**2020/01/24**

**INTRODUCCIÓN A R**

**Fátima Sánchez Cabo**

Bioinformática Data Science. Matemática. Análisis de datos genómicos.

Dr Fuster, dirige el CNIC. Fátima dirige la unidad de bioinformática.

R nació en el año 1998. Era un lenguaje matricial y desde el 2017 se está intentando mover a un leguaje más orientado a objetos.

Una de las cosas buenas de R es que tiene muchísimas funciones o paquetes desarrollados.

**Tipos de archivo**

.**rmd** = R markdown -> a partir de él puedo genera pdf, html, y más. Este formato puede intercalar texto y código, como los notebooks.

**.r** -> Los scripts planos (sin texto libre) son formato .r

**Bibliografía**

*Hands-on programming with R*

*Data Science with R*

Son libros buenos de consulta, gratis en versión electrónica. Tienen un github asociado donde podemos descargar el cógido y ejemplos usados.

vignette() es la documentación de los paquetes…

**Qué es R**

Es un lenguaje interpretado: no necesita compilación

No está hecho por informáticos, sino por estadísticos y analistas de datos. Es open source.

El código fuente está en R, en Fortran y C. Al estar en R podemos descargar, meternos dentro de cualquier función, entender cómo está hecha y modificarlo si queremos.

Puntos fuertes:

- Hace buenos gráficos. Aunque es verdad que ahora python con matplotlib hace gráficos muy buenos.

- Gran comunidad detrás desarrollando los paquetes.

- Es intereactivo. No hace falta compilarlo para ejecutarlo, se puede ejecutar según se escribe.

Características:

- La funciones de R son *objetos* de R. Tienen su propio *tipo.*

- La distribución básica de R trae 8 paquetes: stats, graphics, grDevices, utils, datasets, methods, base.

**Paquetes**

CRAN: Comprehensive R Archive Network. Aquí es donde se suben todos los paquetes. Todos podemos desarrollar un paquete y subirlo ahí aunque tienen unos mínimos requerimientos (documentación, normas de escritura del código).

Dentro de CRAN hay un subconjunto de paquetes llamado Bioconductor.

También hay paquetes, no supervisados, fuera de CRAN que cualquiera puede publicar en su web o su github.

**Paquetes importantes para trabajar**

**devtools**

**xlsx** -> para importar excel

**tidyverse** -> paquete muy importante. Crea objetos tibble.

**data.table** -> para crear data.frames

**dslabs**

**rvest**

**RMySQL**

**RmariaDB** -> para conectarnos al servidor de bases de datos de MariaDB.

**dbplyr**

**WDI** -> para bajar datos del World Bank, el cual contiene World Development Indicators

**Getting Started**

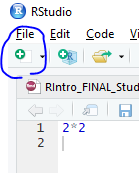
Abrimos Rstudio desde Anaconda de windows. También se podría instalar Rstudio para windows independiente de anaconda.

Lo primero es decirle a R dónde estamos trabajando, osea nuestro directorio base donde tenemos el archivo de trabajo. En *Session > Set Working Directory > choose directory*

Se escribe automáticamente una línea de código así:

setwd("C:/Users/usuario/AppData/Local/Packages/CanonicalGroupLimited.UbuntuonWindows\_79rhkp1fndgsc/LocalState/rootfs/home/angelrps/git/MasterDataScience/200124\_R\_Import\_Collection\_Of\_Data")

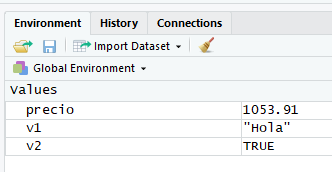
Podemos abrir un nuevo R Script aquí. Un Script es un simple texto.



Para ejecutar una linea del R Script en la consola de abajo pulsamos ctrl + intro

En un rmd (R markdown) donde se mezcla texto y código se ejecutan los chunks de código ahí mismo.

En Environment irán apareciendo todos los objetos (variables) que iremos creando en la sesión.



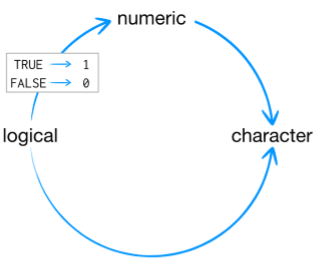
Los objetos se van guardando en memoria RAM.

Se pueden asignar variable con <- o con =. Se suele usar <-

Los objetos en R pueden contener diferentes tipos de datos.

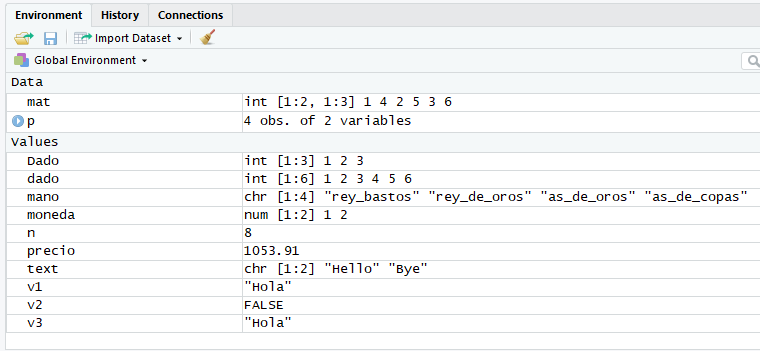
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **TIPOS DE OBJETO** | **INDEXACIÓN**  (comienza por 1) | **DATOS DE MISMO TIPO** |  | **TIPOS DE DATO** |
| 1D **vector**  c( , , , …) | [ ] 1 índice | sí |  | **Numeric**: integer or double |
| 2D **matriz** | [ , ] 2 índices | sí (si no, R los transforma) |  | **Character** |
| nD **array** | [ , , , , …] n índices | sí |  | **Logical** (TRUE, FALSE) |
| **list** | [[1]] doble corchete  objetos diferente tamaño | no |  | **Factor**(para definir categorías) |
| **data.frame** | df$variable[ ] | no |  |  |
| **tibble** | mejora del data.frame | no |  |  |

Cuando R se ve obligado a transformar un tipo de dato a otro, sigue este criterio:



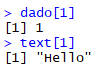
**NOTA SOBRE TIPOS:**

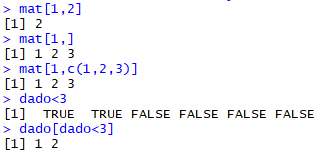
los vectores de una dimensión, ya seaen num, int, char, o lógicos, se agrupan en *values.* Las matrices y los data frames se agrupan en *Data.*

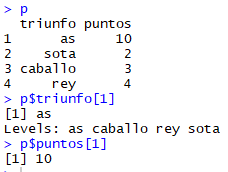


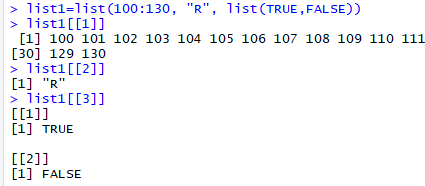
Factores: ojo, R por detrás los almacena numéricamente osea que si los quiero ordenar creo que no lo haría alfabéticamente.

**Ejemplos de indexación**





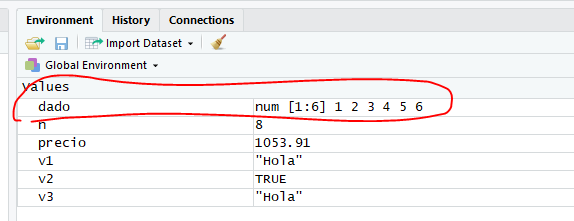




> dado <- c(1,2,3,4,5,6)

> dado <- 1:6

Los dos de arriba crean la misma variable. A veces te pone num, otras, int porque no maneja bien los tipos. Si queremos que entienda forzamente los números como int tenemos que crear así la variable



Para crear un vector siempre se usa c(). La C viene de *concatenar.*

> dado <- c(1L,2L,3L,4L,5L,6L)

> length(dado)

para interrogar el tipo de objeto

> is.numeric(dado)

[1] TRUE

> class(dado)

[1] "integer"

> typeof(Dado)

[1] "integer"

operaciones elemento a elemento

> dado\*2

[1] 2 4 6 8 10 12

> dado/2

[1] 0.5 1.0 1.5 2.0 2.5 3.0

> dado+2

[1] 3 4 5 6 7 8

> dado/dado

[1] 1 1 1 1 1 1

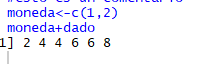
operación matricial

> dado%\*%dado

[,1]

[1,] 91

Cuidado! Si sumas dos vectores de diferente rango R te los suma añadiendo el vector menor tantas veces como sea necesario para llegar al final

Es algo muy raro, CUIDADO!!

**La precisión numérica de R son 16 dígitos.** Ojo por si a veces hay problemas de precisión de cálculos.

NOTA: en la vida real cuando trabajemos en R es mejor trabajar en Rmd en lugar de un R script.



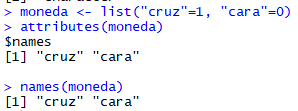
**NOTAS**:

- mejor no poner espacios porque a veces R da problemas añadiendo un punto. Así que ponemos barra baja.

Se pueden sumar vectores lógicos. TRUE sería 1 y FALSE 0

**Atributos**

Parecido a los diccionarios. Se pueden añadir atributos a los valores de una lista. Luego se pueden llamar a esos atributos usando **attributes**(lista). En el caso de abajo, el identificador que hemos añadido a los valores de la lista se llama names (en lugar de *key*) y se llaman con **names**(lista)



**Funciones**

La sintaxis es:

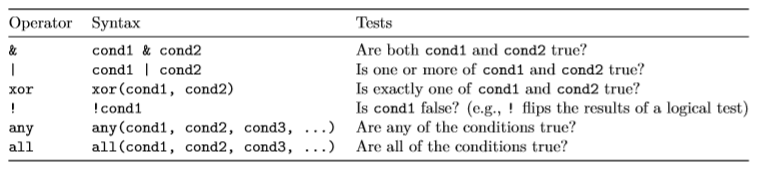
functionName <- function(){

return

}

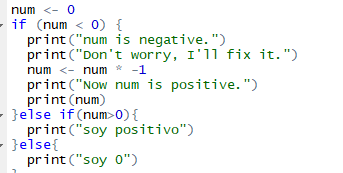
.Rdata es un *objeto* que guarda el workspace que incluye las variables que he usado. Normalmente no lo guardamos.

**Operadores booleanos**



**2020/01/25**

**Conditional Statements**



**NOTAS**:

Con source(‘nombredelscript.r) se ejecuta un archivo que hayamos guardado con extensión .r

**Instalación de paquetes**

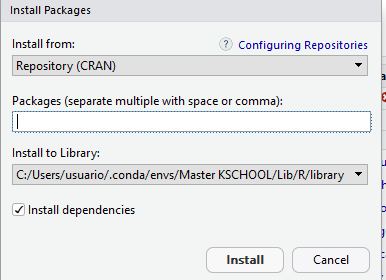
- Si están en CRAN uso: **install.packages**(nombre del paquete)

o varios a la vez metiendo un vector con los nombres de los paquetes:

**install.packages**(c(nombre1, nombre2, nombre3, …))

- Si están en GitHub uso: **install\_github**()

Desde los menús: Tools > Install Packages



Install simplemente lo descarga al ordenador pero una vez se han instalado los paquetes (guardados en tu ordenador) hay que hacerlos disponibles con la función **library**(nombredelpaquete)

**sessioninfo**() nos da información sobre la sesion incluido los paquetes que hay disponibles

**Useful functions**

**getwd**() -> te dice cuál es el working directory actual

**typeof**() -> te da el tipo de dato

**class**() -> te da el tipo de objeto

**is.numeric**()

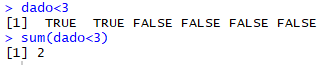
**is.character**()

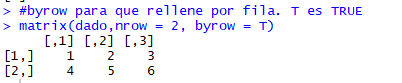
**length**()

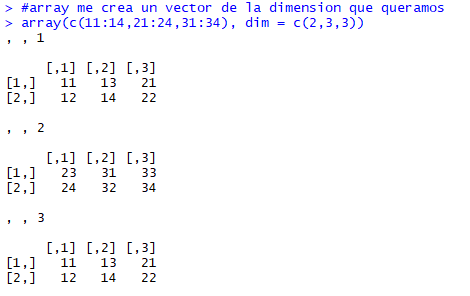
**Transformar vector numércio en matriz o array:**

**matrix**(dado, nrow=2, byrow=T)

**array**(c(12:14, 21:24, 31:34), dim = c(2, 2, 3))



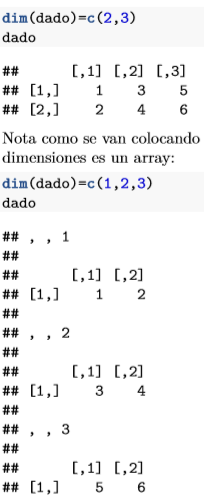
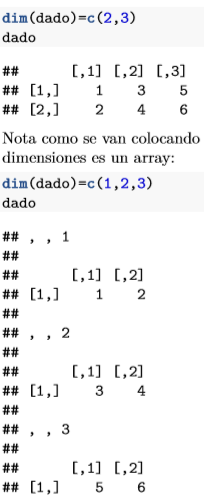




**cbind**(vector1, vector2, vector3,…) pegar vectores por columna

**rbind**(vector1, vector2, vector3,…) pegar vectores por fila

**dim**() <- para cambiar la dimensión de un vector unidimensional a una matriz o array.



**Convertir de un tipo a otro**

**as.charater()**

**as.numeric()**

**as.logical()**

**max**() -> máximo valor de una serie

**min**() -> mínimo valor de una serie

**which.max**() -> índice del máximo valor de una serie

**which.min**() -> índice del mínimo valor de una serie

which() -> devuelve los TRUE indices de una logical statement.

**na.rm** -> remove NA values from a vector. i.e. below:

si no fuera por na.rm no hubiera podido calcula la media.

**is.na**(vec) -> devuelve un lógico indicando si algún componente del vector es NA.

**NOTAS**:

Las principales funciones de sumarización de R devuelven NA si hay un NA en los datos. Por eso es muy importante gestionar estos valores antes.

Para ignorar los valores perdidos podemos usar na.rm así

**%in%** -> used to identify if an element is in a vector.

**1 %in% c(1,2,3,4,5)** will return TRUE

**sample**(x, size, replace = FALSE, prob = NULL) -> takes a sample of specified size from the elements of x using either with or without replacement.

**expand.grid**(wheel, wheel, wheel) -> Create a data frame from all combinations of the supplied vectors or factors. El ejemplo crearía un data frame con todas las combinaciones posibles de la máquina tragaperras del ejercicio.

***Leer ficheros***

**read.delim**("DataSets/murders.csv", sep=",", header=T)

lee cualquier tipo de archivo de texto plano (csv, tsv, txt, …)

**read.table**() -> lo mismo que read.delim

**fread**("DataSets/test.csv") -> hace lo mismo que read.delim pero más rápido. Bueno para leer ficheros grandes.

**read.csv**("DataSets/murders.csv", stringsAsFactors = F)

es una particularización de read.delim que viene son sep=',' y header=T. Si el separador fuera otro habría que especificar el separador. IMPORTANTE: poner stringsAsFactors = F para que no convierta strings a factors.

**read.xlsx**("DataSets/murders.xlsx", sheetIndex = 1)

Para leer archivos excel. Pertenece al paquete  *xlsx*.

**NOTAS**:

Los tres métodos read de arriba te devuelven un data frame.

**head**(mydataframe) -> muestra 6 primera líneas del data frame

**str**(mydataframe) -> muestra columnas en filas e información adicional como nº de filas.

**View**(mydataframe) -> abre el data frame en un visor nuevo con estilo de tabla.

**write.table**(DT,

"DataSets/test.csv",

sep=",",

row.names=FALSE,

quote=FALSE)

Crea un fichero CSV a partir de un data frame. Si no pones quote=FALSE te pone comillas en cada elemento.

Estos tres de abajo son igual que los de antes pero con guión bajo. Pertenecen al paquete tidyverse y te devuelven un objeto ***tibble***

**read\_delim**

**read\_csv** ->ojo te puede leer un csv localizado en una url tipo github, read\_csv(url)

**read\_xlsx**

**… and more**

df <- **mutate**(df, column1 = as.factor(column1)

Mutate ha transformado el typo de datos de la column1 de character a factor.

**tibble vs data.frame**:

- Tibble nunca cambia el tipo de una variable.

- Tibble nunca cambia el nombre de una variable cuando contiene espacios.

- Subsetting siempre devuelve otro objeto tbble, sea de las dimensiones que sea.

- Si una columnas no tiene exactamente el nombre por el que se la llama tibble no hace partial matching.

**download.file**(url, “NombreDeFicheroADescargar”) -> descargar un fichero de internet.

**2020/01/31**

**source**(“fichero .r”) -> ejecuta lo que haya dentro del script seleccionado

**Rscript** fichero.r -> para ejecutar scripts desde la linea de comandos

vamos a trabajar con el universo tidyverse

**paquete dplyer**

%>% es un pipe específico de algunos paquetes. Lo usamos con tidyverse

**filter**() -> seleccionar filas

a filter() hay que pasarle un vector logico para ver qué filas nos queremos quedar. Por eso el vector de logicos tendrá que tener el mismo tamaño que las filas del dataframe

**select**() -> seleccionar columnas

select(-year) <- con un menos quito la columna “year”

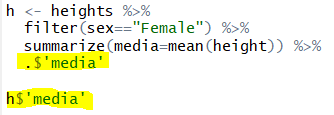
**mutate**() <- añadir columna nueva o hacer una operación en una columna existente.

En todas las funciones de tidyverse primero hay que pasarles el objeto, ya sea como un pipe o como el primer parámetro dentro del paréntesis. Usando pipes los objetos no se guardan sino que se van pasando on the fly y por eso a nivel de memoria es más rápido y eficiente.

**setNames**() -> set names of an object with “names” attribute. Por ejemplo, puedes renombrar las columnas de un data frame:

mytable %>% setNames(c("uno", "dos", "tres", "cuatro", "cinco", "seis", "siete"))

Las funciones de dplyr devuelven siempre el mismo tipo de objeto. Cuando después de un filtrado el resultado final es un número y queremos obtener un objeto “número”, no un data frame, hacemos las dos siguientes opciones:



- Añadir .$’nombreColumna’ al final del pipe

- Llamar a la columna usando single cuotes

**sumarize**() para calcular cualquier función después de un pipe. ejemplo:

algo%>%sumarize(mean)

ese “algo” tiene que ser exactamente el input que está esperando la función, si no, fallará.

**add\_row**()

**arrange**(murder\_rate) -> ordenar tabla según variable murder\_rate

**arrange**(desc(murder\_rate)) -> ordenar tabla según variable murder\_rate en orden descendiente

**arrange**(continent, murder\_rate) -> ordenar tabla según variable continent y luego murder\_rate

arrange(**desc**(mean.rate)) <- ordena en orden descendente según variable mean.rate

**top\_n**(10) -> es lo mismo que head() pero le puedes poner el número de filas.

group\_by()

**case\_when**

x <- c(-2, -1, 0, 1, 2)

case\_when(x < 0 ~ "Negative",

x > 0 ~ "Positive",

TRUE ~ "Zero") # al final pongo true que es lo mismo que decir "el resto de los casos"

**between**(x, a, b) -> true or false si x está entre a y b.

***Tidy* Data**

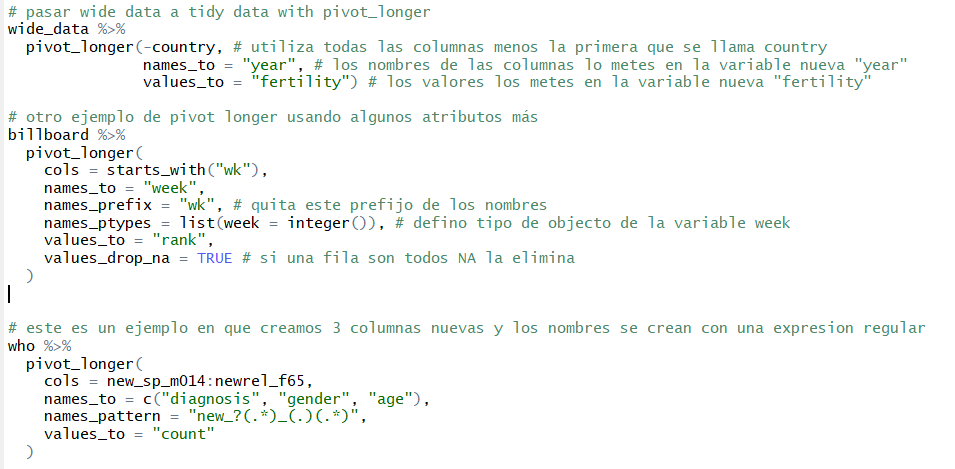
Para trabajar en el universo tidyverse necesitamos trabajar con un tipo de datos llamados datos “tidy”. Son necesarios para trabajar con ggplot2.

En el formato Tidy, cada variable debe ser una columna, y cada fila una observación de datos.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **formato WIDE** | | | | | | | |
| **country** | **1960** | **1962** | **1963** | **1964** | **1965** | **1966** | … |
| spain | 8.5 | 8.6 | 8.4 | 8.5 | 8.4 | 8.7 | … |
| Italy | 9.1 | 9.2 | 9.3 | 9.0 | 9.5 | 9.3 | … |
| … | … | … | … | … | … | … | … |

|  |  |
| --- | --- |
| **formato TIDY** | |
| **country** | **year** |
| spain | 1960 |
| spain | 1961 |
| spain | 1962 |
| spain | 1963 |
| spain | 1964 |
| spain | 1965 |
| spain | 1966 |
| … | … |
| Italy | 1960 |
| Italy | 1961 |
| Italy | 1962 |
| Italy | 1963 |
| Italy | 1964 |
| Italy | 1965 |
| Italy | 1966 |
| … | … |

**pivot\_longer()** para pasar de wide ---> tidy



**pivot\_wider()** para pasar de tidy ---> wide

Atributos similares a pivot\_longer() pero poniendo *names\_from* y *values\_from*

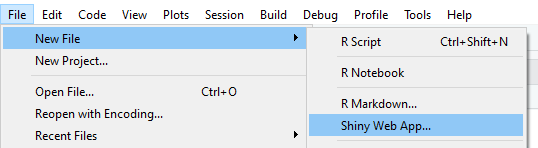
**2020/02/01 Visualización de datos con R**

**NOTA:**

si al cargar una librería nos sale esto significa que por ejemplo la función filter() está repetida en el paquete dplyr y stats. Por defecto usará la de la librería que cargamos la última.



En caso de conflicto para asegurarnos que utilizamos la función correcta debemos utilizar esta nomenclatura: dplyr::filter()



**Shiny Web App**, es un formato de archivo que permite crear paginas web con nuestros graficos de R integrados. Tiene buena pinta, exlorarlo. Si hay 2 millones de variables dice la profe que va un poco lento.

con par(new=T) entiende que no haga otro plot después sino que pinte en el mismo que está

plot(murders$total,murders$population/10^5,pch="\*",col="red")

points(murders$total,murders$population/10^5,lty=2)

abline(a=0,b=1,lty=2)

points añade puntitos encima del grafico anterior sin necesidad de usar par(new=T)

abline es igual que points pero añade una linea

table(murders$bin) cuanta los numeros de elementos iguales que hay en un vector