíz cuadrada de x (sqrt(x)), factorial de x (fac(x)) entre otras. **Funciones Estadísticas:** Como parte de estas funciones tenemos media aritmética de x (mean(x)), mediana de x (median(x)), varianza de x (var(x)), desviación estándar de x (sd(x)), valores estándar (z-valores) de x (scale(x)), los cuantiles de x (quantile(x)), el sumario de x (summary(x)).

Alcance del Environment En R, el ambiente (environment) es una colección de objetos como funciones, variables, data frame, etc. R abre el ambiente cada vez que Rstudio es ejecutado. La parte más alta de este ambiente está disponible en global environment, llamado R_GlobalEnv. Y nosotros además tenemos un local environment. Nosotros podemos listar el contenido de nuestro actual ambiente con : `ls(environment())` y con `rm(list=ls())` se eliminan todas las variables del environment.

```
#Realizaremos los siguientes códigos para
#poder entender mejor lo anterior:
y<-10
f<-function(x){
  x+y}
f(5)
```

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

```
#Luego
f<-function(x){
  y<-10
  x+y
}
f(5)
```

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

```
#Finalmente
y <-2
f <- function(x){
y <- 4
x+y
}
f(5)
```

```
## [1] 9
```

#Ejemplo más estructurado:

```
es.divisible.por <-function(entero.grande, entero.chico){
  if(entero.grande %% entero.chico!=0)
    return(FALSE)
    return(TRUE)
}
es.divisible.por(15,6); es.divisible.por(15,5)
```

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

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

#Reescribiéndo la función es.par de manera más simple

```
es.par <- function(num){
  es.divisible.por(num,2)
}
```

Aplicación de Funciones

Las funciones conocidas dentro del entorno R son *apply*, *lapply*, **sapply**, **tapply**, *mapply* y *vapply*. Sin embargo, más adelante se utilizara **tidyverse** que tiene toda esta potencialidad incorporada.

- **Apply**: Aplica una función a una matriz, lista o vector que se le pase como parámetro. El código presenta la siguiente estructura:

$$apply(x, \text{MARGIN}, \text{FUN})$$

Argumentos:

`x` = Un arreglo o matriz. `MARGIN` = tomo un valor o un rango entre 1 y 2 para definir donde aplicar la función:
`MARGIN=1` : la manipulación trabaja en las filas. `MARGIN=2` : la manipulación trabaja en las columnas.
`MARGIN=c(1,2)` : la manipulación trabaja en filas y columnas. `FUN`=nos indica las funciones que vamos a aplicar.
Construir funciones como `mean`, `median`, `sum`, `min`, `max` e incluso funciones creadas por el usuario pueden ser aplicadas.

```
#Realizaremos los siguientes códigos  
#para poder entender lo anterior:  
mi_matriz <- matrix(1:9,nrow=3,ncol=3)  
apply(mi_matriz, 2, sum)
```

```
## [1] 6 15 24
```

```
apply(mi_matriz,1, sum)
```

```
## [1] 12 15 18
```

```
apply(mi_matriz, c(1,2),mean)
```

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

#Otro ejemplo

```
m1 <- matrix(c<-(1:10), nrow=5,ncol=6)  
m1
```

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

```
a_m1 <- apply(m1,2,sum)  
a_m1
```

```
## [1] 15 40 15 40 15 40
```

```
m1[a_m1]
```

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

- **lapply**: Se diferencia con **apply** en que opera con listas. Recibe una lista y devuelve una lista. El código presenta la siguiente estructura:

lapply(x, FUN)

Argumentos:

x = Un vector o un objeto. **FUN** = función aplicada para cada elemento de x.

```
#Realizaremos los siguientes códigos para poder entender mejor el uso de lapply
movies<-c("SPIDERMAN","BATMAN","VERTIGO","CHINATOWN")
movies_lower<-lapply(movies, tolower)
str(movies_lower)

## List of 4
## $ : chr "spiderman"
## $ : chr "batman"
## $ : chr "vertigo"
## $ : chr "chinatown"

movies_lower<-unlist(lapply(movies, tolower)) #
```

```
#Otro ejemplo
P<-matrix(1:9, nrow=3, ncol=3)
Q<-matrix(11:19, nrow=3, ncol=3)
R<-matrix(21:29, nrow=3, ncol=3)
mi_lista<-list(P,Q,R)
lapply(mi_lista, "[", 1, 1)
```

```
#Podemos intercambiar lapply() y sapply() para cortar un data frame.
below_ave <- function(x){
  ave <- mean(x)
  return(x[x>ave])
}
dt_s <- sapply(dt, below_ave)
dt_l <- lapply(dt, below_ave)
identical(dt_s,dt_l)
```

- **Tapply:** Realiza una operación (parámetro 3) respecto a un vector (parámetro 1) agrupada por los factores que se indiquen como argumento (parámetro 2). El código presenta la siguiente estructura:

$$tapply(x, INDEX, FUN)$$

Argumentos:

x = Un objeto, usualmente un vector.

INDEX= Una lista conteniendo factores.

FUN= Función aplicada a cada elemento de x.

```
#Realizaremos los siguientes códigos para poder entender lo anterior (ejemplo iris data frame):  
data(iris)  
tapply(iris$Sepal.Width,iris$Species, median)
```

```
##      setosa versicolor  virginica  
##      3.4          2.8          3.0
```

```
#Otro ejemplo  
x <- 1:20  
y <- factor(rep(letters[1:5],each=4))  
tapply(x, y, sum)
```

```
##  a  b  c  d  e  
## 10 26 42 58 74
```

- **Mapply**: Realiza operaciones entre matrices y devuelve una lista o vector, a continuación se muestran algunas de las operaciones que admite.

```
#Suma el primer elemento de cada vector, despues
mapply(sum, 1:5, 1:5, 1:5)
```

```
## [1] 3 6 9 12 15
```

```
#Repite cada elemento del primer vector el numero
mapply(rep, 1:4, 4:1)
```

```
## [[1]]
## [1] 1 1 1 1
##
## [[2]]
## [1] 2 2 2
##
## [[3]]
## [1] 3 3
##
## [[4]]
## [1] 4
```

- **Vapply**: Devuelve un vector con la longitud que tiene cada una de las listas introducidas como parámetro.

```
x <- list(A=1,B=1:3,C=1:7)
vapply(x, FUN= length, FUN.VALUE=0L)
```

```
## A B C
## 1 3 7
```

Administración de Paquetes e Instalación

Librerías:

R viene con un conjunto de características por defecto, tales como códigos básicos desarrollados, plantillas generales de trabajo, no obstante, las características de mayor ayuda corresponden a *módulos opcionales que se pueden descargar de la web e instalar en R*. Existen cientos de miles de paquetes los cuales tienen una amplia variedad de tópicos de área de toda índole, económica, ingeniería, ciencia, etc. Por lo cual debemos entender primero *que es un paquete*, **son colecciones de funciones en R**. Por ejemplo, la función `library()` muestra la lista de paquetes guardados en la librería. R, como se menciono anteriormente, viene con un conjunto de paquetes por default (stats, graphics, utils, methods y otros), ellos provienen de un rango de funciones y conjunto de datos que están disponibles de forma determinada. El comando `search()` indica que paquetes están cargados y listos a usar, por ejemplo:

```
#usar la funcion
search()
```

```
## [1] ".GlobalEnv"          "package:xaringantheme" "package:stats"
## [4] "package:graphics"      "package:grDevices"      "package:utils"
## [7] "package:datasets"       "package:methods"       "Autoloads"
## [10] "package:base"
```

CRAN (Comprehensive R Archive Network)

CRAN es una red de servidores ftp y web en todo el mundo que almacena versiones idénticas y actualizadas de código y documentación para R. Siempre es bueno utilizar el espejo CRAN más cercano para minimizar la carga de la red en la instalación de paquetes. Para instalar paquetes desde CRAN podemos hacerlo de dos formas. No olvidar instalar primero [Rtools](#).

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

- Abrir Rstudio [Tools](#) -> [Install Packages](#) -> [Name Package](#) -> [Install](#);¹

Lista de paquetes importantes a instalar:

Ver <https://roelverbelen.netlify.app/resources/r/packages/>.

GitHub (Internet hosting for software development)

Github es un proveedor de alojamiento de Internet para el desarrollo de software y el control de versiones mediante Git. Ofrece el control de versiones distribuido y la funcionalidad de administración de código fuente de Git, además de sus propias características. Millones de desarrolladores y compañías construyen, interactúan y mantienen su software en GitHub - es una más grande y más avanzada plataforma de desarrollo en mundo. Creada inicialmente por **Chris Wanstrath, P. J. Hyett, Tom Preston-Werner and Scott Chacon** usando Ruby on Rails, y comenzó en Febrero del 2008. Para instalar paquetes desde GitHub:

```
install.packages("remotes", "devtools")
library(remotes)
library(devtools)
install_github("AotinianoZ/foofactors")
library(foofactors)
```

Revisar <https://github.com/AotinianoZ/foofactors>.

Bioconductor(Open Source Software for Bioinformatics)

Bioconductor proporciona herramientas para el análisis y la comprensión de datos genómicos de alto rendimiento. Bioconductor utiliza el lenguaje de programación estadístico R y es de código abierto y desarrollo abierto. Tiene dos lanzamientos cada año y una comunidad de usuarios activa. Bioconductor también está disponible como [AMI](#) (Imagen de máquina de Amazon) e imágenes de [Docker](#).

```
# Infinidad de paquetes auxiliares útiles para ciencia de datos.
install.packages("BiocManager")
library(BiocManager)
```

Bibliografía Adicional:

La bibliografía para comenzar en el fascinante mundo de R es:

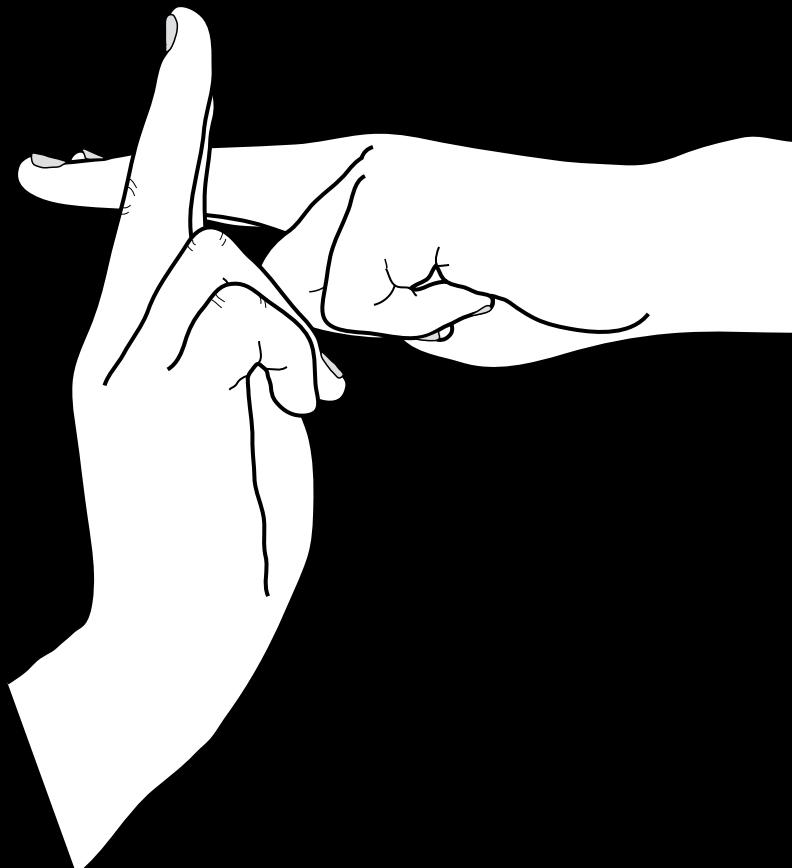
- R for Data Science, ver [R4DS, Hands-On Programming with R](#).
- Colección de Paquetes Data Science, ver [Tidyverse](#) y [listado de paquetes](#).
- Generación de Libros Interactivos, documentos y análisis, ver [R Markdown Cookbook](#), [R Markdown: The Definitive Guide](#), [bookdown](#), [Mastering Shiny](#), [Interactive web-based data visualization with R](#), [plotly](#), and [shiny](#)
- Rstudio y R, ver [Rstudio](#), [Studio Blog](#), [useR](#).
- Journals - Books, ver [The R Journal](#), [Chapman & Hall/CRC The R Series](#), [Serie Use R](#).
- Machine Learning, ver [Hands-On Machine Learning with R](#).
- Geoquímica, ver [Geochemical Modelling of Igneous Processes – Principles And Recipes in R Language](#), paquetes importantes Geological Survey of Canada - [rgr:Applied Geochemistry Data EDA](#) y [USGS - GcClust: Clustering regional geochemical data](#)
- Hidroquímica, ver [CRAN Task View: Hydrological Data and Modeling](#).

Terminamos a Descansar!!!



Avanzando en R y Geociencias (Ep)

Siempre Adelante



Thanks!

Slides son creados via el R package **xaringan**.

El chacra proviene de **remark.js**, **knitr**, y R Markdown.